
AWS Lambda

Developer Guide



AWS Lambda: Developer Guide

Copyright © 2023 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is AWS Lambda?	1
When to use Lambda	1
Key features	2
Getting started	3
Prerequisites	3
Create a function	4
Invoke the function	4
Clean up	5
Additional resources	6
Accessing Lambda	6
Authoring and deploying functions	6
Lambda foundations	8
Concepts	9
Function	9
Trigger	9
Event	9
Execution environment	10
Instruction set architecture	10
Deployment package	10
Runtime	10
Layer	11
Extension	11
Concurrency	11
Qualifier	11
Destination	12
Programming model	13
Execution environment	14
Runtime environment lifecycle	14
Deployment packages	18
Container images	18
.zip file archives	18
Layers	19
Using other AWS services	19
Lambda console	21
Applications	21
Functions	21
Code signing	21
Layers	21
Edit code using the console editor	21
Instruction sets (ARM/x86)	29
Advantages of using arm64 architecture	29
Function migration to arm64 architecture	29
Function code compatibility with arm64 architecture	30
Suggested migration steps	30
Configuring the instruction set architecture	30
Additional features	32
Scaling	32
Concurrency controls	32
Function URLs	32
Asynchronous invocation	33
Event source mappings	33
Destinations	34
Function blueprints	35
Testing and deployment tools	35

Application templates	36
Learn how to build serverless solutions	36
Lambda runtimes	37
Runtime deprecation policy	39
Runtime updates	41
Runtime management controls	41
Two-phase runtime version rollout	42
Roll back a runtime version	42
Identifying runtime version changes	43
Configure runtime management settings	45
Shared responsibility model	46
High-compliance applications	47
Supported Regions	47
Runtime modifications	49
Language-specific environment variables	49
Wrapper scripts	51
Runtime API	54
Next invocation	54
Invocation response	55
Initialization error	55
Invocation error	56
Custom runtimes	59
Using a custom runtime	59
Building a custom runtime	59
Implementing response streaming in a custom runtime	60
Tutorial – Custom runtime	62
Prerequisites	62
Create a function	62
Create a layer	64
Update the function	65
Update the runtime	66
Share the layer	66
Clean up	66
AVX2 vectorization	68
Compiling from source	68
Enabling AVX2 for Intel MKL	68
AVX2 support in other languages	68
Configuring functions	70
Configuring function options	71
Function versions	71
Using the function overview	71
Configuring functions (console)	72
Configuring functions (API)	73
Configuring function memory (console)	73
Configuring function timeout (console)	73
Configuring ephemeral storage (console)	74
Accepting function memory recommendations (console)	74
Configuring triggers (console)	74
Testing functions (console)	75
Environment variables	76
Configuring environment variables	76
Configuring environment variables with the API	77
Example scenario for environment variables	77
Retrieve environment variables	78
Defined runtime environment variables	79
Securing environment variables	80
Sample code and templates	82

Versions	83
Creating function versions	83
Using versions	84
Granting permissions	84
Aliases	85
Creating a function alias (Console)	85
Managing aliases with the Lambda API	85
Managing aliases with AWS SAM and AWS CloudFormation	86
Using aliases	86
Resource policies	86
Alias routing configuration	86
Private networking	89
VPC network elements	89
Connecting Lambda functions to your VPC	90
Lambda Hyperplane ENIs	90
Connections	91
Security	91
Observability	92
Creating layers	93
Creating layer content	93
Compiling the .zip file archive for your layer	93
Including library dependencies in a layer	94
Language-specific instructions	95
Creating a layer	95
Deleting a layer version	97
Configuring layer permissions	97
Using AWS CloudFormation with layers	97
Response streaming	99
Writing response streaming-enabled functions	99
Invoking a response streaming enabled function using Lambda function URLs	100
Tutorial: Creating a response streaming function with a function URL	101
Deploying functions	106
.zip file archives	106
Container images	106
.zip file archives	107
Creating the function	107
Using the console code editor	108
Updating function code	108
Changing the runtime	109
Changing the architecture	109
Using the Lambda API	73
AWS CloudFormation	110
Container images	111
Prerequisites	111
Permissions	112
Creating the function	113
Testing the function	114
Overriding container settings	115
Updating function code	115
Using the Lambda API	116
AWS CloudFormation	116
Invoking functions	118
Synchronous invocation	120
Asynchronous invocation	123
How Lambda handles asynchronous invocations	123
Configuring error handling for asynchronous invocation	125
Configuring destinations for asynchronous invocation	125

Asynchronous invocation configuration API	128
Dead-letter queues	129
Event source mapping	131
Batching behavior	132
Event filtering	136
Event filtering basics	136
Handling records that don't meet filter criteria	138
Filter rule syntax	138
Attaching filter criteria to an event source mapping (console)	139
Attaching filter criteria to an event source mapping (AWS CLI)	140
Attaching filter criteria to an event source mapping (AWS SAM)	141
Using filters with different AWS services	141
Filtering with DynamoDB	142
Filtering with Kinesis	145
Filtering with Amazon MQ	147
Filtering with Amazon MSK and self-managed Apache Kafka	152
Filtering with Amazon SQS	155
Function states	159
Function states while updating	159
Error handling	161
Testing functions	163
Private test events	163
Shareable test events	163
Invoking functions with test events	164
Deleting shareable test event schemas	164
Invoking functions defined as container images	165
Function lifecycle	165
Invoking the function	165
Image security	165
Function URLs	166
Creating and managing function URLs	167
Security and auth model	172
Invoking function URLs	177
Monitoring function URLs	184
Tutorial: Creating a function with a function URL	186
Managing functions	190
Tutorial - Lambda with CLI	191
Prerequisites	191
Create the execution role	191
Create the function	192
Update the function	195
List the Lambda functions in your account	195
Retrieve a Lambda function	196
Clean up	196
Function scaling	197
Understanding and visualizing concurrency	197
How to calculate concurrency	200
Reserved concurrency and provisioned concurrency	201
Concurrency quotas	207
Accurately estimating required concurrency	208
Concurrency metrics	208
Configuring reserved concurrency	210
Configuring provisioned concurrency	213
Burst concurrency	220
Networking	222
Managing VPC connections	222
Execution role and user permissions	223

Configuring VPC access (console)	223
Configuring VPC access (API)	224
Using IAM condition keys for VPC settings	225
Internet and service access for VPC-connected functions	228
VPC tutorials	228
Sample VPC configurations	228
Interface VPC endpoints	229
Considerations for Lambda interface endpoints	229
Creating an interface endpoint for Lambda	230
Creating an interface endpoint policy for Lambda	230
Database	232
Creating a database proxy (console)	232
Using the function's permissions for authentication	233
Sample application	233
File system	236
Execution role and user permissions	236
Configuring a file system and access point	236
Connecting to a file system (console)	237
Configuring file system access with the Lambda API	238
AWS CloudFormation and AWS SAM	238
Sample applications	240
Code signing	241
Signature validation	241
Configuration prerequisites	242
Creating code signing configurations	242
Updating a code signing configuration	242
Deleting a code signing configuration	243
Enabling code signing for a function	243
Configuring IAM policies	243
Configuring code signing with the Lambda API	244
Using layers	245
Prerequisites	245
Configuring functions to use layers	245
Accessing layer content from your function	248
Finding layer information	248
Using AWS SAM to add a layer to a function	97
Sample applications	249
Tags	250
Permissions	250
Using tags with the console	250
Using tags with the AWS CLI	252
Requirements for tags	253
Building with Node.js	254
Node.js initialization	256
Designating a function handler as an ES module	256
Handler	258
Using <code>async/await</code>	258
Using callbacks	260
Deploy .zip file archives	263
Prerequisites	263
Updating a function with no dependencies	263
Updating a function with additional dependencies	264
Deploy container images	266
AWS base images for Node.js	266
Using a base image	267
Node.js runtime interface clients	270
Context	271

Logging	273
Creating a function that returns logs	273
Using the Lambda console	274
Using the CloudWatch console	274
Using the AWS Command Line Interface (AWS CLI)	274
Deleting logs	277
Errors	278
Syntax	278
How it works	278
Using the Lambda console	279
Using the AWS Command Line Interface (AWS CLI)	279
Error handling in other AWS services	280
What's next?	281
Tracing	282
Using ADOT to instrument your Node.js functions	282
Using the X-Ray SDK to instrument your Node.js functions	282
Activating tracing with the Lambda console	283
Activating tracing with the Lambda API	284
Activating tracing with AWS CloudFormation	284
Interpreting an X-Ray trace	284
Storing runtime dependencies in a layer (X-Ray SDK)	286
Building with TypeScript	288
Development environment	288
Handler	290
Using async/await	290
Using callbacks	291
Using types for the event object	291
Deploy .zip file archives	293
Using AWS SAM	293
Using the AWS CDK	294
Using the AWS CLI and esbuild	296
Deploy container images	298
Using a base image	298
Context	302
Logging	304
Tools and libraries	304
Using AWS Lambda Powertools for TypeScript and AWS SAM for structured logging	304
Using AWS Lambda Powertools for TypeScript and the AWS CDK for structured logging	306
Using the Lambda console	309
Using the CloudWatch console	309
Errors	310
Tracing	312
Using AWS Lambda Powertools for TypeScript and AWS SAM for tracing	312
Using AWS Lambda Powertools for TypeScript and the AWS CDK for tracing	314
Interpreting an X-Ray trace	317
Building with Python	318
Handler	320
Naming	320
How it works	320
Returning a value	321
Examples	321
Deploy .zip file archives	324
Prerequisites	324
What is a runtime dependency?	325
Deployment package with no dependencies	325
Deployment package with dependencies	325
Using a virtual environment	326

Deploy your .zip file to the function	327
Deploy container images	329
AWS base images for Python	329
Create a Python image from an AWS base image	330
Create a Python image from an alternative base image	331
Python runtime interface clients	331
Deploy the container image	331
Context	332
Logging	334
Tools and libraries	334
Creating a function that returns logs	334
Using AWS Lambda Powertools for Python and AWS SAM for structured logging	335
Using AWS Lambda Powertools for Python and the AWS CDK for structured logging	338
Using the Lambda console	342
Using the CloudWatch console	342
Using the AWS Command Line Interface (AWS CLI)	343
Deleting logs	345
Logging library	345
Errors	346
How it works	346
Using the Lambda console	347
Using the AWS Command Line Interface (AWS CLI)	347
Error handling in other AWS services	348
Error examples	348
Sample applications	349
What's next?	281
Tracing	350
Using AWS Lambda Powertools for Python and AWS SAM for tracing	350
Using AWS Lambda Powertools for Python and the AWS CDK for tracing	338
Using ADOT to instrument your Python functions	356
Using the X-Ray SDK to instrument your Python functions	356
Activating tracing with the Lambda console	356
Activating tracing with the Lambda API	357
Activating tracing with AWS CloudFormation	357
Interpreting an X-Ray trace	357
Storing runtime dependencies in a layer (X-Ray SDK)	359
Building with Ruby	361
Handler	364
Deploy .zip file archives	365
Prerequisites	365
Tools and libraries	365
Updating a function with no dependencies	366
Updating a function with additional dependencies	366
Deploy container images	368
AWS base images for Ruby	368
Using a Ruby base image	369
Ruby runtime interface clients	369
Create a Ruby image from an AWS base image	369
Deploy the container image	370
Context	371
Logging	372
Creating a function that returns logs	372
Using the Lambda console	373
Using the CloudWatch console	373
Using the AWS Command Line Interface (AWS CLI)	373
Deleting logs	375
Logger library	376

Errors	377
Syntax	377
How it works	377
Using the Lambda console	378
Using the AWS Command Line Interface (AWS CLI)	378
Error handling in other AWS services	379
Sample applications	380
What's next?	380
Tracing	381
Enabling active tracing with the Lambda API	384
Enabling active tracing with AWS CloudFormation	384
Storing runtime dependencies in a layer	385
Building with Java	386
Handler	389
Choosing input and output types	390
Handler interfaces	391
Sample handler code	392
Deploy .zip file archives	393
Prerequisites	393
Tools and libraries	393
Building a deployment package with Gradle	394
Building a deployment package with Maven	395
Uploading a deployment package with the Lambda console	396
Uploading a deployment package with the Lambda API	397
Uploading a deployment package with AWS SAM	398
Deploy container images	400
AWS base images for Java	400
Using a Java base image	401
Java runtime interface clients	406
Context	407
Context in sample applications	408
Logging	410
Tools and libraries	410
Creating a function that returns logs	410
Using AWS Lambda Powertools for Java and AWS SAM for structured logging	412
Using AWS Lambda Powertools for Java and the AWS CDK for structured logging	415
Using the Lambda console	423
Using the CloudWatch console	423
Using the AWS Command Line Interface (AWS CLI)	423
Deleting logs	425
Advanced logging with Log4j 2 and SLF4J	425
Sample logging code	427
Errors	429
Syntax	429
How it works	430
Creating a function that returns exceptions	430
Using the Lambda console	431
Using the AWS Command Line Interface (AWS CLI)	432
Error handling in other AWS services	432
Sample applications	433
What's next?	433
Tracing	434
Using AWS Lambda Powertools for Java and AWS SAM for tracing	434
Using AWS Lambda Powertools for Java and the AWS CDK for tracing	436
Using ADOT to instrument your Java functions	444
Using the X-Ray SDK to instrument your Java functions	444
Activating tracing with the Lambda console	445

Activating tracing with the Lambda API	445
Activating tracing with AWS CloudFormation	445
Interpreting an X-Ray trace	446
Storing runtime dependencies in a layer (X-Ray SDK)	447
X-Ray tracing in sample applications (X-Ray SDK)	448
Tutorial - Eclipse IDE	449
Prerequisites	449
Create and build a project	449
Sample apps	452
Building with Go	453
Handler	454
Naming	455
Lambda function handler using structured types	455
Using global state	456
Context	458
Accessing invoke context information	458
Deploy .zip file archives	460
Prerequisites	460
Tools and libraries	460
Sample applications	460
Creating a .zip file on macOS and Linux	461
Creating a .zip file on Windows	461
Creating a Lambda function using a .zip archive	462
Build Go with the provided.al2 runtime	463
Deploy container images	465
AWS base images for Go	465
Go runtime interface clients	466
Using the Go:1.x base image	466
Create a Go image from the provided.al2 base image	469
Create a Go image from an alternative base image	470
Logging	473
Creating a function that returns logs	473
Using the Lambda console	474
Using the CloudWatch console	474
Using the AWS Command Line Interface (AWS CLI)	474
Deleting logs	477
Errors	478
Creating a function that returns exceptions	478
How it works	478
Using the Lambda console	479
Using the AWS Command Line Interface (AWS CLI)	479
Error handling in other AWS services	480
What's next?	481
Tracing	482
Using ADOT to instrument your Go functions	482
Using the X-Ray SDK to instrument your Go functions	482
Activating tracing with the Lambda console	483
Activating tracing with the Lambda API	483
Activating tracing with AWS CloudFormation	483
Interpreting an X-Ray trace	484
Environment variables	486
Building with C#	487
Handler	489
Handling streams	489
Handling standard data types	490
Handler signatures	492
Using top-level statements	492

Using Lambda Annotations (Preview)	493
Serializing Lambda functions	494
Lambda function handler restrictions	495
Using <code>async</code> in C# functions with Lambda	495
Deployment package	497
.NET Core CLI	497
AWS Toolkit for Visual Studio	500
Deploy container images	502
AWS base images for .NET	502
Using a .NET base image	503
.NET runtime interface clients	504
Context	505
Logging	506
Tools and libraries	506
Creating a function that returns logs	506
Using log levels	508
Using AWS Lambda Powertools for .NET and AWS SAM for structured logging	508
Using the Lambda console	510
Using the CloudWatch console	510
Using the AWS Command Line Interface (AWS CLI)	511
Deleting logs	513
Errors	514
Syntax	514
How it works	516
Using the Lambda console	517
Using the AWS Command Line Interface (AWS CLI)	517
Error handling in other AWS services	518
What's next?	518
Tracing	519
Using AWS Lambda Powertools for .NET and AWS SAM for tracing	519
Using the X-Ray SDK to instrument your .NET functions	521
Activating tracing with the Lambda console	522
Activating tracing with the Lambda API	522
Activating tracing with AWS CloudFormation	522
Interpreting an X-Ray trace	523
Native AOT compilation	525
Limitations	525
Prerequisites	525
Lambda runtime	526
Set up your project	526
Edit your Lambda function code	526
Deploy your Lambda function	526
Add support for complex types	526
Troubleshooting	527
Building with PowerShell	528
Development Environment	529
Deployment package	530
Creating a Lambda function	530
Handler	532
Returning data	532
Context	533
Logging	534
Creating a function that returns logs	534
Using the Lambda console	535
Using the CloudWatch console	535
Using the AWS Command Line Interface (AWS CLI)	536
Deleting logs	538

Errors	539
Syntax	539
How it works	540
Using the Lambda console	540
Using the AWS Command Line Interface (AWS CLI)	541
Error handling in other AWS services	541
What's next?	542
Building with Rust	543
Handler	545
Using shared state	545
Context	547
Accessing invoke context information	547
HTTP events	548
Deploy .zip file archives	550
Prerequisites	550
Building the function	550
Deploying the function	551
Invoking the function	552
Logging	553
Creating a function that writes logs	553
Advanced logging with the Tracing crate	553
Errors	555
Creating a function that returns errors	555
Integrating other services	556
Listing of services and links to more information	556
Event-driven invocation	558
Lambda polling	558
Use cases	559
Example 1: Amazon S3 pushes events and invokes a Lambda function	559
Example 2: AWS Lambda pulls events from a Kinesis stream and invokes a Lambda function	560
Alexa	561
API Gateway	562
Adding an endpoint to your Lambda function	562
Proxy integration	562
Event format	563
Response format	564
Permissions	564
Handling errors with an API Gateway API	566
Choosing an API type	567
Sample applications	568
Tutorial	568
Sample template	582
CloudTrail	583
CloudTrail logs	585
Sample code	588
EventBridge (CloudWatch Events)	591
Tutorial	592
Schedule expressions	595
CloudWatch Logs	597
CloudFormation	598
CloudFront (Lambda@Edge)	601
CodeCommit	603
CodePipeline	604
Permissions	605
CodeWhisperer	606
Cognito	607
Config	608

Connect	609
DocumentDB	610
Example Amazon DocumentDB event	610
Prerequisites and permissions	611
Network configuration	612
Configuring a DocumentDB change stream as an event source	612
Event source mapping API	613
Monitoring your DocumentDB event source	615
Tutorial	615
DynamoDB	635
Example event	635
Polling and batching streams	636
Simultaneous readers	637
Execution role permissions	637
Configuring a stream as an event source	637
Event source mapping APIs	638
Error handling	640
Amazon CloudWatch metrics	641
Time windows	641
Reporting batch item failures	645
Amazon DynamoDB Streams configuration parameters	647
Tutorial	648
Sample code	653
Sample template	656
EC2	658
Permissions	658
ElastiCache	660
Prerequisites	660
Create the execution role	660
Create an ElastiCache cluster	661
Create a deployment package	661
Create the Lambda function	662
Test the Lambda function	662
Clean up your resources	104
Elastic Load Balancing	664
EFS	666
Connections	666
Throughput	667
IOPS	667
IoT	668
IoT Events	669
Apache Kafka	671
Example event	671
Kafka cluster authentication	672
Managing API access and permissions	674
Authentication and authorization errors	676
Network configuration	677
Adding a Kafka cluster as an event source	677
Using a Kafka cluster as an event source	680
Auto scaling of the Kafka event source	680
Event source API operations	681
Event source errors	681
Amazon CloudWatch metrics	681
Self-managed Apache Kafka configuration parameters	682
Kinesis Firehose	683
Kinesis Streams	684
Example event	684

Polling and batching streams	685
Configuring your data stream and function	686
Execution role permissions	686
Configuring a stream as an event source	687
Filtering Kinesis events	688
Event source mapping API	688
Error handling	690
Amazon CloudWatch metrics	691
Time windows	691
Reporting batch item failures	693
Amazon Kinesis configuration parameters	695
Tutorial	696
Sample code	701
Sample template	704
Lex	706
Roles and permissions	706
MQ	708
Lambda consumer group	709
Execution role permissions	711
Configuring a broker as an event source	712
Event source mapping API	712
Event source mapping errors	714
Amazon MQ and RabbitMQ configuration parameters	715
MSK	716
Example event	716
MSK cluster authentication	717
Managing API access and permissions	720
Authentication and authorization errors	722
Network configuration	723
Adding Amazon MSK as an event source	724
Auto scaling of the Amazon MSK event source	725
Amazon CloudWatch metrics	726
Amazon MSK configuration parameters	726
RDS	728
RDS Tutorial	728
Configuring the function	740
S3	741
Tutorial: Use an S3 trigger	742
Tutorial: Use an Amazon S3 trigger to create thumbnails	749
S3 Batch	764
Invoking Lambda functions from Amazon S3 batch operations	765
S3 Object Lambda	766
Secrets Manager	767
SES	768
SNS	770
Tutorial	771
Sample code	775
SQS	778
Example standard queue message event	778
Example FIFO queue message event	779
Configuring a queue to use with Lambda	780
Execution role permissions	780
Configuring a queue as an event source	780
Scaling and processing	782
Maximum concurrency	782
Event source mapping APIs	783
Backoff strategy for failed invocations	784

Implementing partial batch responses	784
Amazon SQS configuration parameters	786
Tutorial	787
SQS cross-account tutorial	791
Sample code	795
Sample template	798
VPC Lattice	799
VPC Lattice concepts	799
Prerequisites and permissions	800
Limitations	801
Registering your Lambda function with a VPC Lattice network	801
Updating the target of a service in a VPC Lattice network	803
Deregistering a Lambda function target	804
Cross-account networking	804
Receiving events from VPC Lattice	805
Sending responses back to VPC Lattice	805
Monitoring a service in a VPC Lattice network	806
X-Ray	807
Execution role permissions	808
The AWS X-Ray daemon	809
Enabling active tracing with the Lambda API	809
Enabling active tracing with AWS CloudFormation	809
Best practices	811
Function code	811
Function configuration	812
Metrics and alarms	813
Working with streams	813
Security best practices	814
Access permissions	815
Execution role	816
Creating an execution role in the IAM console	816
Grant least privilege access to your Lambda execution role	817
Managing roles with the IAM API	817
Session duration for temporary security credentials	818
AWS managed policies for Lambda features	818
Working with Lambda execution environment credentials	820
User policies	823
Function development	823
Layer development and use	826
Cross-account roles	827
Condition keys for VPC settings	827
Control access using tags	828
Prerequisites	828
Step 1: Require tags	828
Step 2: Control actions using tags	829
Step 3: Grant list permissions	829
Step 4: Grant IAM permissions	830
Step 5: Create the IAM role	830
Step 6: Create the IAM user	830
Step 7: Test the permissions	831
Resource-based policies	832
Granting function access to AWS services	833
Granting function access to an organization	834
Granting function access to other accounts	834
Granting layer access to other accounts	836
Cleaning up resource-based policies	836
Resources and conditions	838

Policy conditions	839
Function resource names	839
Function actions	841
Event source mapping actions	843
Layer actions	843
Permissions boundaries	845
Security compliance	847
Data protection	847
Encryption in transit	848
Encryption at rest	848
Identity and access management	849
Audience	849
Authenticating with identities	849
Managing access using policies	851
How AWS Lambda works with IAM	853
Identity-based policy examples	853
Troubleshooting	855
Compliance validation	857
Resilience	857
Infrastructure security	858
Configuration and vulnerability analysis	858
Detect vulnerabilities in your Lambda functions With Amazon Inspector	859
Monitoring functions	860
Monitoring console	861
Pricing	861
Using the Lambda console	861
Types of monitoring graphs	861
Viewing graphs on the Lambda console	861
Viewing queries on the CloudWatch Logs console	862
What's next?	863
Function insights	864
How it works	864
Pricing	864
Supported runtimes	864
Enabling Lambda Insights in the console	864
Enabling Lambda Insights programmatically	865
Using the Lambda Insights dashboard	865
Detecting function anomalies	867
Troubleshooting a function	868
What's next?	863
Function metrics	870
Viewing metrics on the CloudWatch console	870
Types of metrics	870
Function logs	873
Prerequisites	873
Pricing	873
Using the Lambda console	873
Using the AWS Command Line Interface	874
Runtime function logging	876
What's next?	876
Code profiler	876
Supported runtimes	876
Activating CodeGuru Profiler from the Lambda console	877
What happens when you activate CodeGuru Profiler from the Lambda console?	877
What's next?	877
Example workflows	878
Prerequisites	878

Pricing	879
Viewing a service map	879
Viewing trace details	879
Using Trusted Advisor to view recommendations	880
What's next?	880
Creating container images	881
Base images for Lambda	882
AWS base images for Lambda	882
Base images for custom runtimes	882
Runtime interface clients	883
Runtime interface emulator	883
Testing images	884
Guidelines for using the RIE	884
Environment variables	884
Test an image with RIE included in the image	885
Build RIE into your base image	885
Test an image without adding RIE to the image	886
Prerequisites	887
Image types	887
Container tools	887
Container image settings	888
Creating images from AWS base images	888
Creating images from alternative base images	890
Upload the image to the Amazon ECR repository	891
Create an image using the AWS SAM toolkit	893
Lambda extensions	894
Execution environment	894
Impact on performance and resources	895
Permissions	895
Configuring extensions	896
Configuring extensions (.zip file archive)	896
Using extensions in container images	896
Next steps	896
Extensions partners	898
AWS managed extensions	898
Extensions API	900
Lambda execution environment lifecycle	900
Extensions API reference	906
Telemetry API	911
Creating extensions using the Telemetry API	912
Registering your extension	913
Creating a telemetry listener	913
Specifying a destination protocol	914
Configuring memory usage and buffering	915
Sending a subscription request to the Telemetry API	916
Inbound Telemetry API messages	916
API reference	919
Event schema reference	922
Converting events to OTel Spans	935
Logs API	939
Troubleshooting	948
Deployment	948
General: Permission is denied / Cannot load such file	948
General: Error occurs when calling the UpdateFunctionCode	948
Amazon S3: Error Code PermanentRedirect	949
General: Cannot find, cannot load, unable to import, class not found, no such file or directory ...	949
General: Undefined method handler	949

Lambda: Layer conversion failed	950
Lambda: InvalidParameterValueException or RequestEntityTooLargeException	950
Lambda: InvalidParameterValueException	951
Lambda: Concurrency and memory quotas	951
Invocation	951
IAM: lambda:InvokeFunction not authorized	951
Lambda: Operation cannot be performed ResourceConflictException	952
Lambda: Function is stuck in Pending	952
Lambda: One function is using all concurrency	952
General: Cannot invoke function with other accounts or services	952
General: Function invocation is looping	952
Lambda: Alias routing with provisioned concurrency	953
Lambda: Cold starts with provisioned concurrency	953
Lambda: Latency variability with provisioned concurrency	953
Lambda: Variability between initialization and invocation times	954
Lambda: Cold starts with new versions	954
EFS: Function could not mount the EFS file system	954
EFS: Function could not connect to the EFS file system	954
EFS: Function could not mount the EFS file system due to timeout	954
Lambda: Lambda detected an IO process that was taking too long	955
Execution	955
Lambda: Execution takes too long	955
Lambda: Logs or traces don't appear	955
Lambda: The function returns before execution finishes	956
AWS SDK: Versions and updates	956
Python: Libraries load incorrectly	957
Networking	957
VPC: Function loses internet access or times out	957
VPC: Function needs access to AWS services without using the internet	957
VPC: Elastic network interface limit reached	957
Container images	958
Container: CodeArtifactUserException errors related to the code artifact.	958
Container: ManifestKeyCustomerException errors related to the code manifest key.	958
Container: Error occurs on runtime InvalidEntrypoint	958
Lambda: System provisioning additional capacity	958
CloudFormation: ENTRYPPOINT is being overridden with a null or empty value	959
Lambda applications	960
Manage applications	961
Monitoring applications	961
Custom monitoring dashboards	961
Tutorial – Create an application	964
Prerequisites	965
Create an application	965
Invoke the function	966
Add an AWS resource	967
Update the permissions boundary	969
Update the function code	969
Next steps	970
Troubleshooting	971
Clean up	972
Rolling deployments	973
Example AWS SAM Lambda template	973
Mobile SDK for Android	975
Tutorial	975
Sample code	981
Orchestrating functions	983
Application patterns	983

State machine components	983
State machine application patterns	983
Applying patterns to state machines	984
Example branching application pattern	984
Manage state machines	986
Viewing state machine details	987
Editing a state machine	987
Running a state machine	987
Orchestration examples	988
Configuring a Lambda function as a task	988
Configuring a state machine as an event source	988
Handling function and service errors	989
AWS CloudFormation and AWS SAM	990
Lambda SnapStart	992
Supported features and limitations	992
Supported Regions	992
Compatibility considerations	993
Pricing	993
SnapStart and provisioned concurrency	994
Additional resources	994
Activating SnapStart	995
Activating SnapStart (console)	995
Activating SnapStart (AWS CLI)	995
Activating SnapStart (API)	997
Function states	997
Updating a snapshot	998
Using SnapStart with the AWS SDK for Java	998
Using SnapStart with AWS CloudFormation, AWS SAM, and AWS CDK	998
Deleting snapshots	998
Handling uniqueness	999
Avoid saving state that depends on uniqueness during initialization	999
Use CSPRNGs	999
Scanning tool	1000
Runtime hooks	1001
Step 1: Update the build configuration	1001
Step 2: Update the Lambda handler	1001
Monitoring	1003
CloudWatch logs	1003
Telemetry API	1003
API Gateway and function URL metrics	1004
Security model	1005
Best practices	1006
Network connections	1006
Performance tuning	1006
Sample applications	1008
Blank function	1010
Architecture and handler code	1010
Deployment automation with AWS CloudFormation and the AWS CLI	1011
Instrumentation with the AWS X-Ray	1013
Dependency management with layers	1013
Error processor	1015
Architecture and event structure	1015
Instrumentation with AWS X-Ray	1016
AWS CloudFormation template and additional resources	1016
List manager	1018
Architecture and event structure	1018
Instrumentation with AWS X-Ray	1020

AWS CloudFormation templates and additional resources	1020
Working with AWS SDKs	1021
Code examples	1022
Actions	1024
Create a function	1024
Delete a function	1033
Get a function	1037
Invoke a function	1042
List functions	1048
Update function code	1052
Update function configuration	1058
Scenarios	1062
Get started with functions	1063
Cross-service examples	1122
Create a REST API to track COVID-19 data	1123
Create a lending library REST API	1123
Create a messenger application	1124
Create a serverless application to manage photos	1125
Create a websocket chat application	1125
Invoke a Lambda function from a browser	1126
Use API Gateway to invoke a Lambda function	1126
Use Step Functions to invoke Lambda functions	1128
Use scheduled events to invoke a Lambda function	1129
Lambda Quotas	1131
Compute and storage	1131
Function configuration, deployment, and execution	1131
Lambda API requests	1133
Other services	1133
AWS glossary	1134
API reference	1135
Actions	1135
AddLayerVersionPermission	1137
AddPermission	1141
CreateAlias	1146
CreateCodeSigningConfig	1150
CreateEventSourceMapping	1153
CreateFunction	1165
CreateFunctionUrlConfig	1178
DeleteAlias	1182
DeleteCodeSigningConfig	1184
DeleteEventSourceMapping	1186
DeleteFunction	1193
DeleteFunctionCodeSigningConfig	1195
DeleteFunctionConcurrency	1197
DeleteFunctionEventInvokeConfig	1199
DeleteFunctionUrlConfig	1201
DeleteLayerVersion	1203
DeleteProvisionedConcurrencyConfig	1205
GetAccountSettings	1207
GetAlias	1209
GetCodeSigningConfig	1212
GetEventSourceMapping	1214
GetFunction	1220
GetFunctionCodeSigningConfig	1224
GetFunctionConcurrency	1227
GetFunctionConfiguration	1229
GetFunctionEventInvokeConfig	1237

GetFunctionUrlConfig	1240
GetLayerVersion	1243
GetLayerVersionByArn	1247
GetLayerVersionPolicy	1250
GetPolicy	1252
GetProvisionedConcurrencyConfig	1254
GetRuntimeManagementConfig	1257
Invoke	1260
InvokeAsync	1266
InvokeWithResponseStream	1268
ListAliases	1274
ListCodeSigningConfigs	1277
ListEventSourceMappings	1279
ListFunctionEventInvokeConfigs	1283
ListFunctions	1286
ListFunctionsByCodeSigningConfig	1290
ListFunctionUrlConfigs	1292
ListLayers	1295
ListLayerVersions	1298
ListProvisionedConcurrencyConfigs	1301
ListTags	1304
ListVersionsByFunction	1306
PublishLayerVersion	1310
PublishVersion	1315
PutFunctionCodeSigningConfig	1324
PutFunctionConcurrency	1327
PutFunctionEventInvokeConfig	1330
PutProvisionedConcurrencyConfig	1334
PutRuntimeManagementConfig	1337
RemoveLayerVersionPermission	1341
RemovePermission	1343
TagResource	1345
UntagResource	1347
UpdateAlias	1349
UpdateCodeSigningConfig	1353
UpdateEventSourceMapping	1356
UpdateFunctionCode	1367
UpdateFunctionConfiguration	1377
UpdateFunctionEventInvokeConfig	1389
UpdateFunctionUrlConfig	1393
Data Types	1396
AccountLimit	1399
AccountUsage	1400
AliasConfiguration	1401
AliasRoutingConfiguration	1403
AllowedPublishers	1404
AmazonManagedKafkaEventSourceConfig	1405
CodeSigningConfig	1406
CodeSigningPolicies	1408
Concurrency	1409
Cors	1410
DeadLetterConfig	1412
DestinationConfig	1413
DocumentDBEventSourceConfig	1414
Environment	1415
EnvironmentError	1416
EnvironmentResponse	1417

EphemeralStorage	1418
EventSourceMappingConfiguration	1419
FileSystemConfig	1424
Filter	1425
FilterCriteria	1426
FunctionCode	1427
FunctionCodeLocation	1429
FunctionConfiguration	1430
FunctionEventInvokeConfig	1436
FunctionUrlConfig	1438
ImageConfig	1440
ImageConfigError	1441
ImageConfigResponse	1442
InvokeResponseStreamUpdate	1443
InvokeWithResponseStreamCompleteEvent	1444
InvokeWithResponseStreamResponseEvent	1445
Layer	1446
LayersListItem	1447
LayerVersionContentInput	1448
LayerVersionContentOutput	1449
LayerVersionsListItem	1450
OnFailure	1452
OnSuccess	1453
ProvisionedConcurrencyConfigListItem	1454
RuntimeVersionConfig	1456
RuntimeVersionError	1457
ScalingConfig	1458
SelfManagedEventSource	1459
SelfManagedKafkaEventSourceConfig	1460
SnapStart	1461
SnapStartResponse	1462
SourceAccessConfiguration	1463
TracingConfig	1465
TracingConfigResponse	1466
VpcConfig	1467
VpcConfigResponse	1468
Certificate errors when using an SDK	1468
Common Errors	1469
Common Parameters	1471
Document history	1473
Earlier updates	1485

What is AWS Lambda?

AWS Lambda is a **compute service** that lets you **run code without provisioning or managing servers**.

Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, and logging. With Lambda, all you need to do is supply your code in one of the language runtimes that Lambda supports.

You organize your code into Lambda functions. The Lambda service runs your function only when needed and **scales automatically**. You only **pay for the compute time that you consume**—there is no charge when your code is not running. For more information, see [AWS Lambda Pricing](#).

Tip

To learn how to build **serverless solutions**, check out the [Serverless Developer Guide](#).

When to use Lambda

Lambda is an ideal compute service for application scenarios that **need to scale up rapidly**, and **scale down to zero when not in demand**. For example, you can use Lambda for:

- **File processing:** Use Amazon Simple Storage Service (Amazon S3) to trigger Lambda data processing in real time after an upload.
- **Stream processing:** Use Lambda and Amazon Kinesis to process real-time streaming data for application activity tracking, transaction order processing, clickstream analysis, data cleansing, log filtering, indexing, social media analysis, Internet of Things (IoT) device telemetry, and metering.
- **Web applications:** Combine Lambda with other AWS services to build powerful web applications that automatically scale up and down and run in a highly available configuration across multiple data centers.
- **IoT backends:** Build serverless backends using Lambda to handle web, mobile, IoT, and third-party API requests.
- **Mobile backends:** Build backends using Lambda and Amazon API Gateway to authenticate and process API requests. Use AWS Amplify to easily integrate with your iOS, Android, Web, and React Native frontends.

When using Lambda, **you are responsible only for your code**. Lambda manages the compute fleet that offers a balance of memory, CPU, network, and other resources to run your code. Because Lambda manages these resources, you cannot log in to compute instances or customize the operating system on provided runtimes.

Lambda performs operational and administrative activities on your behalf, including managing capacity, monitoring, and logging your Lambda functions.

If you do need to manage your compute resources, AWS has other compute services to consider, such as:

- **AWS App Runner** builds and deploys containerized web applications automatically, load balances traffic with encryption, scales to meet your traffic needs, and allows for the configuration of how services are accessed and communicate with other AWS applications in a private Amazon VPC.
- **AWS Fargate** with Amazon ECS runs containers without having to provision, configure, or scale clusters of virtual machines.
- **Amazon EC2** lets you customize operating system, network and security settings, and the entire software stack. You are responsible for provisioning capacity, monitoring fleet health and performance, and using Availability Zones for fault tolerance.

Key features

The following key features help you develop Lambda applications that are scalable, secure, and easily extensible:

[Configuring function options \(p. 71\)](#)

Configure your Lambda function using the console or AWS CLI.

[Environment variables \(p. 76\)](#)

Use **environment variables** to adjust your function's behavior without updating code.

[Versions \(p. 83\)](#)

Manage the deployment of your functions with **versions**, so that, for example, a new function can be used for beta testing without affecting users of the stable production version.

[Container images \(p. 881\)](#)

Create a container image for a Lambda function by using an AWS provided base image or an alternative base image so that you can reuse your existing container tooling or deploy larger workloads that rely on sizable dependencies, such as machine learning.

[Layers \(p. 93\)](#)

Package libraries and other dependencies to reduce the size of deployment archives and makes it faster to deploy your code.

[Lambda extensions \(p. 900\)](#)

Augment your Lambda functions with tools for monitoring, observability, security, and governance.

[Function URLs \(p. 166\)](#)

Add a dedicated HTTP(S) endpoint to your Lambda function.

[Response streaming \(p. 99\)](#)

Configure your Lambda function URLs to stream response payloads back to clients from Node.js functions, to improve time to first byte (TTFB) performance or to return larger payloads.

[Concurrency and scaling controls \(p. 197\)](#)

Apply fine-grained control over the scaling and responsiveness of your production applications.

[Code signing \(p. 241\)](#)

Verify that only approved developers publish unaltered, trusted code in your Lambda functions

[Private networking \(p. 89\)](#)

Create a private network for resources such as databases, cache instances, or internal services.

[Database access and proxy \(p. 232\)](#)

Create an Amazon RDS Proxy database proxy to manage a pool of database connections and relay queries from a function. With a proxy, your function can achieve high [concurrency \(p. 11\)](#) levels without exhausting database connections.

[File system access \(p. 236\)](#)

Configure a function to mount an Amazon Elastic File System (Amazon EFS) to a local directory, so that your function code can access and modify shared resources safely and at high concurrency.

[Lambda SnapStart for Java \(p. 992\)](#)

Improve startup performance for **Java runtimes** by up to 10x at no extra cost, typically with no changes to your function code.

Getting started with Lambda

To get started with Lambda, use the Lambda console to create a function. In a few minutes, you can create and deploy a function, invoke it, and then view logs and metrics.

Tip

To learn how to build **serverless solutions**, check out the [Serverless Developer Guide](#).

Prerequisites

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, [assign administrative access to an administrative user](#), and use only the root user to perform [tasks that require root user access](#).

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to <https://aws.amazon.com/> and choosing **My Account**.

Create an administrative user

After you sign up for an AWS account, create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the [AWS Management Console](#) as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.
For help signing in by using root user, see [Signing in as the root user](#) in the *AWS Sign-In User Guide*.
2. Turn on multi-factor authentication (MFA) for your root user.

For instructions, see [Enable a virtual MFA device for your AWS account root user \(console\)](#) in the *IAM User Guide*.

Create an administrative user

- For your daily administrative tasks, grant administrative access to an administrative user in AWS IAM Identity Center (successor to AWS Single Sign-On).

For instructions, see [Getting started](#) in the *AWS IAM Identity Center (successor to AWS Single Sign-On) User Guide*.

Sign in as the administrative user

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

For help signing in using an IAM Identity Center user, see [Signing in to the AWS access portal](#) in the [AWS Sign-In User Guide](#).

Create a Lambda function with the console

In this getting started exercise, create a Lambda function using a blueprint. A blueprint provides sample code to do some minimal processing. Most blueprints process events from specific event sources, such as Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, or a custom application.

To create a Lambda function with the console

- Open the [Functions page](#) of the Lambda console.
- Choose **Create function**.
- Select **Use a blueprint**.
- Open the **Select blueprint** dropdown list and search for **Hello world function**. Select node.js or python.
- Enter a **Function name**.
- For **Execution role**, choose **Create a new role with basic Lambda permissions**. Lambda creates an [execution role \(p. 816\)](#) that grants the function permission to upload logs to Amazon CloudWatch. The Lambda function assumes the execution role when you invoke your function, and uses the execution role to create credentials for the AWS SDK and to read data from event sources.

Invoke the function

To invoke the function from the console, create a test event.

- Choose the **Test** tab.
- For **Test event action**, choose **Create new event**.
- For **Event name**, enter a name for the test event.
- For **Event sharing settings**, choose **Private**.
- For **Template**, leave the default **hello-world** option.
- In the **Event JSON**, replace **value1** with **hello, world!**. Don't change **key1** or the event structure. Example:

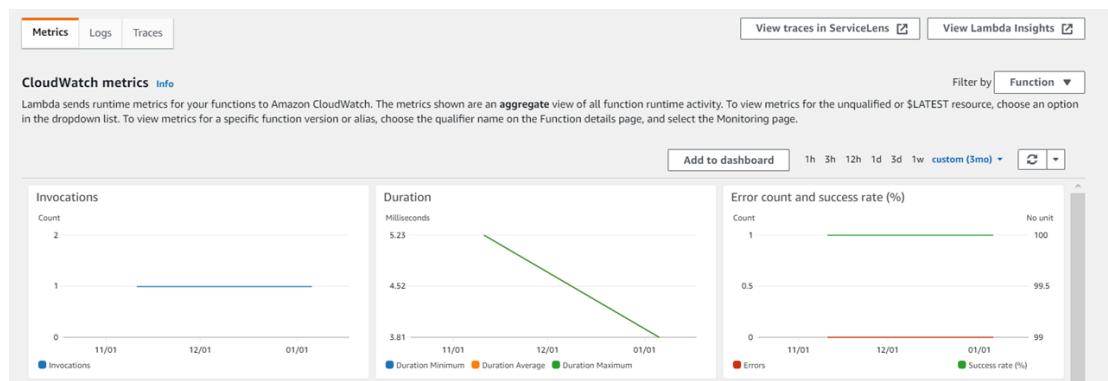
```
{  
    "key1": "hello, world!",  
    "key2": "value2",  
    "key3": "value3"  
}
```

- Choose **Save**, and then choose **Test**. Lambda invokes the function on your behalf. The [function handler \(p. 320\)](#) receives and then processes the sample event.
- Review the **Execution result**. Under **Details**, you should see the value that you entered in step 6: **"hello, world!"**. The execution result also includes the following information:

- The **Summary** section shows the key information from the REPORT line in the invocation log.
 - The **Log output** section shows the complete invocation log. Lambda writes all invocation logs to Amazon CloudWatch.
- Choose **Test** to invoke the function a few more times and gather additional metrics that you can view in the next step.
 - Choose the **Monitor** tab. This page shows graphs of the metrics that Lambda sends to CloudWatch.

Note

It might take 5 to 10 minutes for logs to show up after a function invocation.



For more information on these graphs, see [Monitoring functions on the Lambda console \(p. 861\)](#).

Clean up

If you are done working with the example function, delete it. You can also delete the log group that stores the function's logs, and the execution role that the console created.

To delete a Lambda function

- Open the [Functions page](#) of the Lambda console.
- Choose a function.
- Choose **Actions, Delete**.
- In the **Delete function** dialog box, enter *delete*, and then choose **Delete**.

To delete the log group

- Open the [Log groups page](#) of the CloudWatch console.
- Select the function's log group (/aws/lambda/my-function).
- Choose **Actions, Delete log group(s)**.
- In the **Delete log group(s)** dialog box, choose **Delete**.

To delete the execution role

- Open the [Roles page](#) of the AWS Identity and Access Management (IAM) console.
- Select the function's execution role (for example, my-function-role-*31exmpl*).
- Choose **Delete**.
- In the **Delete role** dialog box, enter the role name and then choose **Delete**.

You can automate the creation and cleanup of functions, log groups, and roles with AWS CloudFormation and the AWS Command Line Interface (AWS CLI). For fully functional sample applications, see [Lambda sample applications \(p. 1008\)](#).

Additional resources

After creating your first Lambda function, try a tutorial:

- [Tutorial: Using an Amazon S3 trigger to invoke a Lambda function \(p. 742\)](#): Use the Lambda console to create a trigger that invokes your function every time that you add an object to an Amazon S3 bucket.
- [Tutorial: Using Lambda with API Gateway \(p. 568\)](#): Create an Amazon API Gateway REST API that invokes a Lambda function.
- [Tutorial: Using AWS Lambda with scheduled events \(p. 592\)](#): Configure a Lambda function to run every minute. Configure Amazon Simple Notification Service (Amazon SNS) to email you if the function returns an error.

To learn more about serverless application development, see the following:

- [Serverless Developer Guide](#) for directed learning paths to build *serverless solutions*. Also see the related [Serverless Patterns Workshop](#) for a hands-on experience.
- [AWS Compute Blog](#)
- [AWS Serverless Land](#)
- The [AWS Online Tech Talks](#) YouTube channel includes videos about Lambda-related topics. For an overview of serverless applications and Lambda, see the [Introduction to AWS Lambda & Serverless Applications](#) video.

Accessing Lambda

You can create, invoke, and manage Lambda functions using any of the following interfaces:

- **AWS Management Console** – Provides a web interface for you to access your functions. For more information, see [Lambda console \(p. 21\)](#).
- **AWS Command Line Interface (AWS CLI)** – Provides commands for a broad set of AWS services, including Lambda, and is supported on Windows, macOS, and Linux. For more information, see [Using Lambda with the AWS CLI \(p. 191\)](#).
- **AWS SDKs** – Provide language-specific APIs and manage many of the connection details, such as signature calculation, request retry handling, and error handling. For more information, see [AWS SDKs](#).
- **AWS CloudFormation** – Enables you to create templates that define your Lambda applications. For more information, see [AWS Lambda applications \(p. 960\)](#). AWS CloudFormation also supports the [AWS Cloud Development Kit \(AWS CDK\)](#).
- **AWS Serverless Application Model (AWS SAM)** – Provides templates and a CLI to configure and manage AWS serverless applications. For more information, see [Getting started with AWS SAM](#).

Authoring and deploying functions

The following table lists the [languages that Lambda supports \(p. 37\)](#) and the tools and options that you can use with them. The available tools and options depend on:

- The language that you use to write your Lambda function code.
- The libraries that you use in your code. The Lambda runtimes provide some of the libraries, and you must upload any additional libraries that you use.

Language	Tools and options for authoring code
Node.js (p. 254)	<ul style="list-style-type: none"> • Lambda console • Visual Studio, with IDE plugin (see AWS Lambda Support in Visual Studio on the AWS Developer Blog) • Your own authoring environment
Java (p. 386)	<ul style="list-style-type: none"> • Eclipse, with the AWS Toolkit for Eclipse • IntelliJ, with the AWS Toolkit for JetBrains • Your own authoring environment
C# (p. 487)	<ul style="list-style-type: none"> • Visual Studio, with IDE plugin (see AWS Toolkit for Visual Studio) • .NET Core (see Download .NET on the Microsoft website) • Your own authoring environment
Python (p. 318)	<ul style="list-style-type: none"> • Lambda console • PyCharm, with the AWS Toolkit for JetBrains • Your own authoring environment
Ruby (p. 361)	<ul style="list-style-type: none"> • Lambda console • Your own authoring environment
Go (p. 453)	<ul style="list-style-type: none"> • Your own authoring environment
PowerShell (p. 528)	<ul style="list-style-type: none"> • Your own authoring environment • PowerShell Core 6.0 (see Installing various versions of PowerShell on the Microsoft Docs website) • .NET Core 3.1 SDK (see Download .NET on the Microsoft website) • AWSLambdaPSCore module (see AWSLambdaPSCore on the PowerShell Gallery website)

You deploy your function code to Lambda using a [deployment package](#). Lambda supports two types of deployment packages:

- A [.zip file archive \(p. 107\)](#) that contains your function code and its dependencies.
- A [container image \(p. 881\)](#) that is compatible with the [Open Container Initiative \(OCI\)](#) specification.

AWS Lambda foundations

The Lambda function is the principle resource of the Lambda service.

You can configure your functions using the Lambda console, Lambda API, AWS CloudFormation or AWS SAM. You create code for the function and upload the code using a deployment package. Lambda invokes the function when an event occurs. Lambda runs multiple instances of your function in parallel, governed by concurrency and scaling limits.

Topics

- [Lambda concepts \(p. 9\)](#)
- [Lambda programming model \(p. 13\)](#)
- [Lambda execution environment \(p. 14\)](#)
- [Lambda deployment packages \(p. 18\)](#)
- [Lambda console \(p. 21\)](#)
- [Lambda instruction set architectures \(ARM/x86\) \(p. 29\)](#)
- [Additional Lambda features \(p. 32\)](#)
- [Learn how to build serverless solutions \(p. 36\)](#)

Lambda concepts

Lambda runs instances of your function to process events. You can invoke your function directly using the Lambda API, or you can configure an AWS service or resource to invoke your function.

Concepts

- [Function \(p. 9\)](#)
- [Trigger \(p. 9\)](#)
- [Event \(p. 9\)](#)
- [Execution environment \(p. 10\)](#)
- [Instruction set architecture \(p. 10\)](#)
- [Deployment package \(p. 10\)](#)
- [Runtime \(p. 10\)](#)
- [Layer \(p. 11\)](#)
- [Extension \(p. 11\)](#)
- [Concurrency \(p. 11\)](#)
- [Qualifier \(p. 11\)](#)
- [Destination \(p. 12\)](#)

Function

A *function* is a resource that you can invoke to run your code in Lambda. A function has code to process the [events \(p. 9\)](#) that you pass into the function or that other AWS services send to the function.

Trigger

A *trigger* is a resource or configuration that invokes a Lambda function. Triggers include AWS services that you can configure to invoke a function and [event source mappings \(p. 131\)](#). An event source mapping is a resource in Lambda that reads items from a stream or queue and invokes a function. For more information, see [Invoking Lambda functions \(p. 118\)](#) and [Using AWS Lambda with other services \(p. 556\)](#).

Event

An *event* is a JSON-formatted document that contains data for a Lambda function to process. The runtime converts the event to an object and passes it to your function code. When you invoke a function, you determine the structure and contents of the event.

Example custom event – weather data

```
{  
    "TemperatureK": 281,  
    "WindKmh": -3,  
    "HumidityPct": 0.55,  
    "PressureHPa": 1020  
}
```

When an AWS service invokes your function, the service defines the shape of the event.

Example service event – Amazon SNS notification

```
{  
  "Records": [  
    {  
      "Sns": {  
        "Timestamp": "2019-01-02T12:45:07.000Z",  
        "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkaAi6RibDsvpi+tE/1+82j...65r==",  
        "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",  
        "Message": "Hello from SNS!",  
        ...  
      }  
    }  
  ]  
}
```

For more information about events from AWS services, see [Using AWS Lambda with other services \(p. 556\)](#).

Execution environment

An *execution environment* provides a secure and isolated runtime environment for your Lambda function. An execution environment manages the processes and resources that are required to run the function. The execution environment provides lifecycle support for the function and for any [extensions \(p. 11\)](#) associated with your function.

For more information, see [Lambda execution environment \(p. 14\)](#).

Instruction set architecture

The *instruction set architecture* determines the type of computer processor that Lambda uses to run the function. Lambda provides a choice of instruction set architectures:

- arm64 – 64-bit ARM architecture, for the AWS Graviton2 processor.
- x86_64 – 64-bit x86 architecture, for x86-based processors.

For more information, see [Lambda instruction set architectures \(ARM/x86\) \(p. 29\)](#).

Deployment package

You deploy your Lambda function code using a *deployment package*. Lambda supports two types of deployment packages:

- A .zip file archive that contains your function code and its dependencies. Lambda provides the operating system and runtime for your function.
- A container image that is compatible with the [Open Container Initiative \(OCI\)](#) specification. You add your function code and dependencies to the image. You must also include the operating system and a Lambda runtime.

For more information, see [Lambda deployment packages \(p. 18\)](#).

Runtime

The *runtime* provides a language-specific environment that runs in an execution environment. The runtime relays invocation events, context information, and responses between Lambda and the function. You can use runtimes that Lambda provides, or build your own. If you package your code as a .zip file archive, you must configure your function to use a runtime that matches your programming language. For a container image, you include the runtime when you build the image.

For more information, see [Lambda runtimes \(p. 37\)](#).

Layer

A Lambda *layer* is a .zip file archive that can contain additional code or other content. A layer can contain libraries, a [custom runtime \(p. 59\)](#), data, or configuration files.

Layers provide a convenient way to package libraries and other dependencies that you can use with your Lambda functions. Using layers reduces the size of uploaded deployment archives and makes it faster to deploy your code. Layers also promote code sharing and separation of responsibilities so that you can iterate faster on writing business logic.

You can include up to five layers per function. Layers count towards the standard Lambda [deployment size quotas \(p. 1131\)](#). When you include a layer in a function, the contents are extracted to the /opt directory in the execution environment.

By default, the layers that you create are private to your AWS account. You can choose to share a layer with other accounts or to make the layer public. If your functions consume a layer that a different account published, your functions can continue to use the layer version after it has been deleted, or after your permission to access the layer is revoked. However, you cannot create a new function or update functions using a deleted layer version.

Functions deployed as a container image do not use layers. Instead, you package your preferred runtime, libraries, and other dependencies into the container image when you build the image.

For more information, see [Creating and sharing Lambda layers \(p. 93\)](#).

Extension

Lambda *extensions* enable you to augment your functions. For example, you can use extensions to integrate your functions with your preferred monitoring, observability, security, and governance tools. You can choose from a broad set of tools that [AWS Lambda Partners](#) provides, or you can [create your own Lambda extensions \(p. 900\)](#).

An internal extension runs in the runtime process and shares the same lifecycle as the runtime. An external extension runs as a separate process in the execution environment. The external extension is initialized before the function is invoked, runs in parallel with the function's runtime, and continues to run after the function invocation is complete.

For more information, see [Lambda extensions \(p. 894\)](#).

Concurrency

Concurrency is the number of requests that your function is serving at any given time. When your function is invoked, Lambda provisions an instance of it to process the event. When the function code finishes running, it can handle another request. If the function is invoked again while a request is still being processed, another instance is provisioned, increasing the function's concurrency.

Concurrency is subject to [quotas \(p. 1131\)](#) at the AWS Region level. You can configure individual functions to limit their concurrency, or to enable them to reach a specific level of concurrency. For more information, see [Configuring reserved concurrency \(p. 210\)](#).

Qualifier

When you invoke or view a function, you can include a *qualifier* to specify a version or alias. A *version* is an immutable snapshot of a function's code and configuration that has a numerical qualifier. For

example, my-function:1. An *alias* is a pointer to a version that you can update to map to a different version, or split traffic between two versions. For example, my-function:BLUE. You can use versions and aliases together to provide a stable interface for clients to invoke your function.

For more information, see [Lambda function versions \(p. 83\)](#).

Destination

A *destination* is an AWS resource where Lambda can send events from an asynchronous invocation. You can configure a destination for events that fail processing. Some services also support a destination for events that are successfully processed.

For more information, see [Configuring destinations for asynchronous invocation \(p. 125\)](#).

Lambda programming model

Lambda provides a programming model that is common to all of the runtimes. The programming model defines the interface between your code and the Lambda system. You tell Lambda the entry point to your function by defining a **handler** in the function configuration. The runtime passes in objects to the handler that contain the invocation *event* and the *context*, such as the function name and request ID.

When the handler finishes processing the first event, the runtime sends it another. The function's class stays in memory, so clients and variables that are declared outside of the handler method in *initialization code* can be reused. To save processing time on subsequent events, create reusable resources like AWS SDK clients during initialization. Once initialized, each instance of your function can process thousands of requests.

Your function also has access to local storage in the `/tmp` directory. The directory content remains when the execution environment is frozen, providing a transient cache that can be used for multiple invocations. For more information, see [Lambda execution environment](#).

When [AWS X-Ray tracing \(p. 807\)](#) is enabled, the runtime records separate subsegments for initialization and execution.

The runtime captures logging output from your function and sends it to Amazon CloudWatch Logs. In addition to logging your function's output, the runtime also logs entries when function invocation starts and ends. This includes a report log with the request ID, billed duration, initialization duration, and other details. If your function throws an error, the runtime returns that error to the invoker.

Note

Logging is subject to [CloudWatch Logs quotas](#). Log data can be lost due to throttling or, in some cases, when an instance of your function is stopped.

Lambda scales your function by running additional instances of it as demand increases, and by stopping instances as demand decreases. This model leads to variations in application architecture, such as:

- Unless noted otherwise, incoming requests might be processed out of order or concurrently.
- Do not rely on instances of your function being long lived, instead store your application's state elsewhere.
- Use local storage and class-level objects to increase performance, but keep to a minimum the size of your deployment package and the amount of data that you transfer onto the execution environment.

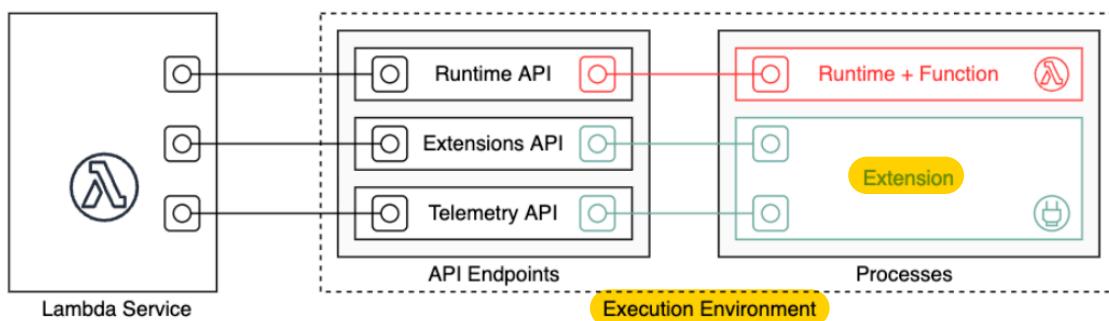
For a hands-on introduction to the programming model in your preferred programming language, see the following chapters.

- [Building Lambda functions with Node.js \(p. 254\)](#)
- [Building Lambda functions with Python \(p. 318\)](#)
- [Building Lambda functions with Ruby \(p. 361\)](#)
- [Building Lambda functions with Java \(p. 386\)](#)
- [Building Lambda functions with Go \(p. 453\)](#)
- [Building Lambda functions with C# \(p. 487\)](#)
- [Building Lambda functions with PowerShell \(p. 528\)](#)

Lambda execution environment

Lambda invokes your function in an execution environment, which provides a secure and isolated runtime environment. The execution environment manages the resources required to run your function. The execution environment also provides lifecycle support for the function's runtime and any [external extensions \(p. 894\)](#) associated with your function.

The function's runtime communicates with Lambda using the [Runtime API \(p. 54\)](#). Extensions communicate with Lambda using the [Extensions API \(p. 900\)](#). Extensions can also receive log messages and other telemetry from the function by using the [Telemetry API \(p. 911\)](#).



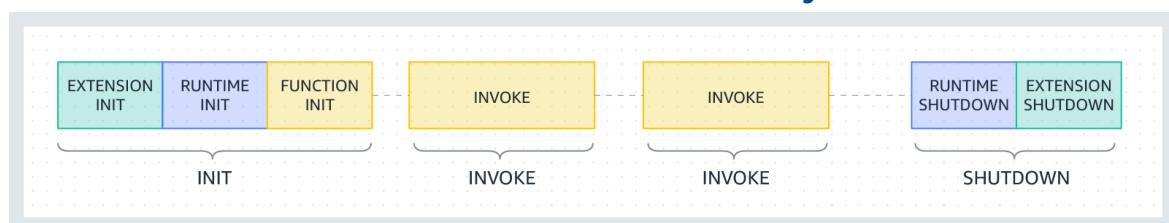
When you create your Lambda function, you specify configuration information, such as the amount of memory available and the maximum execution time allowed for your function. Lambda uses this information to set up the execution environment.

The function's runtime and each external extension are processes that run within the execution environment. Permissions, resources, credentials, and environment variables are shared between the function and the extensions.

Topics

- [Lambda execution environment lifecycle \(p. 14\)](#)

Lambda execution environment lifecycle



Each phase starts with an event that Lambda sends to the runtime and to all registered extensions. The runtime and each extension indicate completion by sending a Next API request. Lambda freezes the execution environment when the runtime and each extension have completed and there are no pending events.

Topics

- [Init phase \(p. 15\)](#)
- [Restore phase \(Lambda SnapStart only\) \(p. 15\)](#)
- [Invoke phase \(p. 15\)](#)
- [Failures during the invoke phase \(p. 16\)](#)

- [Shutdown phase \(p. 16\)](#)

Init phase

In the Init phase, Lambda performs three tasks:

- Start all extensions (Extension init)
- Bootstrap the runtime (Runtime init)
- Run the function's static code (Function init)
- Run any beforeCheckpoint [runtime hooks \(p. 1001\)](#) (Lambda SnapStart only)

The Init phase ends when the runtime and all extensions signal that they are ready by sending a Next API request. The Init phase is limited to 10 seconds. If all three tasks do not complete within 10 seconds, Lambda retries the Init phase at the time of the first function invocation with the configured function timeout.

When [Lambda SnapStart \(p. 992\)](#) is activated, the Init phase happens when you publish a function version. Lambda saves a snapshot of the memory and disk state of the initialized execution environment, persists the encrypted snapshot, and caches it for low-latency access. If you have a beforeCheckpoint [runtime hook \(p. 1001\)](#), then the code runs at the end of Init phase.

Note

The 10-second timeout doesn't apply to SnapStart functions. When Lambda creates a snapshot, your initialization code can run for up to 15 minutes. The time limit is 130 seconds or the [configured function timeout \(p. 73\)](#) (maximum 900 seconds), whichever is higher.

When you use [provisioned concurrency](#), Lambda begins the init phase shortly after you publish a function version. There can be a large gap between your function's initialization and invocation phases. For functions using unreserved (on-demand) concurrency, Lambda may proactively initialize a function instance, even if there's no invocation. When this happens, you can observe an unexpected time gap between your function's initialization and invocation phases. This gap can appear similar to what you would observe when using provisioned concurrency.

Restore phase (Lambda SnapStart only)

When you first invoke a [SnapStart \(p. 992\)](#) function and as the function scales up, Lambda resumes new execution environments from the persisted snapshot instead of initializing the function from scratch. If you have an [afterRestore\(\) runtime hook \(p. 1001\)](#), the code runs at the end of the Restore phase. You are charged for the duration of [afterRestore\(\)](#) runtime hooks. The runtime (JVM) must load and [afterRestore\(\)](#) runtime hooks must complete within the timeout limit (10 seconds). Otherwise, you'll get a [SnapStartTimeoutException](#). When the Restore phase completes, Lambda invokes the function handler (the Invoke phase).

Invoke phase

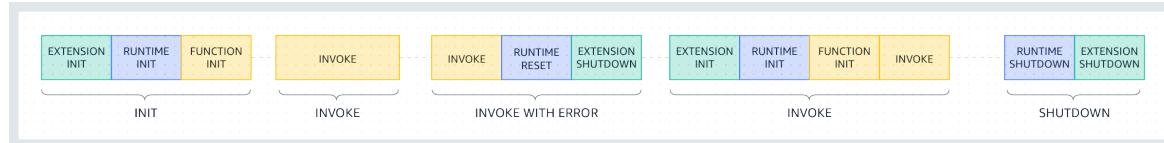
When a Lambda function is invoked in response to a Next API request, Lambda sends an Invoke event to the runtime and to each extension.

The function's timeout setting limits the duration of the entire Invoke phase. For example, if you set the function timeout as 360 seconds, the function and all extensions need to complete within 360 seconds. Note that there is no independent post-invoke phase. The duration is the sum of all invocation time (runtime + extensions) and is not calculated until the function and all extensions have finished executing.

The invoke phase ends after the runtime and all extensions signal that they are done by sending a Next API request.

Failures during the invoke phase

If the Lambda function crashes or times out during the Invoke phase, Lambda resets the execution environment. The following diagram illustrates Lambda execution environment behavior when there's an invoke failure:



In the previous diagram:

- The first phase is the **INIT** phase, which runs without errors.
- The second phase is the **INVOKE** phase, which runs without errors.
- At some point, suppose your function runs into an invoke failure (such as a function timeout or runtime error). The third phase, labeled **INVOKE WITH ERROR**, illustrates this scenario. When this happens, the Lambda service performs a reset. The reset behaves like a Shutdown event. First, Lambda shuts down the runtime, then sends a Shutdown event to each registered external extension. The event includes the reason for the shutdown. If this environment is used for a new invocation, Lambda re-initializes the extension and runtime together with the next invocation.

Note

The Lambda reset does not clear the /tmp directory content prior to the next init phase. This behavior is consistent with the regular shutdown phase.

- The fourth phase represents the **INVOKE** phase immediately following an invoke failure. Here, Lambda initializes the environment again by re-running the **INIT** phase. This is called a *suppressed init*. When suppressed inits occur, Lambda doesn't explicitly report an additional **INIT** phase in CloudWatch Logs. Instead, you may notice that the duration in the REPORT line includes an additional **INIT** duration + the **INVOKE** duration. For example, suppose you see the following logs in 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 this example, the difference between the REPORT and START timestamps is 2.5 seconds. This doesn't match the reported duration of 3022.91 milliseconds, because it doesn't take into account the extra **INIT** (suppressed init) that Lambda performed. In this example, you can infer that the actual **INVOKE** phase took 2.5 seconds.

For more insight into this behavior, you can use the [Lambda Telemetry API \(p. 911\)](#). The Telemetry API emits INIT_START, INIT_RUNTIME_DONE, and INIT_REPORT events with phase=invoke whenever suppressed inits occur in between invoke phases.

- The fifth phase represents the **SHUTDOWN** phase, which runs without errors.

Shutdown phase

When Lambda is about to shut down the runtime, it sends a Shutdown event to each registered external extension. Extensions can use this time for final cleanup tasks. The Shutdown event is a response to a Next API request.

Duration: The entire Shutdown phase is capped at 2 seconds. If the runtime or any extension does not respond, Lambda terminates it via a signal (SIGKILL).

After the function and all extensions have completed, Lambda maintains the execution environment for some time in anticipation of another function invocation. In effect, Lambda freezes the execution environment. When the function is invoked again, Lambda thaws the environment for reuse. Reusing the execution environment has the following implications:

- Objects declared outside of the function's handler method remain initialized, providing additional optimization when the function is invoked again. For example, if your Lambda function establishes a database connection, instead of reestablishing the connection, the original connection is used in subsequent invocations. We recommend adding logic in your code to check if a connection exists before creating a new one.
- Each execution environment provides between 512 MB and 10,240 MB, in 1-MB increments, of disk space in the /tmp directory. The directory content remains when the execution environment is frozen, providing a transient cache that can be used for multiple invocations. You can add extra code to check if the cache has the data that you stored. For more information on deployment size limits, see [Lambda quotas \(p. 1131\)](#).
- Background processes or callbacks that were initiated by your Lambda function and did not complete when the function ended resume if Lambda reuses the execution environment. Make sure that any background processes or callbacks in your code are complete before the code exits.

When you write your function code, do not assume that Lambda automatically reuses the execution environment for subsequent function invocations. Other factors may dictate a need for Lambda to create a new execution environment, which can lead to unexpected results, such as database connection failures.

Lambda deployment packages

Your AWS Lambda function's code consists of scripts or compiled programs and their dependencies. You use a *deployment package* to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives.

Topics

- [Container images \(p. 18\)](#)
- [.zip file archives \(p. 18\)](#)
- [Layers \(p. 19\)](#)
- [Using other AWS services to build a deployment package \(p. 19\)](#)

Container images

A container image includes the base operating system, the runtime, Lambda extensions, your application code and its dependencies. You can also add static data, such as machine learning models, into the image.

Lambda provides a set of open-source base images that you can use to build your container image. To create and test container images, you can use the AWS Serverless Application Model (AWS SAM) command line interface (CLI) or native container tools such as the Docker CLI.

You upload your container images to Amazon Elastic Container Registry (Amazon ECR), a managed AWS container image registry service. To deploy the image to your function, you specify the Amazon ECR image URL using the Lambda console, the Lambda API, command line tools, or the AWS SDKs.

For more information about Lambda container images, see [Creating Lambda container images \(p. 881\)](#).

Note

Container images aren't supported for Lambda functions in the Middle East (UAE) Region.

.zip file archives

A .zip file archive includes your application code and its dependencies. When you author functions using the Lambda console or a toolkit, Lambda automatically creates a .zip file archive of your code.

When you create functions with the Lambda API, command line tools, or the AWS SDKs, you must create a deployment package. You also must create a deployment package if your function uses a compiled language, or to add dependencies to your function. To deploy your function's code, you upload the deployment package from Amazon Simple Storage Service (Amazon S3) or your local machine.

You can upload a .zip file as your deployment package using the Lambda console, AWS Command Line Interface (AWS CLI), or to an Amazon Simple Storage Service (Amazon S3) bucket.

Using the Lambda console

The following steps demonstrate how to upload a .zip file as your deployment package using the Lambda console.

To upload a .zip file on the Lambda console

1. Open the [Functions page](#) on the Lambda console.
2. Select a function.
3. In the **Code Source** pane, choose **Upload from** and then **.zip file**.

4. Choose **Upload** to select your local .zip file.
5. Choose **Save**.

Using the AWS CLI

You can upload a .zip file as your deployment package using the AWS Command Line Interface (AWS CLI). For language-specific instructions, see the following topics.

Node.js

[Deploy Node.js Lambda functions with .zip file archives \(p. 263\)](#)

Python

[Deploy Python Lambda functions with .zip file archives \(p. 324\)](#)

Ruby

[Deploy Ruby Lambda functions with .zip file archives \(p. 365\)](#)

Java

[Deploy Java Lambda functions with .zip or JAR file archives \(p. 393\)](#)

Go

[Deploy Go Lambda functions with .zip file archives \(p. 460\)](#)

C#

[Deploy C# Lambda functions with .zip file archives \(p. 497\)](#)

PowerShell

[Deploy PowerShell Lambda functions with .zip file archives \(p. 530\)](#)

Using Amazon S3

You can upload a .zip file as your deployment package using Amazon Simple Storage Service (Amazon S3). For more information, see [the section called “Using other AWS services”](#).

Layers

If you deploy your function code using a .zip file archive, you can use Lambda layers as a distribution mechanism for libraries, custom runtimes, and other function dependencies. Layers enable you to manage your in-development function code independently from the unchanging code and resources that it uses. You can configure your function to use layers that you create, layers that AWS provides, or layers from other AWS customers.

You do not use layers with container images. Instead, you package your preferred runtime, libraries, and other dependencies into the container image when you build the image.

For more information about layers, see [Creating and sharing Lambda layers \(p. 93\)](#).

Using other AWS services to build a deployment package

The following section describes other AWS services you can use to package dependencies for your Lambda function.

Deployment packages with C or C++ libraries

If your deployment package contains native libraries, you can build the deployment package with AWS Serverless Application Model (AWS SAM). You can use the AWS SAM CLI `sam build` command with the `--use-container` to create your deployment package. This option builds a deployment package inside a Docker image that is compatible with the Lambda execution environment.

For more information, see [sam build](#) in the *AWS Serverless Application Model Developer Guide*.

Deployment packages over 50 MB

If your deployment package is larger than 50 MB, upload your function code and dependencies to an Amazon S3 bucket.

You can create a deployment package and upload the .zip file to your Amazon S3 bucket in the AWS Region where you want to create a Lambda function. When you create your Lambda function, specify the S3 bucket name and object key name on the Lambda console, or using the AWS CLI.

To create a bucket using the Amazon S3 console, see [How do I create an S3 Bucket?](#) in the *Amazon Simple Storage Service Console User Guide*.

Lambda console

You can use the Lambda console to configure applications, functions, code signing configurations, and layers. This page provides an explanation for how to edit code using the console editor.

Topics

- [Applications \(p. 21\)](#)
- [Functions \(p. 21\)](#)
- [Code signing \(p. 21\)](#)
- [Layers \(p. 21\)](#)
- [Edit code using the console editor \(p. 21\)](#)

Applications

The [Applications \(p. 960\)](#) page shows you a list of applications that have been deployed using AWS CloudFormation, or other tools including the AWS Serverless Application Model. Filter to find applications based on keywords.

Functions

The functions page shows you a list of functions defined for your account in this region. The initial console flow to create a function depends on whether the function uses a [container image \(p. 111\)](#) or [.zip file archive \(p. 107\)](#) for the deployment package. Many of the optional [configuration tasks \(p. 71\)](#) are common to both types of function.

The console provides a [code editor \(p. 21\)](#) for your convenience.

Code signing

You can attach a [code signing \(p. 241\)](#) configuration to a function. With code signing, you can ensure that the code has been signed by an approved source and has not been altered since signing, and that the code signature has not expired or been revoked.

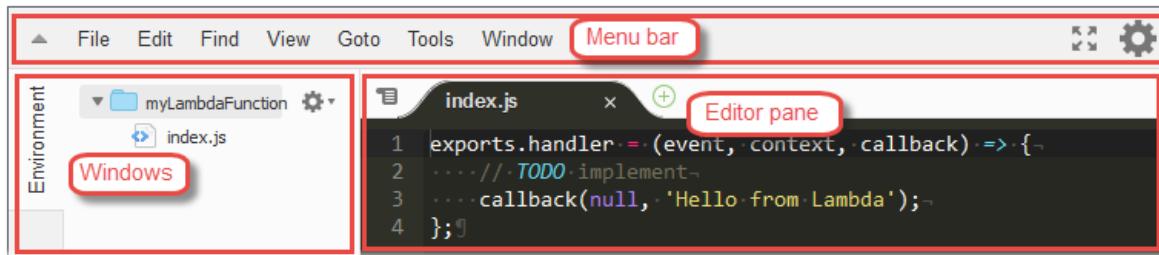
Layers

Create [layers \(p. 93\)](#) to separate your .zip archive function code from its dependencies. A layer is a ZIP archive that contains libraries, a custom runtime, or other dependencies. With layers, you can use libraries in your function without needing to include them in your deployment package.

Edit code using the console editor

You can use the code editor in the AWS Lambda console to write, test, and view the execution results of your Lambda function code. The code editor supports languages that do not require compiling, such as Node.js and Python. The code editor supports only .zip archive deployment packages, and the size of the deployment package must be less than 3 MB.

The code editor includes the *menu bar*, *windows*, and the *editor pane*.



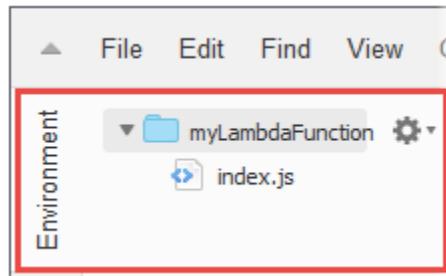
For a list of what the commands do, see the [Menu commands reference](#) in the *AWS Cloud9 User Guide*. Note that some of the commands listed in that reference are not available in the code editor.

Topics

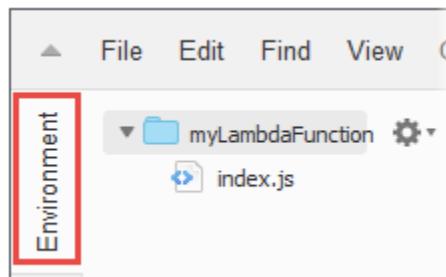
- [Working with files and folders \(p. 22\)](#)
- [Working with code \(p. 24\)](#)
- [Working in fullscreen mode \(p. 27\)](#)
- [Working with preferences \(p. 28\)](#)

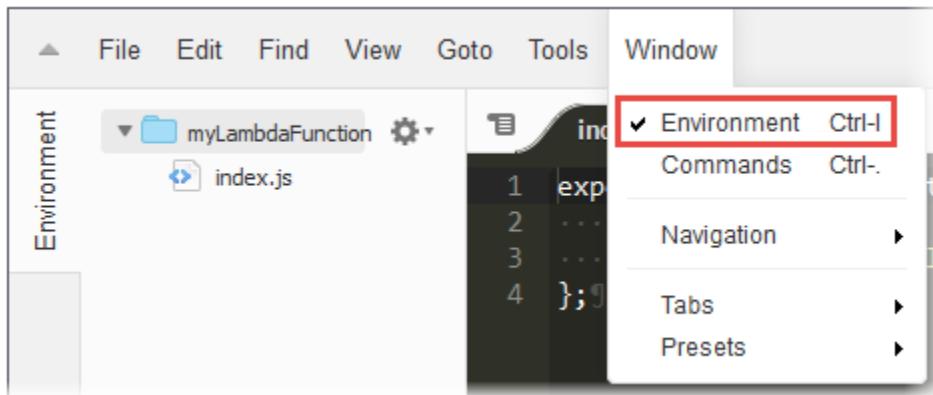
Working with files and folders

You can use the **Environment** window in the code editor to create, open, and manage files for your function.



To show or hide the **Environment** window, choose the **Environment** button. If the **Environment** button is not visible, choose **Window, Environment** on the menu bar.



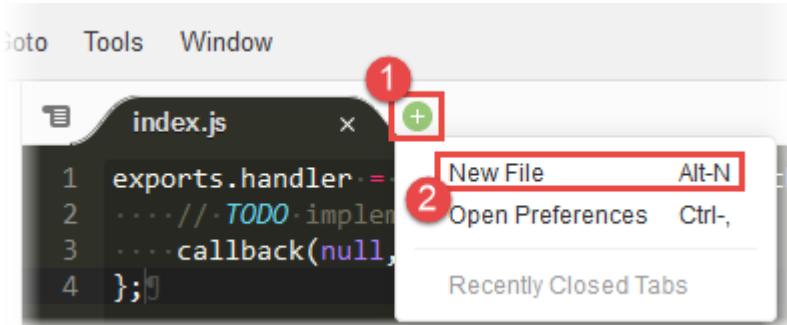


To open a single file and show its contents in the editor pane, double-click the file in the **Environment** window.

To open multiple files and show their contents in the editor pane, choose the files in the **Environment** window. Right-click the selection, and then choose **Open**.

To create a new file, do one of the following:

- In the **Environment** window, right-click the folder where you want the new file to go, and then choose **New File**. Type the file's name and extension, and then press **Enter**.
- Choose **File**, **New File** on the menu bar. When you're ready to save the file, choose **File**, **Save or File**, **Save As** on the menu bar. Then use the **Save As** dialog box that displays to name the file and choose where to save it.
- In the tab buttons bar in the editor pane, choose the + button, and then choose **New File**. When you're ready to save the file, choose **File**, **Save or File**, **Save As** on the menu bar. Then use the **Save As** dialog box that displays to name the file and choose where to save it.



To create a new folder, right-click the folder in the **Environment** window where you want the new folder to go, and then choose **New Folder**. Type the folder's name, and then press **Enter**.

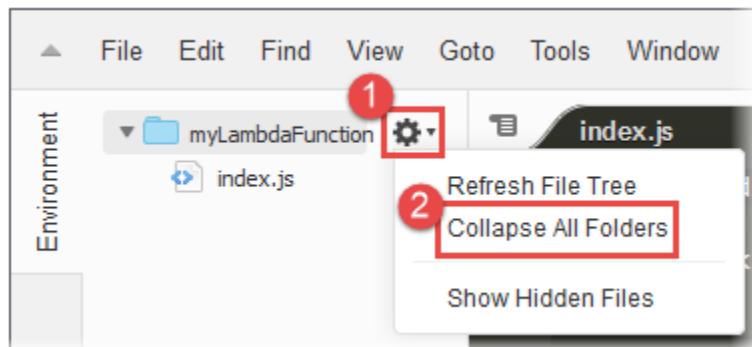
To save a file, with the file open and its contents visible in the editor pane, choose **File**, **Save** on the menu bar.

To rename a file or folder, right-click the file or folder in the **Environment** window. Type the replacement name, and then press **Enter**.

To delete files or folders, choose the files or folders in the **Environment** window. Right-click the selection, and then choose **Delete**. Then confirm the deletion by choosing **Yes** (for a single selection) or **Yes to All**.

To cut, copy, paste, or duplicate files or folders, choose the files or folders in the **Environment** window. Right-click the selection, and then choose **Cut**, **Copy**, **Paste**, or **Duplicate**, respectively.

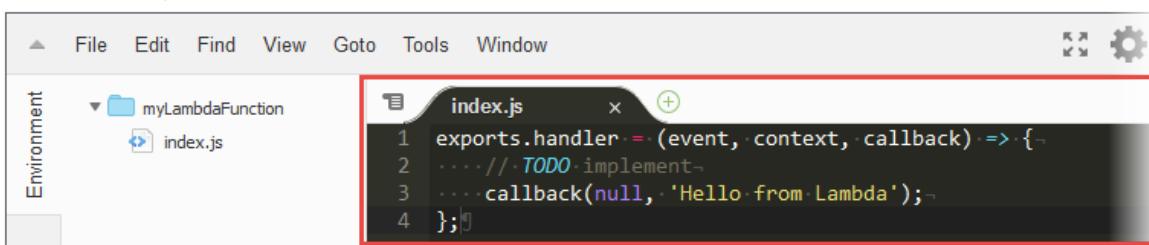
To collapse folders, choose the gear icon in the **Environment** window, and then choose **Collapse All Folders**.



To show or hide hidden files, choose the gear icon in the **Environment** window, and then choose **Show Hidden Files**.

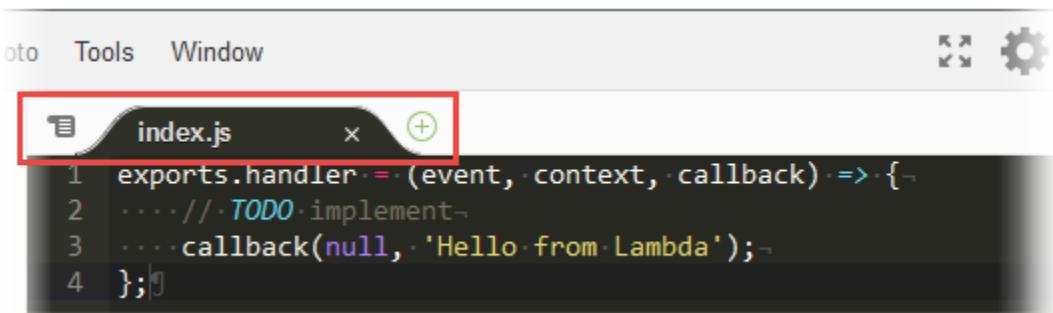
Working with code

Use the editor pane in the code editor to view and write code.



Working with tab buttons

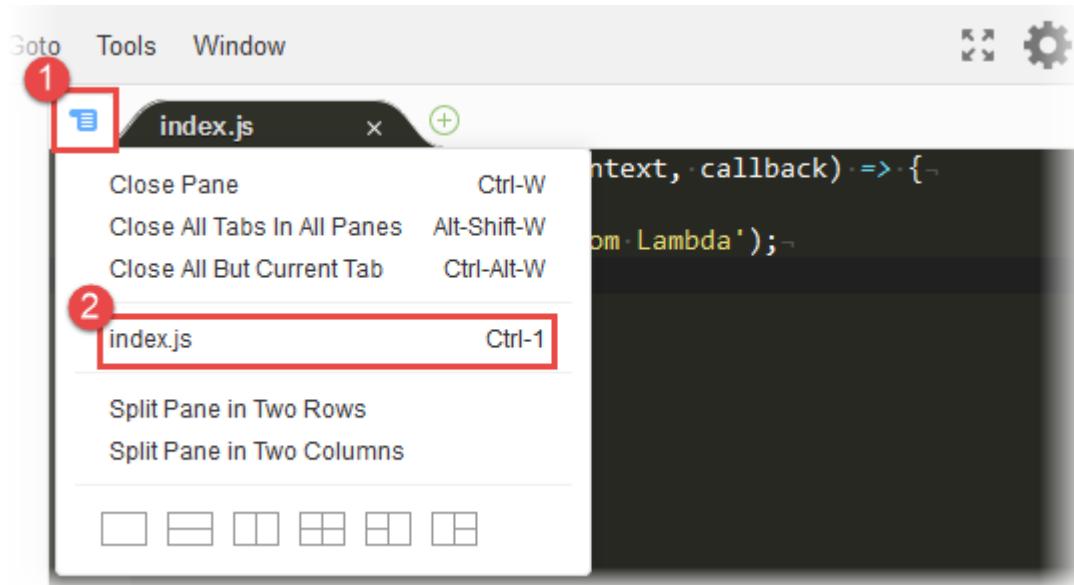
Use the *tab buttons* bar to select, view, and create files.



To display an open file's contents, do one of the following:

- Choose the file's tab.

- Choose the drop-down menu button in the tab buttons bar, and then choose the file's name.



To close an open file, do one of the following:

- Choose the X icon in the file's tab.
- Choose the file's tab. Then choose the drop-down menu button in the tab buttons bar, and choose **Close Pane**.

To close multiple open files, choose the drop-down menu in the tab buttons bar, and then choose **Close All Tabs in All Panes** or **Close All But Current Tab** as needed.

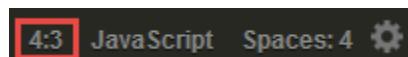
To create a new file, choose the + button in the tab buttons bar, and then choose **New File**. When you're ready to save the file, choose **File, Save** or **File, Save As** on the menu bar. Then use the **Save As** dialog box that displays to name the file and choose where to save it.

Working with the status bar

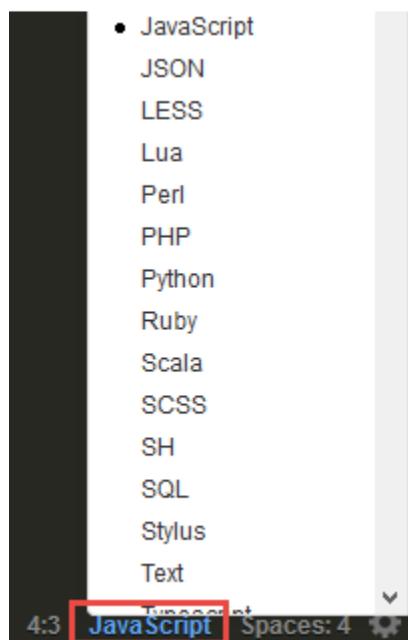
Use the status bar to move quickly to a line in the active file and to change how code is displayed.



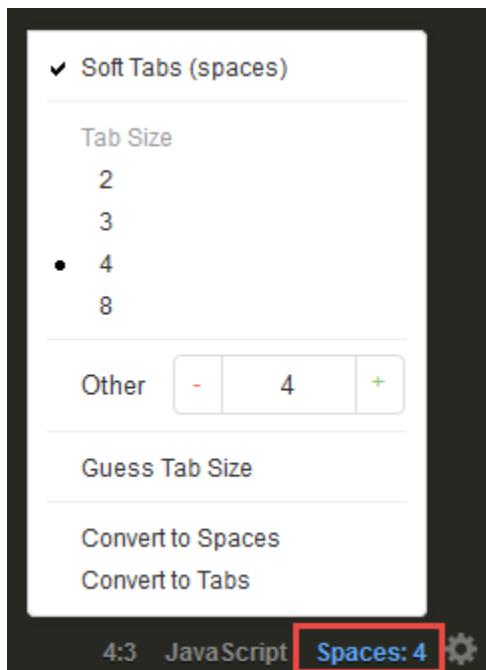
To move quickly to a line in the active file, choose the line selector, type the line number to go to, and then press **Enter**.



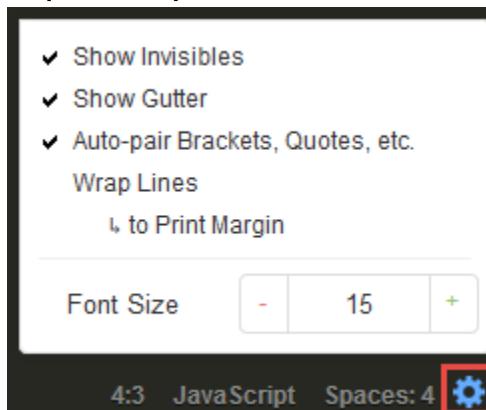
To change the code color scheme in the active file, choose the code color scheme selector, and then choose the new code color scheme.



To change in the active file whether soft tabs or spaces are used, the tab size, or whether to convert to spaces or tabs, choose the spaces and tabs selector, and then choose the new settings.



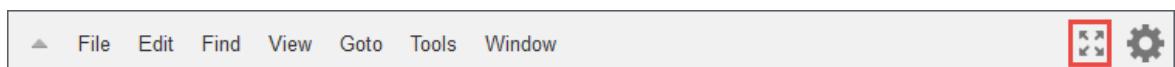
To change for all files whether to show or hide invisible characters or the gutter, auto-pair brackets or quotes, wrap lines, or the font size, choose the gear icon, and then choose the new settings.



Working in fullscreen mode

You can expand the code editor to get more room to work with your code.

To expand the code editor to the edges of the web browser window, choose the **Toggle fullscreen** button in the menu bar.



To shrink the code editor to its original size, choose the **Toggle fullscreen** button again.

In fullscreen mode, additional options are displayed on the menu bar: **Save** and **Test**. Choosing **Save** saves the function code. Choosing **Test** or **Configure Events** enables you to create or edit the function's test events.

Working with preferences

You can change various code editor settings such as which coding hints and warnings are displayed, code folding behaviors, code autocompletion behaviors, and much more.

To change code editor settings, choose the **Preferences** gear icon in the menu bar.



For a list of what the settings do, see the following references in the *AWS Cloud9 User Guide*.

- [Project setting changes you can make](#)
- [User setting changes you can make](#)

Note that some of the settings listed in those references are not available in the code editor.

Lambda instruction set architectures (ARM/x86)

The *instruction set architecture* of a Lambda function determines the type of computer processor that Lambda uses to run the function. Lambda provides a choice of instruction set architectures:

- arm64 – 64-bit ARM architecture, for the AWS Graviton2 processor.
- x86_64 – 64-bit x86 architecture, for x86-based processors.

Note

The arm64 architecture is available in most AWS Regions. For more information, see [AWS Lambda Pricing](#). In the memory prices table, choose the **Arm Price** tab, and then open the **Region** dropdown list to see which AWS Regions support arm64 with Lambda.

Topics

- [Advantages of using arm64 architecture \(p. 29\)](#)
- [Function migration to arm64 architecture \(p. 29\)](#)
- [Function code compatibility with arm64 architecture \(p. 30\)](#)
- [Suggested migration steps \(p. 30\)](#)
- [Configuring the instruction set architecture \(p. 30\)](#)

Advantages of using arm64 architecture

Lambda functions that use arm64 architecture (AWS Graviton2 processor) can achieve significantly better price and performance than the equivalent function running on x86_64 architecture. Consider using arm64 for compute-intensive applications such as high-performance computing, video encoding, and simulation workloads.

The Graviton2 CPU uses the Neoverse N1 core and supports Armv8.2 (including CRC and crypto extensions) plus several other architectural extensions.

Graviton2 reduces memory read time by providing a larger L2 cache per vCPU, which improves the latency performance of web and mobile backends, microservices, and data processing systems. Graviton2 also provides improved encryption performance and supports instruction sets that improve the latency of CPU-based machine learning inference.

For more information about AWS Graviton2, see [AWS Graviton Processor](#).

Function migration to arm64 architecture

When you select a Lambda function to migrate to arm64 architecture, to ensure a smooth migration, make sure that your function meets the following requirements:

- The function currently uses a Lambda Amazon Linux 2 runtime.
- The deployment package contains only open-source components and source code that you control, so that you can make any necessary updates for the migration.
- If the function code includes third-party dependencies, each library or package provides an arm64 version.

Function code compatibility with arm64 architecture

Your Lambda function code must be compatible with the instruction set architecture of the function. Before you migrate a function to arm64 architecture, note the following points about the current function code:

- If you added your function code using the embedded code editor, your code probably runs on either architecture without modification.
- If you uploaded your function code, you must upload new code that is compatible with your target architecture.
- If your function uses layers, you must [check each layer \(p. 248\)](#) to ensure that it is compatible with the new architecture. If a layer is not compatible, edit the function to replace the current layer version with a compatible layer version.
- If your function uses Lambda extensions, you must check each extension to ensure that it is compatible with the new architecture.
- If your function uses a container image deployment package type, you must create a new container image that is compatible with the architecture of the function.

Suggested migration steps

To migrate a Lambda function to the arm64 architecture, we recommend following these steps:

1. Build the list of dependencies for your application or workload. Common dependencies include:
 - All the libraries and packages that the function uses.
 - The tools that you use to build, deploy, and test the function, such as compilers, test suites, continuous integration and continuous delivery (CI/CD) pipelines, provisioning tools, and scripts.
 - The Lambda extensions and third-party tools that you use to monitor the function in production.
2. For each of the dependencies, check the version, and then check whether arm64 versions are available.
3. Build an environment to migrate your application.
4. Bootstrap the application.
5. Test and debug the application.
6. Test the performance of the arm64 function. Compare the performance with the x86_64 version.
7. Update your infrastructure pipeline to support arm64 Lambda functions.
8. Stage your deployment to production.

For example, use [alias routing configuration \(p. 86\)](#) to split traffic between the x86 and arm64 versions of the function, and compare the performance and latency.

For more information about how to create a code environment for arm64 architecture, including language-specific information for Java, Go, .NET, and Python, see the [Getting started with AWS Graviton](#) GitHub repository.

Configuring the instruction set architecture

You can configure the instruction set architecture for new and existing Lambda functions using the Lambda console, AWS SDKs, AWS Command Line Interface (AWS CLI), or AWS CloudFormation. Follow these steps to change the instruction set architecture for an existing Lambda function from the console.

1. Open the [Functions page](#) of the Lambda console.

2. Choose the name of the function that you want to configure the instruction set architecture for.
3. On the main **Code** tab, for the **Runtime settings** section, choose **Edit**.
4. Under **Architecture**, choose the instruction set architecture you want your function to use.
5. Choose **Save**.

Lambda provides the following runtimes for the arm64 architecture. These runtimes all use the Amazon Linux 2 operating system.

- Node.js 12, Node.js 14, Node.js 16, Node.js 18
- Python 3.8, Python 3.9
- Java 8 (AL2), Java 11
- .NET Core 3.1, .NET 6
- Ruby 2.7
- Custom Runtime on Amazon Linux 2

Note

Runtimes that use the Amazon Linux operating system, such as Go 1.x, do not support the arm64 architecture. To use arm64 architecture, you can run Go with the provided.al2 runtime. For example, see [Build a Go function for the provided.al2 runtime \(p. 463\)](#) or [Create a Go image from the provided.al2 base image \(p. 469\)](#).

For an example of how to create a function with arm64 architecture, see [AWS Lambda Functions Powered by AWS Graviton2 Processor](#).

Additional Lambda features

Lambda provides a management console and API for managing and invoking functions. It provides runtimes that support a standard set of features so that you can easily switch between languages and frameworks, depending on your needs. In addition to functions, you can also create versions, aliases, layers, and custom runtimes.

Advanced features

- [Scaling \(p. 32\)](#)
- [Concurrency controls \(p. 32\)](#)
- [Function URLs \(p. 32\)](#)
- [Asynchronous invocation \(p. 33\)](#)
- [Event source mappings \(p. 33\)](#)
- [Destinations \(p. 34\)](#)
- [Function blueprints \(p. 35\)](#)
- [Testing and deployment tools \(p. 35\)](#)
- [Application templates \(p. 36\)](#)

Scaling

Lambda manages the infrastructure that runs your code, and scales automatically in response to incoming requests. When your function is invoked more quickly than a single instance of your function can process events, Lambda scales up by running additional instances. When traffic subsides, inactive instances are frozen or stopped. You pay only for the time that your function is initializing or processing events.

For more information, see [Lambda function scaling \(p. 197\)](#).

Concurrency controls

Use concurrency settings to ensure that your production applications are highly available and highly responsive.

To prevent a function from using too much concurrency, and to reserve a portion of your account's available concurrency for a function, use *reserved concurrency*. Reserved concurrency splits the pool of available concurrency into subsets. A function with reserved concurrency only uses concurrency from its dedicated pool.

To enable functions to scale without fluctuations in latency, use *provisioned concurrency*. For functions that take a long time to initialize, or that require extremely low latency for all invocations, provisioned concurrency enables you to pre-initialize instances of your function and keep them running at all times. Lambda integrates with Application Auto Scaling to support autoscaling for provisioned concurrency based on utilization.

For more information, see [Configuring reserved concurrency \(p. 210\)](#).

Function URLs

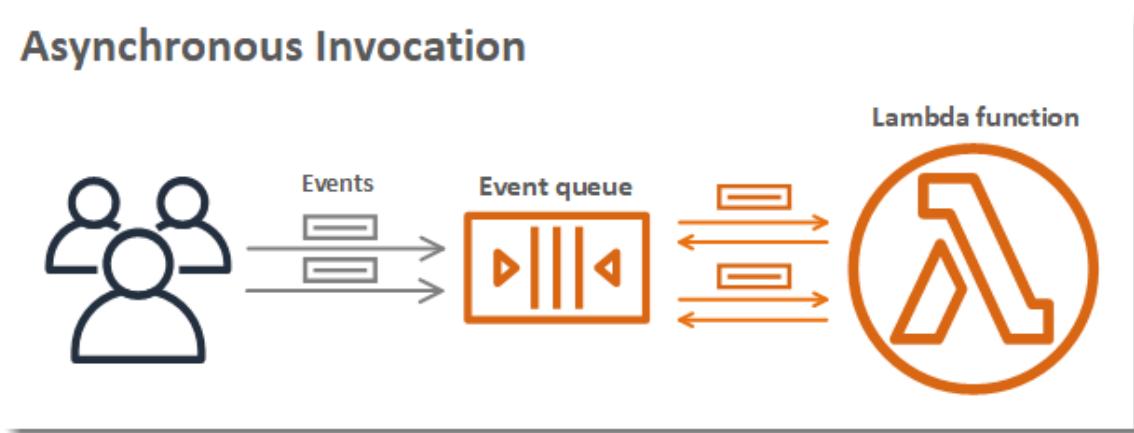
Lambda offers built-in HTTP(S) endpoint support through *function URLs*. With function URLs, you can assign a dedicated HTTP endpoint to your Lambda function. When your function URL is configured, you can use it to invoke your function through a web browser, curl, Postman, or any HTTP client.

You can add a function URL to an existing function, or create a new function with a function URL. For more information, see [Invoking Lambda function URLs \(p. 177\)](#).

Asynchronous invocation

When you invoke a function, you can choose to invoke it synchronously or asynchronously. With [synchronous invocation \(p. 120\)](#), you wait for the function to process the event and return a response. With asynchronous invocation, Lambda queues the event for processing and returns a response immediately.

Asynchronous Invocation



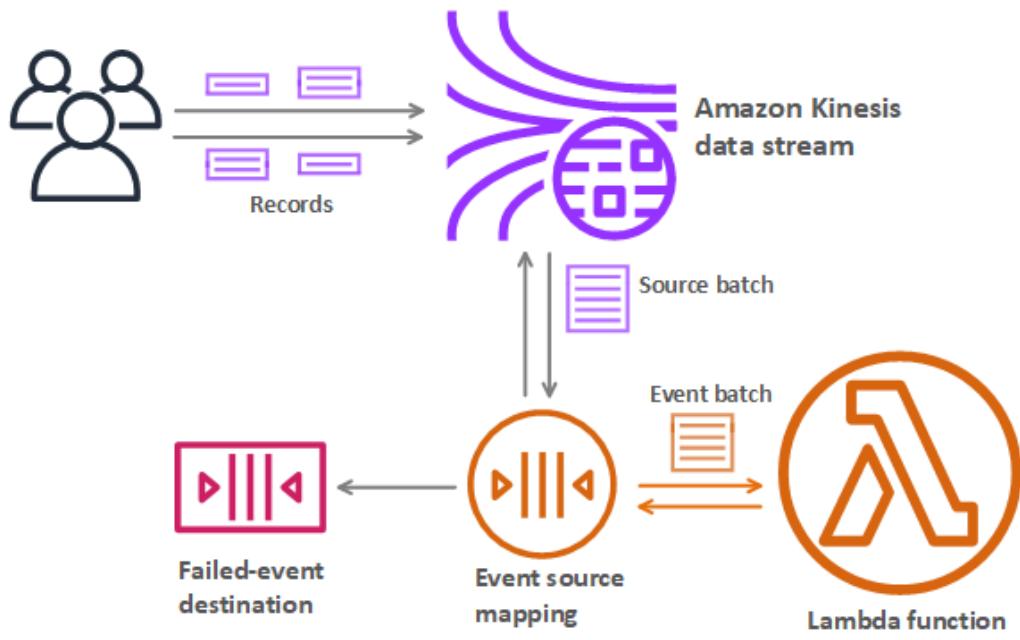
For asynchronous invocations, Lambda handles retries if the function returns an error or is throttled. To customize this behavior, you can configure error handling settings on a function, version, or alias. You can also configure Lambda to send events that failed processing to a [dead-letter queue](#), or to send a record of any invocation to a [destination \(p. 34\)](#).

For more information, see [Asynchronous invocation \(p. 123\)](#).

Event source mappings

To process items from a stream or queue, you can create an [event source mapping](#). An event source mapping is a resource in Lambda that reads items from an Amazon Simple Queue Service (Amazon SQS) queue, an Amazon Kinesis stream, or an Amazon DynamoDB stream, and sends the items to your function in batches. Each event that your function processes can contain hundreds or thousands of items.

Event Source Mapping with Kinesis Stream



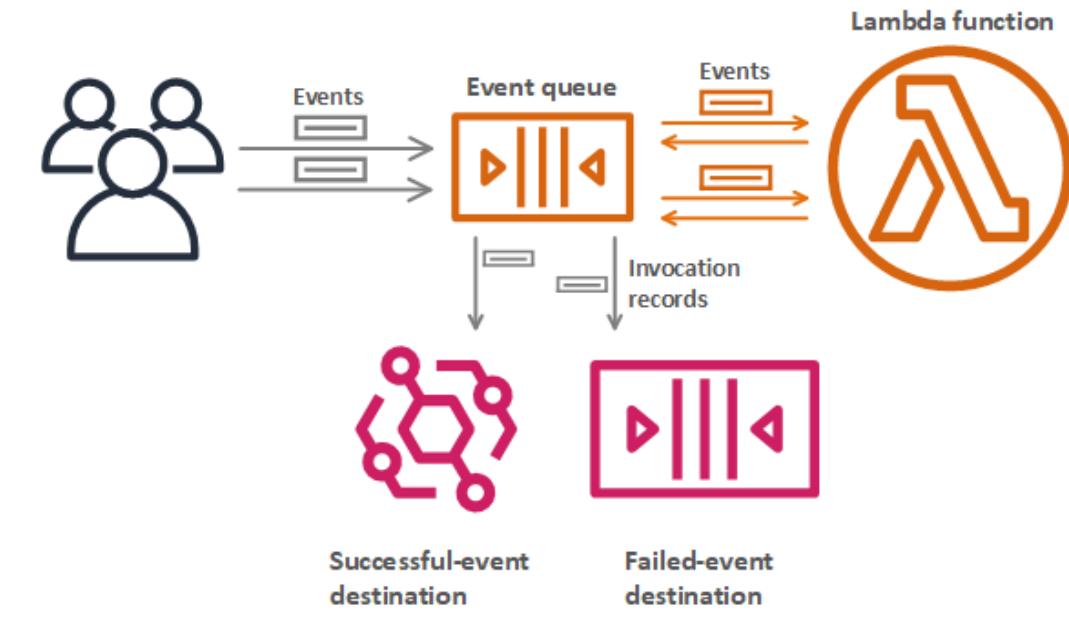
Event source mappings maintain a local queue of unprocessed items and handle retries if the function returns an error or is throttled. You can configure an event source mapping to customize batching behavior and error handling, or to send a record of items that fail processing to a destination.

For more information, see [Lambda event source mappings \(p. 131\)](#).

Destinations

A destination is an AWS resource that receives invocation records for a function. For [asynchronous invocation \(p. 33\)](#), you can configure Lambda to send invocation records to a queue, topic, function, or event bus. You can configure separate destinations for successful invocations and events that failed processing. The invocation record contains details about the event, the function's response, and the reason that the record was sent.

Destinations for Asynchronous Invocation



For [event source mappings \(p. 33\)](#) that read from streams, you can configure Lambda to send a record of batches that failed processing to a queue or topic. A failure record for an event source mapping contains metadata about the batch, and it points to the items in the stream.

For more information, see [Configuring destinations for asynchronous invocation \(p. 125\)](#) and the error handling sections of [Using AWS Lambda with Amazon DynamoDB \(p. 635\)](#) and [Using AWS Lambda with Amazon Kinesis \(p. 684\)](#).

Function blueprints

When you create a function in the Lambda console, you can choose to start from scratch, use a blueprint, or use a [container image \(p. 18\)](#). A blueprint provides sample code that shows how to use Lambda with an AWS service or a popular third-party application. Blueprints include sample code and function configuration presets for Node.js and Python runtimes.

Blueprints are provided for use under the [Amazon Software License](#). They are available only in the Lambda console.

Testing and deployment tools

Lambda supports deploying code as is or as [container images \(p. 18\)](#). You can use AWS services and popular community tools like the Docker command line interface (CLI) to author, build, and deploy your Lambda functions. To set up the Docker CLI, see [Get Docker](#) on the Docker Docs website. For an introduction to using Docker with AWS, see [Getting started with Amazon ECR using the AWS CLI](#) in the [Amazon Elastic Container Registry User Guide](#).

The [AWS CLI](#) and [AWS SAM CLI](#) are command line tools for managing Lambda application stacks. In addition to commands for managing application stacks with the AWS CloudFormation API, the AWS CLI supports higher-level commands that simplify tasks such as uploading deployment packages and

updating templates. The AWS SAM CLI provides additional functionality, including validating templates, testing locally, and integrating with CI/CD systems.

- [Installing the AWS SAM CLI](#)
- [Testing and debugging serverless applications with AWS SAM](#)
- [Deploying serverless applications using CI/CD systems with AWS SAM](#)

Application templates

You can use the Lambda console to create an application with a continuous delivery pipeline. Application templates in the Lambda console include code for one or more functions, an application template that defines functions and supporting AWS resources, and an infrastructure template that defines an AWS CodePipeline pipeline. The pipeline has build and deploy stages that run every time you push changes to the included Git repository.

Application templates are provided for use under the [MIT No Attribution](#) license. They are available only in the Lambda console.

For more information, see [Creating an application with continuous delivery in the Lambda console \(p. 964\)](#).

Learn how to build serverless solutions

Tip

To learn how to build **serverless solutions**, check out the [Serverless Developer Guide](#).

Lambda runtimes

Lambda supports multiple languages through the use of *runtimes*. A runtime provides a language-specific environment that relays invocation events, context information, and responses between Lambda and the function. You can use runtimes that Lambda provides, or build your own.

Each major programming language release has a separate runtime, with a unique *runtime identifier*, such as `python3.10` or `nodejs18.x`. To configure a function to use a new major language version, you need to change the runtime identifier. Since AWS Lambda cannot guarantee backward compatibility between major versions, this is a customer-driven operation.

For a [function defined as a container image \(p. 111\)](#), you choose a runtime and the Linux distribution when you [create the container image \(p. 881\)](#). To change the runtime, you create a new container image.

When you use a .zip file archive for the deployment package, you choose a runtime when you create the function. To change the runtime, you can [update your function's configuration \(p. 107\)](#). The runtime is paired with one of the Amazon Linux distributions. The underlying execution environment provides additional libraries and [environment variables \(p. 76\)](#) that you can access from your function code.

Amazon Linux 2

- Image – Custom
- Linux kernel – 4.14

Amazon Linux

- Image – [amzn-ami-hvm-2018.03.0.20220802.0-x86_64-gp2](#)
- Linux kernel – 4.14

Lambda invokes your function in an [execution environment \(p. 14\)](#). The execution environment provides a secure and isolated runtime environment that manages the resources required to run your function. Lambda re-uses the execution environment from a previous invocation if one is available, or it can create a new execution environment.

To use other languages in Lambda, you can implement a [custom runtime \(p. 59\)](#). The Lambda execution environment provides a [runtime interface \(p. 54\)](#) for getting invocation events and sending responses. You can deploy a custom runtime alongside your function code, or in a [layer \(p. 93\)](#).

Note

For new regions, Lambda will not support runtimes that are set to be deprecated within the next six months.

Supported Runtimes

Name	Identifier	SDK	Operating system	Architectures	Deprecation (Phase 1)
Node.js 18	<code>nodejs18.x</code>	3.188.0	Amazon Linux 2	x86_64, arm64	
Node.js 16	<code>nodejs16.x</code>	2.1083.0	Amazon Linux 2	x86_64, arm64	

Name	Identifier	SDK	Operating system	Architectures	Deprecation (Phase 1)
Node.js 14	nodejs14.x	2.1055.0	Amazon Linux 2	x86_64, arm64	
Node.js 12	nodejs12.x	2.1055.0	Amazon Linux 2	x86_64, arm64	Mar 31, 2023
Python 3.10	python3.10	boto3-1.26.90 botocore-1.29.902	Amazon Linux	x86_64, arm64	
Python 3.9	python3.9	boto3-1.26.90 botocore-1.29.902	Amazon Linux	x86_64, arm64	
Python 3.8	python3.8	boto3-1.26.90 botocore-1.29.902	Amazon Linux	x86_64, arm64	
Python 3.7	python3.7	boto3-1.26.90 botocore-1.29.90	Amazon Linux	x86_64	
Java 17	java17		Amazon Linux 2	x86_64, arm64	
Java 11	java11		Amazon Linux 2	x86_64, arm64	
Java 8	java8.a12		Amazon Linux 2	x86_64, arm64	
Java 8	java8		Amazon Linux	x86_64	
.NET Core 3.1	dotnetcore3.1		Amazon Linux 2	x86_64, arm64	Apr 3, 2023
.NET 7	dotnet7		Amazon Linux 2	x86_64, arm64	
.NET 6	dotnet6		Amazon Linux 2	x86_64, arm64	
.NET 5	dotnet5.0		Amazon Linux 2	x86_64	
Go 1.x	go1.x		Amazon Linux	x86_64	
Ruby 2.7*	ruby2.7	3.1.0	Amazon Linux 2	x86_64, arm64	Nov 15, 2023
Custom Runtime	provided.a12		Amazon Linux 2	x86_64, arm64	
Custom Runtime	provided		Amazon Linux	x86_64	

*The deprecation date for Ruby 2.7 above is an estimate. Lambda will support the Ruby 2.7 runtime for at least 6 months after the general availability (GA) release of the Ruby 3.2 runtime.

Lambda keeps managed runtimes up to date with patches and support for minor version releases. For more information see [Lambda runtime updates](#).

Runtime deprecation policy

[Lambda runtimes \(p. 37\)](#) for .zip file archives are built around a combination of operating system, programming language, and software libraries that are subject to maintenance and security updates. When security updates are no longer available for a component of a runtime, Lambda deprecates the runtime.

Deprecation (end of support) for a runtime occurs in two phases.

Phase 1 - Lambda no longer applies security patches or other updates to the runtime. You can no longer **create** functions that use the runtime, but you can continue to update existing functions. This includes updating the runtime, and rolling back to the previous runtime. Note that functions that use a deprecated runtime are no longer eligible for technical support.

Phase 2 - you can no longer **create or update** functions that use the runtime. To update a function, you need to migrate it to a supported runtime. After you migrate the function to a supported runtime, you cannot rollback the function to the previous runtime. Phase 2 starts at least 30 days after the start of Phase 1.

Lambda does not block invocations of functions that use deprecated runtime. Function invocations continue indefinitely after the runtime reaches end of support. However, AWS strongly recommends that you migrate functions to a supported runtime so that you continue to receive security patches and remain eligible for technical support.

In the table below, each phase starts at midnight (Pacific time zone) on the specified date. The following runtimes have reached end of support:

Deprecated runtimes

Name	Identifier	Operating system	Deprecation Phase 1	Deprecation Phase 2
Python 3.6	python3.6	Amazon Linux	Jul 18, 2022	Aug 29, 2022
Python 2.7	python2.7	Amazon Linux	Jul 15, 2021	May 30, 2022
.NET Core 2.1	dotnetcore2.1	Amazon Linux	Jan 5, 2022	Apr 13, 2022
Ruby 2.5	ruby2.5	Amazon Linux	Jul 30, 2021	Mar 31, 2022
Node.js 10	nodejs10.x	Amazon Linux 2	Jul 30, 2021	Feb 14, 2022
Node.js 8.10	nodejs8.10	Amazon Linux		Mar 6, 2020
Node.js 4.3	nodejs4.3	Amazon Linux		Mar 5, 2020
Node.js 6.10	nodejs6.10	Amazon Linux		Aug 12, 2019
.NET Core 1.0	dotnetcore1.0	Amazon Linux		Jul 30, 2019
.NET Core 2.0	dotnetcore2.0	Amazon Linux		May 30, 2019
Node.js 4.3 edge	nodejs4.3-edge	Amazon Linux		Apr 30, 2019
Node.js 0.10	nodejs	Amazon Linux		Oct 31, 2016

In almost all cases, the end-of-life date of a language version or operating system is known well in advance. The links below give end-of-life schedules for each language that Lambda supports as a managed runtime. In addition, Trusted Advisor includes a check that provides [120 days' notice of](#)

[upcoming Lambda runtime end of support](#), and Lambda notifies you by email if you have functions using a runtime that is scheduled for end of support in the next 60 days.

Language and framework support policies

- **Node.js** – [github.com](#)
- **Python** – [devguide.python.org](#)
- **Ruby** – [www.ruby-lang.org](#)
- **Java** – [www.oracle.com](#) and [Corretto FAQs](#)
- **Go** – [golang.org](#)
- **.NET Core** – [dotnet.microsoft.com](#)

Lambda runtime updates

Lambda keeps each managed runtime up to date with security updates, bug fixes, new features, performance enhancements, and support for minor version releases. These runtime updates are published as *runtime versions*. Lambda applies runtime updates to functions by migrating the function from an earlier runtime version to a new runtime version.

By default, for functions using managed runtimes, Lambda applies runtime updates automatically. With automatic runtime updates, Lambda takes on the operational burden of patching the runtime versions. For most customers, automatic updates are the right choice. For more information, see [Runtime management controls \(p. 41\)](#).

Lambda also publishes each new runtime version as a container image. To update runtime versions for container-based functions, you must [create a new container image \(p. 881\)](#) from the updated base image and redeploy your function.

Each runtime version is associated with a version number and an ARN (Amazon Resource Name). Runtime version numbers use a numbering scheme that Lambda defines, independent of the version numbers that the programming language uses. The runtime version ARN is a unique identifier for each runtime version.

You can view the ARN of your function's current runtime version in the INIT_START line of your function logs and [in the Lambda console \(p. 45\)](#).

Runtime versions should not be confused with runtime identifiers. Each runtime has a unique *runtime identifier*, such as python3.9 or nodejs18.x. These correspond to each major programming language release. Runtime versions describe the patch version of an individual runtime.

Note

The ARN for the same runtime version number can vary between AWS Regions and CPU architectures.

Topics

- [Runtime management controls \(p. 41\)](#)
- [Two-phase runtime version rollout \(p. 42\)](#)
- [Roll back a runtime version \(p. 42\)](#)
- [Identifying runtime version changes \(p. 43\)](#)
- [Configure runtime management settings \(p. 45\)](#)
- [Shared responsibility model \(p. 46\)](#)
- [High-compliance applications \(p. 47\)](#)
- [Supported Regions \(p. 47\)](#)

Runtime management controls

Lambda strives to provide runtime updates that are backward compatible with existing functions. However, as with software patching, there are rare cases in which a runtime update can negatively impact an existing function. For example, security patches can expose an underlying issue with an existing function that depends on the previous, insecure behavior. Lambda runtime management controls help reduce the risk of impact to your workloads in the rare event of a runtime version incompatibility. For each [function version \(p. 83\)](#) (\$LATEST or published version), you can choose one of the following runtime update modes:

- **Auto (default)** – Automatically update to the most recent and secure runtime version using [Two-phase runtime version rollout \(p. 42\)](#). We recommend this mode for most customers so that you always benefit from runtime updates.

- **Function update** – Update to the most recent and secure runtime version when you update your function. When you update your function, Lambda updates the runtime of your function to the most recent and secure runtime version. This approach synchronizes runtime updates with function deployments, giving you control over when Lambda applies runtime updates. With this mode, you can detect and mitigate rare runtime update incompatibilities early. When using this mode, you must regularly update your functions to keep their runtime up to date.
- **Manual** – Manually update your runtime version. You specify a runtime version in your function configuration. The function uses this runtime version indefinitely. In the rare case in which a new runtime version is incompatible with an existing function, you can use this mode to roll back your function to an earlier runtime version. We recommend against using **Manual** mode to try to achieve runtime consistency across deployments. For more information, see [Roll back a runtime version \(p. 42\)](#).

Responsibility for applying runtime updates to your functions varies according to which runtime update mode you choose. For more information, see [Shared responsibility model \(p. 46\)](#).

Two-phase runtime version rollout

Lambda introduces new runtime versions in the following order:

1. In the first phase, Lambda applies the new runtime version whenever you create or update a function. A function gets updated when you call the [UpdateFunctionCode \(p. 1367\)](#) or [UpdateFunctionConfiguration \(p. 1377\)](#) API operations.
2. In the second phase, Lambda updates any function that uses the **Auto** runtime update mode and that hasn't already been updated to the new runtime version.

The overall duration of the rollout process varies according to multiple factors, including the severity of any security patches included in the runtime update.

If you're actively developing and deploying your functions, you will most likely pick up new runtime versions during the first phase. This synchronizes runtime updates with function updates. In the rare event that the latest runtime version negatively impacts your application, this approach lets you take prompt corrective action. Functions that aren't in active development still receive the operational benefit of automatic runtime updates during the second phase.

This approach doesn't affect functions set to **Function update** or **Manual** mode. Functions using **Function update** mode receive the latest runtime updates only when you create or update them. Functions using **Manual** mode don't receive runtime updates.

Lambda publishes new runtime versions in a gradual, rolling fashion across AWS Regions. If your functions are set to **Auto** or **Function update** modes, it's possible that functions deployed at the same time to different Regions, or at different times in the same Region, will pick up different runtime versions. Customers who require guaranteed runtime version consistency across their environments should use [use container images to deploy their Lambda functions \(p. 111\)](#). The **Manual** mode is designed as a temporary mitigation to enable runtime version rollback in the rare event that a runtime version is incompatible with your function.

Roll back a runtime version

In the rare event that a new runtime version is incompatible with your existing function, you can roll back its runtime version to an earlier one. This keeps your application working and minimizes disruption, providing time to remedy the incompatibility before returning to the latest runtime version.

Lambda doesn't impose a time limit on how long you can use any particular runtime version. However, we strongly recommend updating to the latest runtime version as soon as possible to benefit from the

latest security patches, performance improvements, and features. Lambda provides the option to roll back to an earlier runtime version only as a temporary mitigation in the rare event of a runtime update compatibility issue. Functions using an earlier runtime version for an extended period may eventually experience degraded performance or issues, such as a certificate expiry, which can cause them to stop working properly.

You can roll back a runtime version in the following ways:

- [Using the Manual runtime update mode \(p. 43\)](#)
- [Using published function versions \(p. 43\)](#)

For more information, see [Introducing AWS Lambda runtime management controls](#) on the AWS Compute Blog.

Roll back a runtime version using **Manual** runtime update mode

If you're using the **Auto** runtime version update mode, or you're using the \$LATEST runtime version, you can roll back your runtime version using the **Manual** mode. For the [function version \(p. 83\)](#) you want to roll back, change the runtime version update mode to **Manual** and specify the ARN of the previous runtime version. For more information about finding the ARN of the previous runtime version, see [Identifying runtime version changes \(p. 43\)](#).

Note

If the \$LATEST version of your function is configured to use **Manual** mode, then you can't change the CPU architecture or runtime version that your function uses. To make these changes, you must change to **Auto** or **Function update** mode.

Roll back a runtime version using published function versions

Published [function versions \(p. 83\)](#) are an immutable snapshot of the \$LATEST function code and configuration at the time that you created them. In **Auto** mode, Lambda automatically updates the runtime version of published function versions during phase two of the runtime version rollout. In **Function update** mode, Lambda doesn't update the runtime version of published function versions.

Published function versions using **Function update** mode therefore create a static snapshot of the function code, configuration, and runtime version. By using **Function update** mode with function versions, you can synchronize runtime updates with your deployments. You can also coordinate rollback of code, configuration, and runtime versions by redirecting traffic to an earlier published function version. You can integrate this approach into your continuous integration and continuous delivery (CI/CD) for fully automatic rollback in the rare event of runtime update incompatibility. When using this approach, you must update your function regularly and publish new function versions to pick up the latest runtime updates. For more information, see [Shared responsibility model \(p. 46\)](#).

Identifying runtime version changes

The runtime version number and ARN are logged in the INIT_START log line, which Lambda emits to CloudWatch Logs each time that it creates a new [execution environment \(p. 10\)](#). Because the execution environment uses the same runtime version for all function invocations, Lambda emits the INIT_START log line only when Lambda executes the init phase. Lambda doesn't emit this log line for each function invocation. Lambda emits the log line to CloudWatch Logs, but it is not visible in the console.

Example Example INIT_START log line

```
INIT_START Runtime Version: python:3.9.v14    Runtime Version ARN: arn:aws:lambda:eu-south-1::runtime:7b620fc2e66107a1046b140b9d320295811af3ad5d4c6a011fad1fa65127e9e6I
```

Note

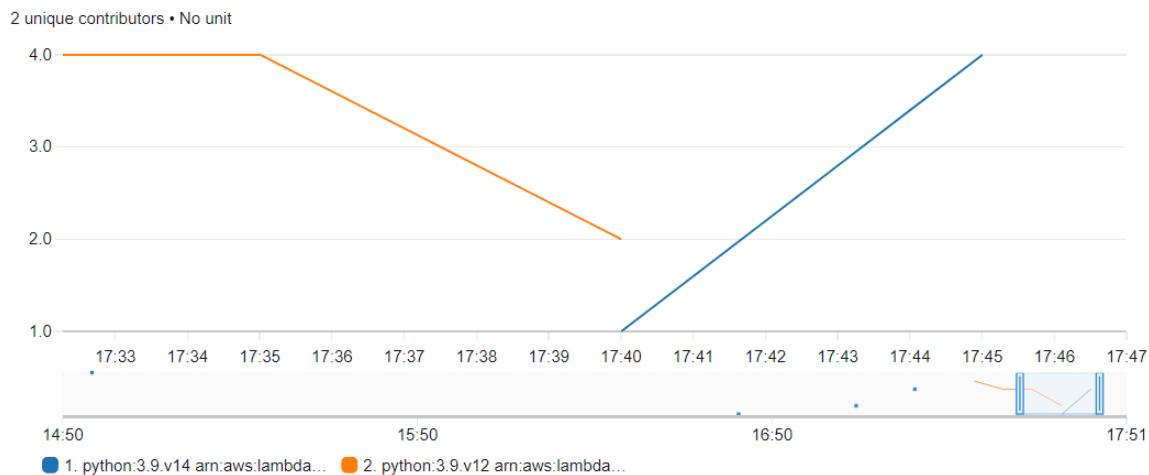
The INIT_START log line is only emitted for functions created or updated after January 24, 2023.

Rather than working directly with the logs, you can use [Amazon CloudWatch Contributor Insights](#) to identify transitions between runtime versions. The following rule counts the distinct runtime versions from each INIT_START log line. To use the rule, replace the example log group name /aws/lambda/* with the appropriate prefix for your function or group of functions.

```
{  
  "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"  
  }  
}
```

The following CloudWatch Contributor Insights report shows an example of a runtime version transition as captured by the preceding rule. The orange line shows execution environment initialization for the earlier runtime version (**python:3.9.v12**), and the blue line shows execution environment initialization for the new runtime version (**python:3.9.v14**).

Top 2 of 2 unique contributors



Configure runtime management settings

You can configure runtime management settings using the Lambda console or the AWS Command Line Interface (AWS CLI).

Note

You can configure runtime management settings separately for each [function version \(p. 83\)](#).

To configure how Lambda updates your runtime version (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of a function.
3. On the **Code** tab, under **Runtime settings**, choose **Edit runtime management configuration**.
4. Under **Runtime management configuration**, choose one of the following:
 - To have your function update to the latest runtime version automatically, choose **Auto**.
 - To have your function update to the latest runtime version when you change the function, choose **Function update**.
 - To have your function update to the latest runtime version only when you change the runtime version ARN, choose **Manual**.

Note

You can find the runtime version ARN under **Runtime management configuration**. You can also find the ARN in the INIT_START line of your function logs.

5. Choose **Save**.

To configure how Lambda updates your runtime version (AWS CLI)

To configure runtime management for a function, you can use the [`put-runtime-management-config`](#) AWS CLI command, together with the runtime update mode. When using Manual mode, you must also provide the runtime version ARN.

```
aws lambda put-runtime-management-config --function-name arn:aws:lambda:eu-west-1:069549076217:function:myfunction --update-runtime-on Manual --runtime-version-arn arn:aws:lambda:eu-west-1::runtime:8efff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1
```

You should see output similar to the following:

```
{
  "UpdateRuntimeOn": "Manual",
  "FunctionArn": "arn:aws:lambda:eu-west-1:069549076217:function:myfunction",
  "RuntimeVersionArn": "arn:aws:lambda:eu-
west-1::runtime:8efff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1"
}
```

Shared responsibility model

Lambda is responsible for curating and publishing security updates for all supported managed runtimes and container images. Responsibility for updating existing functions to use the latest runtime version varies depending on which runtime update mode you use.

Lambda is responsible for applying runtime updates to all functions configured to use the **Auto** runtime update mode.

For functions configured with the **Function update** runtime update mode, you're responsible for regularly updating your function. Lambda is responsible for applying runtime updates when you make those updates. If you don't update your function, then Lambda doesn't update the runtime. If you don't regularly update your function, then we strongly recommend configuring it for automatic runtime updates so that it continues to receive security updates.

For functions configured to use the **Manual** runtime update mode, you're responsible for updating your function to use the latest runtime version. We strongly recommend that you use this mode only to roll back the runtime version as a temporary mitigation in the rare event of runtime update incompatibility. We also recommend that you change to **Auto** mode as quickly as possible to minimize the time in which your functions aren't patched.

If you're [using container images to deploy your functions \(p. 111\)](#), then Lambda is responsible for publishing updated base images. In this case, you're responsible for rebuilding your function's container image from the latest base image and redeploying the container image.

This is summarized in the following table:

Deployment mode	Lambda's responsibility	Customer's responsibility
Managed runtime, Auto mode	Publish new runtime versions containing the latest patches. Apply runtime patches to existing functions.	Roll back to a previous runtime version in the rare event of a runtime update compatibility issue.
Managed runtime, Function update mode	Publish new runtime versions containing the latest patches.	Update functions regularly to pick up the latest runtime version. Switch a function to Auto mode when you're not regularly updating the function. Roll back to a previous runtime version in the rare event of a runtime update compatibility issue.
Managed runtime, Manual mode	Publish new runtime versions containing the latest patches.	Use this mode only for temporary runtime rollback in the rare event of a runtime update compatibility issue.

Deployment mode	Lambda's responsibility	Customer's responsibility
		Switch functions to Auto or Function update mode and the latest runtime version as soon as possible.
Container image	Publish new container images containing the latest patches.	Redeploy functions regularly using the latest container base image to pick up the latest patches.

For more information about shared responsibility with AWS, see [Shared Responsibility Model](#) on the AWS Cloud Security site.

High-compliance applications

To meet patching requirements, Lambda customers typically rely on automatic runtime updates. If your application is subject to strict patching freshness requirements, you may want to limit use of earlier runtime versions. You can restrict Lambda's runtime management controls by using AWS Identity and Access Management (IAM) to deny users in your AWS account access to the [PutRuntimeManagementConfig \(p. 1337\)](#) API operation. This operation is used to choose the runtime update mode for a function. Denying access to this operation causes all functions to default to the **Auto** mode. You can apply this restriction across your organization by using a [service control policies \(SCP\)](#). In the event that you must roll back a function to an earlier runtime version, you can grant a policy exception on a case-by-case basis.

Supported Regions

Lambda supports runtime version numbers, runtime management controls, the INIT_START log line, and two-phase version rollout in the following AWS Regions:

- US East (N. Virginia)
- US East (Ohio)
- US West (N. California)
- US West (Oregon)
- Europe (Ireland)
- Europe (London)
- Europe (Paris)
- Europe (Stockholm)
- Europe (Frankfurt)
- Europe (Milan)
- Asia Pacific (Hong Kong)
- Asia Pacific (Tokyo)
- Asia Pacific (Seoul)
- Asia Pacific (Osaka)
- Asia Pacific (Mumbai)
- Asia Pacific (Singapore)
- Asia Pacific (Sydney)
- Asia Pacific (Jakarta)
- Canada (Central)
- South America (São Paulo)

- Africa (Cape Town)
- Middle East (Bahrain)

In other Regions, Lambda automatically applies runtime updates to all functions in a Region-by-Region deployment sequence.

Modifying the runtime environment

You can use [internal extensions \(p. 894\)](#) to modify the runtime process. Internal extensions are not separate processes—they run as part of the runtime process.

Lambda provides language-specific [environment variables \(p. 76\)](#) that you can set to add options and tools to the runtime. Lambda also provides [wrapper scripts \(p. 51\)](#), which allow Lambda to delegate the runtime startup to your script. You can create a wrapper script to customize the runtime startup behavior.

Language-specific environment variables

Lambda supports configuration-only ways to enable code to be pre-loaded during function initialization through the following language-specific environment variables:

- **JAVA_TOOL_OPTIONS** – On Java, Lambda supports this environment variable to set additional command-line variables in Lambda. This environment variable allows you to specify the initialization of tools, specifically the launching of native or Java programming language agents using the `agentlib` or `javaagent` options.
- **NODE_OPTIONS** – On Node.js 10x and above, Lambda supports this environment variable.
- **DOTNET_STARTUP_HOOKS** – On .NET Core 3.1 and above, this environment variable specifies a path to an assembly (dll) that Lambda can use.

Using language-specific environment variables is the preferred way to set startup properties.

Example: Intercept Lambda invokes with `javaagent`

The Java virtual machine (JVM) tries to locate the class that was specified with the `javaagent` parameter to the JVM, and invoke its `premain` method before the application's entry point.

The following example uses [Byte Buddy](#), a library for creating and modifying Java classes during the runtime of a Java application without the help of a compiler. Byte Buddy offers an additional API for generating Java agents. In this example, the `Agent` class intercepts every call of the `handleRequest` method made to the [RequestStreamHandler](#) class. This class is used internally in the runtime to wrap the handler invocations.

```
import com.amazonaws.services.lambda.runtime.RequestStreamHandler;
import net.bytebuddy.agent.builder.AgentBuilder;
import net.bytebuddy.asm.Advice;
import net.bytebuddy.matcher.ElementMatchers;

import java.lang.instrument.Instrumentation;

public class Agent {

    public static void premain(String agentArgs, Instrumentation inst) {
        new AgentBuilder.Default()
            .with(new AgentBuilder.InitializationStrategy.SelfInjection.Eager())
            .type(ElementMatchers.isSubTypeOf(RequestStreamHandler.class))
            .transform((builder, typeDescription, classLoader, module) -> builder
                .method(ElementMatchers.nameContains("handleRequest"))
                .intercept(Advice.to(TimerAdvice.class)))
            .installOn(inst);
    }
}
```

The agent in the preceding example uses the `TimerAdvice` method. `TimerAdvice` measures how many milliseconds are spent with the method call and logs the method time and details, such as name and passed arguments.

```
import static net.bytebuddy.asm.Advice.AllArguments;
import static net.bytebuddy.asm.Advice.Enter;
import static net.bytebuddy.asm.Advice.OnMethodEnter;
import static net.bytebuddy.asm.Advice.OnMethodExit;
import static net.bytebuddy.asm.Advice.Origin;

public class TimerAdvice {

    @OnMethodEnter
    static long enter() {
        return System.currentTimeMillis();
    }

    @OnMethodExit
    static void exit(@Origin String method, @Enter long start, @AllArguments Object[] args)
    {
        StringBuilder sb = new StringBuilder();
        for (Object arg : args) {
            sb.append(arg);
            sb.append(", ");
        }
        System.out.println(method + " method with args: " + sb.toString() + " took " +
        (System.currentTimeMillis() - start) + " milliseconds ");
    }
}
```

The `TimerAdvice` method above has the following dependencies.

```
*'com.amazonaws'*, *name*: *'aws-lambda-java-core'*, *version*: *'1.2.1'*  
*'net.bytebuddy'*, *name*: *'byte-buddy-dep'*, *version*: *'1.10.14'*  
*'net.bytebuddy'*, *name*: *'byte-buddy-agent'*, *version*: *'1.10.14'*
```

After you create a layer that contains the agent JAR, you can pass the JAR name to the runtime's JVM by setting an environment variable.

```
JAVA_TOOL_OPTIONS=-javaagent:"/opt/ExampleAgent-0.0.jar"
```

After invoking the function with `{key=lambdaInput}`, you can find the following line in the logs:

```
public java.lang.Object lambdainternal.EventHandlerLoader
$PojoMethodRequestHandler.handleRequest
(java.lang.Object,com.amazonaws.services.lambda.runtime.Context) method with args:
{key=lambdaInput}, lambdainternal.api.LambdaContext@4d9d1b69, took 106 milliseconds
```

Example: Adding a shutdown hook to the JVM runtime process

When an extension is registered during a Shutdown event, the runtime process gets up to 500 ms to handle graceful shutdown. You can hook into the runtime process, and when the JVM begins its shutdown process, it starts all registered hooks. To register a shutdown hook, you must [register as](#)

[an extension \(p. 906\)](#). You do not need to explicitly register for the Shutdown event, as that is automatically sent to the runtime.

```
import java.lang.instrument.Instrumentation;

public class Agent {

    public static void premain(String agentArgs, Instrumentation inst) {
        // Register the extension.
        //
        //

        // Register the shutdown hook
        addShutdownHook();
    }

    private static void addShutdownHook() {
        // Shutdown hooks get up to 500 ms to handle graceful shutdown before the runtime is
        // terminated.
        //
        // You can use this time to egress any remaining telemetry, close open database
        // connections, etc.
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        // Inside the shutdown hook's thread we can perform any remaining task which
        // needs to be done.
        }));
    }
}
```

Example: Retrieving the InvokedFunctionArn

```
@OnMethodEnter
static long enter() {
    String invokedFunctionArn = null;
    for (Object arg : args) {
        if (arg instanceof Context) {
            Context context = (Context) arg;
            invokedFunctionArn = context.getInvokedFunctionArn();
        }
    }
}
```

Wrapper scripts

You can create a *wrapper script* to customize the runtime startup behavior of your Lambda function. A wrapper script enables you to set configuration parameters that cannot be set through language-specific environment variables.

Note

Invocations may fail if the wrapper script does not successfully start the runtime process.

Wrapper scripts are supported on all native [Lambda runtimes \(p. 37\)](#) which use the Amazon Linux 2 operating system. The custom provided.al2 runtime does not support wrapper scripts.

When you use a wrapper script for your function, Lambda starts the runtime using your script. Lambda sends to your script the path to the interpreter and all of the original arguments for the standard

runtime startup. Your script can extend or transform the startup behavior of the program. For example, the script can inject and alter arguments, set environment variables, or capture metrics, errors, and other diagnostic information.

You specify the script by setting the value of the AWS_LAMBDA_EXEC_WRAPPER environment variable as the file system path of an executable binary or script.

Example: Create and use a wrapper script with Python 3.8

In the following example, you create a wrapper script to start the Python interpreter with the -X importtime option. When you run the function, Lambda generates a log entry to show the duration of the import time for each import.

To create and use a wrapper script with Python 3.8

1. To create the wrapper script, paste the following code into a file named importtime_wrapper:

```
#!/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[@]}"
```

2. To give the script executable permissions, enter chmod +x importtime_wrapper from the command line.
3. Deploy the script as a [Lambda layer \(p. 93\)](#).
4. Create a function using the Lambda console.
 - a. Open the [Lambda console](#).
 - b. Choose **Create function**.
 - c. Under **Basic information**, for **Function name**, enter **wrapper-test-function**.
 - d. For **Runtime**, choose **Python 3.8**.
 - e. Choose **Create function**.
5. Add the layer to your function.
 - a. Choose your function, and then choose **Code** if it is not already selected.
 - b. Choose **Add a layer**.
 - c. Under **Choose a layer**, choose the **Name** and **Version** of the compatible layer that you created earlier.
 - d. Choose **Add**.
6. Add the code and the environment variable to your function.
 - a. In the function [code editor \(p. 21\)](#), paste the following function code:

```
import json

def lambda_handler(event, context):
```

```
# TODO implement
return {
    'statusCode': 200,
    'body': json.dumps('Hello from Lambda!')
}
```

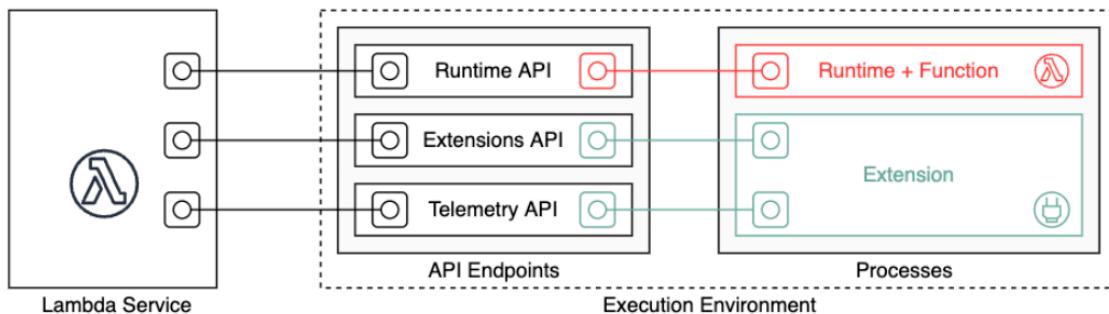
- b. Choose **Save**.
 - c. Under **Environment variables**, choose **Edit**.
 - d. Choose **Add environment variable**.
 - e. For **Key**, enter AWS_LAMBDA_EXEC_WRAPPER.
 - f. For **Value**, enter /opt/importtime_wrapper.
 - g. Choose **Save**.
7. To run the function, choose **Test**.

Because your wrapper script started the Python interpreter with the -X importtime option, the logs show the time required for each import. For example:

```
...
2020-06-30T18:48:46.780+01:00 import time: 213 | 213 | simplejson
2020-06-30T18:48:46.780+01:00 import time: 50 | 263 | simplejson.raw_json
...
```

Lambda runtime API

AWS Lambda provides an HTTP API for [custom runtimes \(p. 59\)](#) to receive invocation events from Lambda and send response data back within the Lambda [execution environment \(p. 37\)](#).



The OpenAPI specification for the runtime API version **2018-06-01** is available in [runtime-api.zip](#)

To create an API request URL, runtimes get the API endpoint from the `AWS_LAMBDA_RUNTIME_API` environment variable, add the API version, and add the desired resource path.

Example Request

```
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next"
```

API methods

- [Next invocation \(p. 54\)](#)
- [Invocation response \(p. 55\)](#)
- [Initialization error \(p. 55\)](#)
- [Invocation error \(p. 56\)](#)

Next invocation

Path – `/runtime/invocation/next`

Method – GET

The runtime sends this message to Lambda to request an invocation event. The response body contains the payload from the invocation, which is a JSON document that contains event data from the function trigger. The response headers contain additional data about the invocation.

Response headers

- `Lambda-Runtime-Aws-Request-Id` – The request ID, which identifies the request that triggered the function invocation.

For example, `8476a536-e9f4-11e8-9739-2dfe598c3fcfd`.
- `Lambda-Runtime-Deadline-Ms` – The date that the function times out in Unix time milliseconds.

For example, `1542409706888`.
- `Lambda-Runtime-Invoked-Function-Arn` – The ARN of the Lambda function, version, or alias that's specified in the invocation.

For example, `arn:aws:lambda:us-east-2:123456789012:function:custom-runtime`.

- Lambda-Runtime-Trace-Id – The [AWS X-Ray tracing header](#).

For example, `Root=1-5bef4de7-ad49b0e87f6ef6c87fc2e700;Parent=9a9197af755a6419;Sampled=1`.

- Lambda-Runtime-Client-Context – For invocations from the AWS Mobile SDK, data about the client application and device.
- Lambda-Runtime-Cognito-Identity – For invocations from the AWS Mobile SDK, data about the Amazon Cognito identity provider.

Do not set a timeout on the GET request as the response may be delayed. Between when Lambda bootstraps the runtime and when the runtime has an event to return, the runtime process may be frozen for several seconds.

The request ID tracks the invocation within Lambda. Use it to specify the invocation when you send the response.

The tracing header contains the trace ID, parent ID, and sampling decision. If the request is sampled, the request was sampled by Lambda or an upstream service. The runtime should set the `_X_AMZN_TRACE_ID` with the value of the header. The X-Ray SDK reads this to get the IDs and determine whether to trace the request.

Invocation response

Path – `/runtime/invocation/AwsRequestId/response`

Method – POST

After the function has run to completion, the runtime sends an invocation response to Lambda. For synchronous invocations, Lambda sends the response to the client.

Example success request

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
curl -X POST "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response" -d "SUCCESS"
```

Initialization error

If the function returns an error or the runtime encounters an error during initialization, the runtime uses this method to report the error to Lambda.

Path – `/runtime/init/error`

Method – POST

Headers

`Lambda-Runtime-Function-Error-Type` – Error type that the runtime encountered. Required: no.

This header consists of a string value. Lambda accepts any string, but we recommend a format of `<category.reason>`. For example:

- Runtime.NoSuchHandler
- Runtime.APIKeyNotFound

- Runtime.ConfigInvalid
- Runtime.UnknownReason

Body parameters

ErrorRequest – Information about the error. Required: no.

This field is a JSON object with the following structure:

```
{  
    errorMessage: string (text description of the error),  
    errorType: string,  
    stackTrace: array of strings  
}
```

Note that Lambda accepts any value for errorType.

The following example shows a Lambda function error message in which the function could not parse the event data provided in the invocation.

Example Function error

```
{  
    "errorMessage" : "Error parsing event data.",  
    "errorType" : "InvalidEventDataException",  
    "stackTrace": [ ]  
}
```

Response body parameters

- StatusResponse – String. Status information, sent with 202 response codes.
- ErrorResponse – Additional error information, sent with the error response codes. ErrorResponse contains an error type and an error message.

Response codes

- 202 – Accepted
- 403 – Forbidden
- 500 – Container error. Non-recoverable state. Runtime should exit promptly.

Example initialization error request

```
ERROR={"errorMessage" : "Failed to load function.", "errorType" :  
    "InvalidFunctionException"}  
curl -X POST "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/init/error" -d "$ERROR"  
--header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

Invocation error

If the function returns an error or the runtime encounters an error, the runtime uses this method to report the error to Lambda.

Path – /runtime/invocation/*AwsRequestId*/error

Method – POST

Headers

Lambda-Runtime-Function-Error-Type – Error type that the runtime encountered. Required: no.

This header consists of a string value. Lambda accepts any string, but we recommend a format of <category.reason>. For example:

- Runtime.NoSuchHandler
- Runtime.APIKeyNotFound
- Runtime.ConfigInvalid
- Runtime.UnknownReason

Body parameters

ErrorRequest – Information about the error. Required: no.

This field is a JSON object with the following structure:

```
{  
    errorMessage: string (text description of the error),  
    errorType: string,  
    stackTrace: array of strings  
}
```

Note that Lambda accepts any value for errorType.

The following example shows a Lambda function error message in which the function could not parse the event data provided in the invocation.

Example Function error

```
{  
    "errorMessage" : "Error parsing event data.",  
    "errorType" : "InvalidEventDataException",  
    "stackTrace": [ ]  
}
```

Response body parameters

- StatusResponse – String. Status information, sent with 202 response codes.
- ErrorResponse – Additional error information, sent with the error response codes. ErrorResponse contains an error type and an error message.

Response codes

- 202 – Accepted
- 400 – Bad Request
- 403 – Forbidden
- 500 – Container error. Non-recoverable state. Runtime should exit promptly.

Example error request

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
```

```
ERROR={"errorMessage": "Error parsing event data.", "errorType":  
    "\"InvalidEventDataException\""}  
curl -X POST "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/  
error" -d "$ERROR" --header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

Custom Lambda runtimes

You can implement an AWS Lambda runtime in any programming language. A runtime is a program that runs a Lambda function's handler method when the function is invoked. You can include a runtime in your function's deployment package as an executable file named `bootstrap`.

A runtime runs the function's setup code, reads the handler name from an environment variable, and reads invocation events from the Lambda runtime API. The runtime passes the event data to the function handler, and posts the response from the handler back to Lambda.

Your custom runtime runs in the standard Lambda [execution environment \(p. 37\)](#). It can be a shell script, a script in a language that's included in Amazon Linux, or a binary executable file that's compiled in Amazon Linux.

To get started with custom runtimes, see [Tutorial – Publishing a custom runtime \(p. 62\)](#). You can also explore a custom runtime implemented in C++ at [awslabs/aws-lambda-cpp](#) on GitHub.

Topics

- [Using a custom runtime \(p. 59\)](#)
- [Building a custom runtime \(p. 59\)](#)
- [Implementing response streaming in a custom runtime \(p. 60\)](#)

Using a custom runtime

To use a custom runtime, set your function's runtime to `provided.al2`. The runtime can be included in your function's deployment package, or in a [layer \(p. 93\)](#).

Example `function.zip`

```
.\n### bootstrap\n### function.sh
```

If there's a file named `bootstrap` in your deployment package, Lambda runs that file. If not, Lambda looks for a runtime in the function's layers. If the `bootstrap` file isn't found or isn't executable, your function returns an error upon invocation.

Building a custom runtime

A custom runtime's entry point is an executable file named `bootstrap`. The `bootstrap` file can be the runtime, or it can invoke another file that creates the runtime. The following example uses a bundled version of Node.js to run a JavaScript runtime in a separate file named `runtime.js`.

Example `bootstrap`

```
#!/bin/sh\ncd $LAMBDA_TASK_ROOT\n./node-v11.1.0-linux-x64/bin/node runtime.js
```

Your runtime code is responsible for completing some initialization tasks. Then it processes invocation events in a loop until it's shut down. The initialization tasks run once [per instance of the function \(p. 14\)](#) to prepare the environment to handle invocations.

Initialization tasks

- **Retrieve settings** – Read environment variables to get details about the function and environment.
 - `_HANDLER` – The location to the handler, from the function's configuration. The standard format is `file.method`, where `file` is the name of the file without an extension, and `method` is the name of a method or function that's defined in the file.
 - `LAMBDA_TASK_ROOT` – The directory that contains the function code.
 - `AWS_LAMBDA_RUNTIME_API` – The host and port of the runtime API.
- For a full list of available variables, see [Defined runtime environment variables \(p. 79\)](#).
- **Initialize the function** – Load the handler file and run any global or static code that it contains. Functions should create static resources like SDK clients and database connections once, and reuse them for multiple invocations.
- **Handle errors** – If an error occurs, call the [initialization error \(p. 55\)](#) API and exit immediately.

Initialization counts towards billed execution time and timeout. When an execution triggers the initialization of a new instance of your function, you can see the initialization time in the logs and [AWS X-Ray trace \(p. 807\)](#).

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
```

While it runs, a runtime uses the [Lambda runtime interface \(p. 54\)](#) to manage incoming events and report errors. After completing initialization tasks, the runtime processes incoming events in a loop. In your runtime code, perform the following steps in order.

Processing tasks

- **Get an event** – Call the [next invocation \(p. 54\)](#) API to get the next event. The response body contains the event data. Response headers contain the request ID and other information.
- **Propagate the tracing header** – Get the X-Ray tracing header from the `Lambda-Runtime-Trace-Id` header in the API response. Set the `_X_AMZN_TRACE_ID` environment variable locally with the same value. The X-Ray SDK uses this value to connect trace data between services.
- **Create a context object** – Create an object with context information from environment variables and headers in the API response.
- **Invoke the function handler** – Pass the event and context object to the handler.
- **Handle the response** – Call the [invocation response \(p. 55\)](#) API to post the response from the handler.
- **Handle errors** – If an error occurs, call the [invocation error \(p. 56\)](#) API.
- **Cleanup** – Release unused resources, send data to other services, or perform additional tasks before getting the next event.

You can include the runtime in your function's deployment package, or distribute the runtime separately in a function layer. For an example walkthrough, see [Tutorial – Publishing a custom runtime \(p. 62\)](#).

Implementing response streaming in a custom runtime

For response streaming functions, the `response` and `error` endpoints have slightly modified behavior that lets the runtime stream partial responses to the client and return payloads in chunks. For more information about the specific behavior, see the following:

- `/runtime/invocation/AwsRequestId/response` – Propagates the Content-Type header from the runtime to send to the client. Lambda returns the response payload in chunks via HTTP/1.1 chunked transfer encoding. The response stream can be a maximum size of 20 MiB. To stream the response to Lambda, the runtime must:
 - Set the Lambda-Runtime-Function-Response-Mode HTTP header to `streaming`.
 - Set the Transfer-Encoding header to `chunked`.
 - Write the response conforming to the HTTP/1.1 chunked transfer encoding specification.
 - Close the underlying connection after it has successfully written the response.
- `/runtime/invocation/AwsRequestId/error` – The runtime can use this endpoint to report function or runtime errors to Lambda, which also accepts the Transfer-Encoding header. This endpoint can only be called before the runtime begins sending an invocation response.
- Report midstream errors using error trailers in `/runtime/invocation/AwsRequestId/response`
 - To report errors that occur after the runtime starts writing the invocation response, the runtime can optionally attach HTTP trailing headers named `Lambda-Runtime-Function-Error-Type` and `Lambda-Runtime-Function-Error-Body`. Lambda treats this as a successful response and forwards the error metadata that the runtime provides to the client.

Note

To attach trailing headers, the runtime must set the `Trailer` header value at the beginning of the HTTP request. This is a requirement of the HTTP/1.1 chunked transfer encoding specification.

- `Lambda-Runtime-Function-Error-Type` – The error type that the runtime encountered. This header consists of a string value. Lambda accepts any string, but we recommend a format of `<category.reason>`. For example, `Runtime.APIKeyNotFound`.
- `Lambda-Runtime-Function-Error-Body` – Base64-encoded information about the error.

Tutorial – Publishing a custom runtime

In this tutorial, you create a Lambda function with a custom runtime. You start by including the runtime in the function's deployment package. Then you migrate it to a layer that you manage independently from the function. Finally, you share the runtime layer with the world by updating its resource-based permissions policy.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console \(p. 4\)](#) to create your first Lambda function.

To complete the following steps, you need the [AWS Command Line Interface \(AWS CLI\) version 2](#). Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.0.57 Python/3.7.4 Darwin/19.6.0 exe/x86_64
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#). Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

You need an IAM role to create a Lambda function. The role needs permission to send logs to CloudWatch Logs and access the AWS services that your function uses. If you don't have a role for function development, create one now.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – Lambda**.
 - **Permissions – AWSLambdaBasicExecutionRole**.
 - **Role name – lambda-role**.

The **AWSLambdaBasicExecutionRole** policy has the permissions that the function needs to write logs to CloudWatch Logs.

Create a function

Create a Lambda function with a custom runtime. This example includes two files, a runtime bootstrap file, and a function handler. Both are implemented in Bash.

The runtime loads a function script from the deployment package. It uses two variables to locate the script. LAMBDA_TASK_ROOT tells it where the package was extracted, and _HANDLER includes the name of the script.

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 -L "$HEADERS" -X GET "http://${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)

    # Run the handler function from the script
    RESPONSE=$(($(_HANDLER" | cut -d. -f2) "$EVENT_DATA"))

    # Send the response
    curl -X POST "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/response" -d "$RESPONSE"
done
```

After loading the script, the runtime processes events in a loop. It uses the runtime API to retrieve an invocation event from Lambda, passes the event to the handler, and posts the response back to Lambda. To get the request ID, the runtime saves the headers from the API response to a temporary file, and reads the Lambda-Runtime-Aws-Request-Id header from the file.

Note

Runtimes have additional responsibilities, including error handling, and providing context information to the handler. For details, see [Building a custom runtime \(p. 59\)](#).

The script defines a handler function that takes event data, logs it to stderr, and returns it.

Example function.sh

```
function handler () {
    EVENT_DATA=$1
    echo "$EVENT_DATA" 1>&2;
    RESPONSE="Echoing request: '$EVENT_DATA'

    echo $RESPONSE
}
```

Save both files in a project directory named `runtime-tutorial`.

```
runtime-tutorial
# bootstrap
# function.sh
```

Make the files executable and add them to a .zip file archive.

```
runtime-tutorial$ chmod 755 function.sh bootstrap
runtime-tutorial$ zip function.zip function.sh bootstrap
  adding: function.sh (deflated 24%)
  adding: bootstrap (deflated 39%)
```

Create a function named bash-runtime.

```
runtime-tutorial$ aws lambda create-function --function-name bash-runtime \
--zip-file fileb://function.zip --handler function.handler --runtime provided \
--role arn:aws:iam::123456789012:role/lambda-role
{
    "FunctionName": "bash-runtime",
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:bash-runtime",
    "Runtime": "provided",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Handler": "function.handler",
    "CodeSha256": "mv/xRv84LPCxdpcbKvmwuFzwo7sLwU01VxcUv3wK1M=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "RevisionId": "2e1d51b0-6144-4763-8e5c-7d5672a01713",
    ...
}
```

Invoke the function and verify the response.

```
runtime-tutorial$ aws lambda invoke --function-name bash-runtime --payload
'{"text":"Hello"}' response.txt --cli-binary-format raw-in-base64-out
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
runtime-tutorial$ cat response.txt
Echoing request: '{"text":"Hello"}'
```

Create a layer

To separate the runtime code from the function code, create a layer that only contains the runtime. Layers let you develop your function's dependencies independently, and can reduce storage usage when you use the same layer with multiple functions.

Create a layer archive that contains the bootstrap file.

```
runtime-tutorial$ zip runtime.zip bootstrap
  adding: bootstrap (deflated 39%)
```

Create a layer with the publish-layer-version command.

```
runtime-tutorial$ aws lambda publish-layer-version --layer-name bash-runtime --zip-file
fileb://runtime.zip
{
    "Content": {
        "Location": "https://awslambda-us-east-1-layers.s3.us-east-1.amazonaws.com/
snapshots/123456789012/bash-runtime-018c209b...",
        "CodeSha256": "bXVLhHi+D3H1QbDARUVPrDwlC7bssPxyS0qt1QZqusE=",
        "CodeSize": 584,
        "UncompressedCodeSize": 0
    },
}
```

```
"LayerArn": "arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime",
"LayerVersionArn": "arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:1",
"Description": "",
"CreatedDate": "2018-11-28T07:49:14.476+0000",
"Version": 1
}
```

This creates the first version of the layer.

Update the function

To use the runtime layer with the function, configure the function to use the layer, and remove the runtime code from the function.

Update the function configuration to pull in the layer.

```
runtime-tutorial$ aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:1
{
    "FunctionName": "bash-runtime",
    "Layers": [
        {
            "Arn": "arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:1",
            "CodeSize": 584,
            "UncompressedCodeSize": 679
        }
    ]
    ...
}
```

This adds the runtime to the function in the /opt directory. Lambda uses this runtime, but only if you remove it from the function's deployment package. Update the function code to only include the handler script.

```
runtime-tutorial$ zip function-only.zip function.sh
adding: function.sh (deflated 24%)
runtime-tutorial$ aws lambda update-function-code --function-name bash-runtime --zip-file
fileb://function-only.zip
{
    "FunctionName": "bash-runtime",
    "CodeSize": 270,
    "Layers": [
        {
            "Arn": "arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:7",
            "CodeSize": 584,
            "UncompressedCodeSize": 679
        }
    ]
    ...
}
```

Invoke the function to verify that it works with the runtime layer.

```
runtime-tutorial$ aws lambda invoke --function-name bash-runtime --payload
'{"text":"Hello"}' response.txt --cli-binary-format raw-in-base64-out
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
runtime-tutorial$ cat response.txt
```

```
Echoing request: '{"text":"Hello"}'
```

Update the runtime

To log information about the execution environment, update the runtime script to output environment variables.

Example bootstrap

```
#!/bin/sh

set -euo pipefail

echo "## Environment variables:"
env

# Initialization - load function handler
source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"
...
```

Create a second version of the layer with the new code.

```
runtime-tutorial$ zip runtime.zip bootstrap
updating: bootstrap (deflated 39%)
runtime-tutorial$ aws lambda publish-layer-version --layer-name bash-runtime --zip-file
fileb://runtime.zip
```

Configure the function to use the new version of the layer.

```
runtime-tutorial$ aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:2
```

Share the layer

Add a permission statement to your runtime layer to share it with other accounts.

```
runtime-tutorial$ aws lambda add-layer-version-permission --layer-name bash-runtime --
version-number 2 \
--principal "*" --statement-id publish --action lambda:GetLayerVersion
{
    "Statement": "{\"Sid\":\"publish\",\"Effect\":\"Allow\",\"Principal\":\"*\",\"Action\":
\"lambda:GetLayerVersion\", \"Resource\":\"arn:aws:lambda:us-east-1:123456789012:layer:bash-
runtime:2\"}",
    "RevisionId": "9d5fe08e-2a1e-4981-b783-37ab551247ff"
}
```

You can add multiple statements that each grant permission to a single account, accounts in an organization, or all accounts.

Clean up

Delete each version of the layer.

```
runtime-tutorial$ aws lambda delete-layer-version --layer-name bash-runtime --version-
number 1
```

```
runtime-tutorial$ aws lambda delete-layer-version --layer-name bash-runtime --version-number 2
```

Because the function holds a reference to version 2 of the layer, it still exists in Lambda. The function continues to work, but functions can no longer be configured to use the deleted version. If you then modify the list of layers on the function, you must specify a new version or omit the deleted layer.

Delete the tutorial function with the `delete-function` command.

```
runtime-tutorial$ aws lambda delete-function --function-name bash-runtime
```

Using AVX2 vectorization in Lambda

Advanced Vector Extensions 2 (AVX2) is a vectorization extension to the Intel x86 instruction set that can perform single instruction multiple data (SIMD) instructions over vectors of 256 bits. For vectorizable algorithms with [highly parallelizable](#) operation, using AVX2 can enhance CPU performance, resulting in lower latencies and higher throughput. Use the AVX2 instruction set for compute-intensive workloads such as machine learning inferencing, multimedia processing, scientific simulations, and financial modeling applications.

Note

Lambda arm64 uses NEON SIMD architecture and does not support the x86 AVX2 extensions.

To use AVX2 with your Lambda function, make sure that your function code is accessing AVX2-optimized code. For some languages, you can install the AVX2-supported version of libraries and packages. For other languages, you can recompile your code and dependencies with the appropriate compiler flags set (if the compiler supports auto-vectorization). You can also compile your code with third-party libraries that use AVX2 to optimize math operations. For example, Intel Math Kernel Library (Intel MKL), OpenBLAS (Basic Linear Algebra Subprograms), and AMD BLAS-like Library Instantiation Software (BLIS). Auto-vectorized languages, such as Java, automatically use AVX2 for computations.

You can create new Lambda workloads or move existing AVX2-enabled workloads to Lambda at no additional cost.

For more information about AVX2, see [Advanced Vector Extensions 2](#) in Wikipedia.

Compiling from source

If your Lambda function uses a C or C++ library to perform compute-intensive vectorizable operations, you can set the appropriate compiler flags and recompile the function code. Then, the compiler automatically vectorizes your code.

For the `gcc` or `clang` compiler, add `-march=haswell` to the command or set `-mavx2` as a command option.

```
~ gcc -march=haswell main.c
or
~ gcc -mavx2 main.c

~ clang -march=haswell main.c
or
~ clang -mavx2 main.c
```

To use a specific library, follow instructions in the library's documentation to compile and build the library. For example, to build TensorFlow from source, you can follow the [installation instructions](#) on the TensorFlow website. Make sure to use the `-march=haswell` compile option.

Enabling AVX2 for Intel MKL

Intel MKL is a library of optimized math operations that implicitly use AVX2 instructions when the compute platform supports them. Frameworks such as PyTorch [build with Intel MKL by default](#), so you don't need to enable AVX2.

Some libraries, such as TensorFlow, provide options in their build process to specify Intel MKL optimization. For example, with TensorFlow, use the `--config=mkl` option.

You can also build popular scientific Python libraries, such as SciPy and NumPy, with Intel MKL. For instructions on building these libraries with Intel MKL, see [Numpy/Scipy with Intel MKL and Intel Compilers](#) on the Intel website.

For more information about Intel MKL and similar libraries, see [Math Kernel Library](#) in Wikipedia, the [OpenBLAS website](#), and the [AMD BLIS repository](#) on GitHub.

AVX2 support in other languages

If you don't use C or C++ libraries and don't build with Intel MKL, you can still get some AVX2 performance improvement for your applications. Note that the actual improvement depends on the compiler or interpreter's ability to utilize the AVX2 capabilities on your code.

Python

Python users generally use SciPy and NumPy libraries for compute-intensive workloads. You can compile these libraries to enable AVX2, or you can use the Intel MKL-enabled versions of the libraries.

Node

For compute-intensive workloads, use AVX2-enabled or Intel MKL-enabled versions of the libraries that you need.

Java

Java's JIT compiler can auto-vectorize your code to run with AVX2 instructions. For information about detecting vectorized code, see the [Code vectorization in the JVM](#) presentation on the OpenJDK website.

Go

The standard Go compiler doesn't currently support auto-vectorization, but you can use [gccgo](#), the GCC compiler for Go. Set the `-mavx2` option:

```
gcc -o avx2 -mavx2 -Wall main.c
```

Intrinsics

It's possible to use [intrinsic functions](#) in many languages to manually vectorize your code to use AVX2. However, we don't recommend this approach. Manually writing vectorized code takes significant effort. Also, debugging and maintaining such code is more difficult than using code that depends on auto-vectorization.

Configuring AWS Lambda functions

Learn how to configure the core capabilities and options for your Lambda function using the Lambda API or console. These configurations apply to a function deployed as a [container image \(p. 111\)](#) and for a function deployed as a [.zip file archive \(p. 107\)](#).

[Configuring function options \(p. 71\)](#)

You can find an overview of how to configure your Lambda function using the console and AWS CLI.

[Environment variables \(p. 76\)](#)

You can make your function code portable and keep secrets out of your code by storing them in your function's configuration by using environment variables.

[Versions \(p. 83\)](#)

By publishing a version of your function, you can store your code and configuration as a separate resource that cannot be changed.

[Aliases \(p. 85\)](#)

You can configure your clients to invoke a specific Lambda function version by using an alias, instead of updating the client.

[Private networking \(p. 89\)](#)

You can use Amazon VPC to create a private network for resources such as databases, cache instances, or internal services.

[Creating layers \(p. 93\)](#)

You create a layer manage your function's dependencies independently and keep your development package small.

[Response streaming \(p. 99\)](#)

You can configure your Lambda function URLs to stream response payloads back to clients.

Response streaming can benefit latency sensitive applications by improving time to first byte (TTFB) performance. This is because you can send partial responses back to the client as they become available. Additionally, you can use response streaming to build functions that return larger payloads.

Configuring Lambda function options

After you create a function, you can configure additional capabilities for the function, such as triggers, network access, and file system access. You can also adjust resources associated with the function, such as memory and concurrency. These configurations apply to functions defined as .zip file archives and to functions defined as container images.

You can also create and edit test events to test your function using the console.

For function configuration best practices, see [Function configuration \(p. 812\)](#).

Sections

- [Function versions \(p. 71\)](#)
- [Using the function overview \(p. 71\)](#)
- [Configuring functions \(console\) \(p. 72\)](#)
- [Configuring functions \(API\) \(p. 73\)](#)
- [Configuring function memory \(console\) \(p. 73\)](#)
- [Configuring function timeout \(console\) \(p. 73\)](#)
- [Configuring ephemeral storage \(console\) \(p. 74\)](#)
- [Accepting function memory recommendations \(console\) \(p. 74\)](#)
- [Configuring triggers \(console\) \(p. 74\)](#)
- [Testing functions \(console\) \(p. 75\)](#)

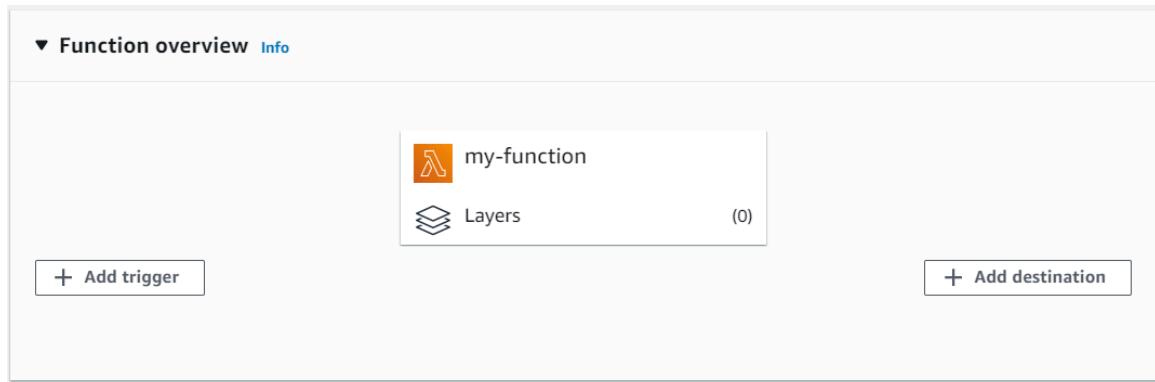
Function versions

A function has an unpublished version, and can have published versions and aliases. By default, the console displays configuration information for the unpublished version of the function. You change the unpublished version when you update your function's code and configuration.

A published version is a snapshot of your function code and configuration that can't be changed (except for a few configuration items relevant to a function version, such as provisioned concurrency).

Using the function overview

The **Function overview** shows a visualization of your function and its upstream and downstream resources. You can use it to jump to trigger and destination configuration. You can use it to jump to layer configuration for functions defined as .zip file archives.



Configuring functions (console)

For the following function configurations, you can change the settings only for the unpublished version of a function. In the console, the function **Configuration** tab provides the following sections:

- **General configuration** – Configure [memory \(p. 73\)](#) or opt in to the [AWS Compute Optimizer \(p. 74\)](#). You can also configure function [timeout \(p. 73\)](#) and the execution role.
- **Permissions** – Configure the execution role and other [permissions \(p. 815\)](#).
- **Environment variables** – Key-value pairs that Lambda sets in the execution environment. To extend your function's configuration outside of code, [use environment variables \(p. 76\)](#).
- **Tags** – Key-value pairs that Lambda attaches to your function resource. [Use tags \(p. 250\)](#) to organize Lambda functions into groups for cost reporting and filtering in the Lambda console.

Tags apply to the entire function, including all versions and aliases.
- **Virtual private cloud (VPC)** – If your function needs network access to resources that are not available over the internet, [configure it to connect to a virtual private cloud \(VPC\) \(p. 222\)](#).
- **Monitoring and operations tools** – configure CloudWatch and other [monitoring tools \(p. 860\)](#).
- **Concurrency** – [Reserve concurrency for a function \(p. 210\)](#) to set the maximum number of simultaneous executions for a function. Provision concurrency to ensure that a function can scale without fluctuations in latency. Reserved concurrency applies to the entire function, including all versions and aliases.
- **Function URL** – Configure a [function URL \(p. 166\)](#) to add a unique HTTP(S) endpoint to your Lambda function. You can configure a function URL on the \$LATEST unpublished function version, or on any function alias.

You can configure the following options on a function, a function version, or an alias.

- **Triggers** – Configure [triggers \(p. 74\)](#).
- **Destinations** – Configure [destinations \(p. 125\)](#) for asynchronous invocations.
- **Asynchronous invocation** – [Configure error handling behavior \(p. 123\)](#) to reduce the number of retries that Lambda attempts, or the amount of time that unprocessed events stay queued before Lambda discards them. [Configure a dead-letter queue \(p. 129\)](#) to retain discarded events.
- **Code signing** – To use [Code signing \(p. 241\)](#) with your function, configure the function to include a code-signing configuration.
- **Database proxies** – [Create a database proxy \(p. 232\)](#) for functions that use an Amazon RDS DB instance or cluster.
- **File systems** – Connect your function to a [file system \(p. 236\)](#).
- **State machines** – Use a state machine to orchestrate and apply error handling to your function.

The console provides separate tabs to configure aliases and versions:

- **Aliases** – An alias is a named resource that maps to a function version. You can change an alias to map to a different function version.
- **Versions** – Lambda assigns a new version number each time you publish your function. For more information about managing versions, see [Lambda function versions \(p. 83\)](#).

You can configure the following items for a published function version:

- Triggers
- Destinations
- Provisioned concurrency

- Asynchronous invocation
- Database proxies

Configuring functions (API)

To configure functions with the Lambda API, use the following actions:

- [UpdateFunctionCode \(p. 1367\)](#) – Update the function's code.
- [UpdateFunctionConfiguration \(p. 1377\)](#) – Update version-specific settings.
- [TagResource \(p. 1345\)](#) – Tag a function.
- [AddPermission \(p. 1141\)](#) – Modify the [resource-based policy \(p. 832\)](#) of a function, version, or alias.
- [PutFunctionConcurrency \(p. 1327\)](#) – Configure a function's reserved concurrency.
- [PublishVersion \(p. 1315\)](#) – Create an immutable version with the current code and configuration.
- [CreateAlias \(p. 1146\)](#) – Create aliases for function versions.
- [PutFunctionEventInvokeConfig](#) – Configure error handling for asynchronous invocation.
- [CreateFunctionUrlConfig](#) – Create a function URL configuration.
- [UpdateFunctionUrlConfig](#) – Update an existing function URL configuration.

Configuring function memory (console)

Lambda allocates CPU power in proportion to the amount of memory configured. *Memory* is the amount of memory available to your Lambda function at runtime. You can increase or decrease the memory and CPU power allocated to your function using the **Memory (MB)** setting. To configure the memory for your function, set a value between 128 MB and 10,240 MB in 1-MB increments. At 1,769 MB, a function has the equivalent of one vCPU (one vCPU-second of credits per second).

You can configure the memory of your function in the Lambda console.

To update the memory of a function

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. On the function configuration page, on the **General configuration** pane, choose **Edit**.
4. For **Memory (MB)**, set a value from 128 MB to 10,240 MB.
5. Choose **Save**.

Configuring function timeout (console)

Lambda runs your code for a set amount of time before timing out. *Timeout* is the maximum amount of time in seconds that a Lambda function can run. The default value for this setting is 3 seconds, but you can adjust this in increments of 1 second up to a maximum value of 15 minutes.

You can configure the timeout of your function in the Lambda console.

To change the timeout of a function

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. On the function configuration page, on the **General configuration** pane, choose **Edit**.

4. For **Timeout**, set a value from 1 second to 15 minutes.
5. Choose **Save**.

Configuring ephemeral storage (console)

By default, Lambda allocates 512 MB for a function's /tmp directory. You can increase or decrease this amount using the **Ephemeral storage (MB)** setting. To configure the size of a function's /tmp directory, set a whole number value between 512 MB and 10,240 MB, in 1-MB increments.

Note

Configuring ephemeral storage past the default 512 MB allocated incurs a cost. For more information, see [Lambda pricing](#).

You can configure the size of a function's /tmp directory in the Lambda console.

To update the size of a function's /tmp directory

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. On the function configuration page, on the **General configuration** pane, choose **Edit**.
4. For **Ephemeral storage (MB)**, set a value from 512 MB to 10,240 MB.
5. Choose **Save**.

Accepting function memory recommendations (console)

If you have administrator permissions in AWS Identity and Access Management (IAM), you can opt in to receive Lambda function memory setting recommendations from AWS Compute Optimizer. For instructions on opting in to memory recommendations for your account or organization, see [Opting in your account](#) in the *AWS Compute Optimizer User Guide*.

Note

Compute Optimizer supports only functions that use x86_64 architecture.

When you've opted in and your [Lambda function meets Compute Optimizer requirements](#), you can view and accept function memory recommendations from Compute Optimizer in the Lambda console.

To accept a function memory recommendation

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. On the function configuration page, on the **General configuration** pane, choose **Edit**.
4. Under **Memory (MB)**, in the memory alert, choose **Update**.
5. Choose **Save**.

Configuring triggers (console)

You can configure other AWS services to trigger your function each time a specified event occurs.

For details about how services trigger Lambda functions, see [Using AWS Lambda with other services \(p. 556\)](#).

To add a trigger to your function.

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to update.
3. Under **Function overview**, choose **Add trigger**.
4. From the drop-down list of triggers, choose a trigger. The console displays additional configuration fields required for this trigger.
5. Choose **Add**.

Testing functions (console)

You can create test events for your function from the **Test** tab. For more information, see [Testing Lambda functions in the console](#).

You can also invoke your function without saving your test event by choosing **Test** before saving. This creates an unsaved test event that Lambda will preserve for the duration of the session. You can access your unsaved test events from either the **Test** or **Code** tab.

Using AWS Lambda environment variables

You can use environment variables to adjust your function's behavior without updating code. An environment variable is a pair of strings that is stored in a function's version-specific configuration. The Lambda runtime makes environment variables available to your code and sets additional environment variables that contain information about the function and invocation request.

Note

To increase database security, we recommend that you use AWS Secrets Manager instead of environment variables to store database credentials. For more information, see [Configuring database access for a Lambda function](#).

Environment variables are not evaluated prior to the function invocation. Any value you define is considered a literal string and not expanded. Perform the variable evaluation in your function code.

Sections

- [Configuring environment variables \(p. 76\)](#)
- [Configuring environment variables with the API \(p. 77\)](#)
- [Example scenario for environment variables \(p. 77\)](#)
- [Retrieve environment variables \(p. 78\)](#)
- [Defined runtime environment variables \(p. 79\)](#)
- [Securing environment variables \(p. 80\)](#)
- [Sample code and templates \(p. 82\)](#)

Configuring environment variables

You define environment variables on the unpublished version of your function. When you publish a version, the environment variables are locked for that version along with other [version-specific configuration \(p. 71\)](#).

You create an environment variable for your function by defining a key and a value. Your function uses the name of the key to retrieve the value of environment variable.

To set environment variables in the Lambda console

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration**, then choose **Environment variables**.
4. Under **Environment variables**, choose **Edit**.
5. Choose **Add environment variable**.
6. Enter a key and value.

Requirements

- Keys start with a letter and are at least two characters.
- Keys only contain letters, numbers, and the underscore character (_).
- Keys aren't [reserved by Lambda \(p. 79\)](#).
- The total size of all environment variables doesn't exceed 4 KB.

7. Choose **Save**.

Configuring environment variables with the API

To manage environment variables with the AWS CLI or AWS SDK, use the following API operations.

- [UpdateFunctionConfiguration \(p. 1377\)](#)
- [GetFunctionConfiguration \(p. 1229\)](#)
- [CreateFunction \(p. 1165\)](#)

The following example sets two environment variables on a function named `my-function`.

```
aws lambda update-function-configuration --function-name my-function \
--environment "Variables={BUCKET=my-bucket,KEY=file.txt}"
```

When you apply environment variables with the `update-function-configuration` command, the entire contents of the `Variables` structure is replaced. To retain existing environment variables when you add a new one, include all existing values in your request.

To get the current configuration, use the `get-function-configuration` command.

```
aws lambda get-function-configuration --function-name my-function
```

You should see the following output:

```
{
  "FunctionName": "my-function",
  "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
  "Runtime": "nodejs18.x",
  "Role": "arn:aws:iam::123456789012:role/lambda-role",
  "Environment": {
    "Variables": {
      "BUCKET": "my-bucket",
      "KEY": "file.txt"
    }
  },
  "RevisionId": "0894d3c1-2a3d-4d48-bf7f-abade99f3c15",
  ...
}
```

To ensure that the values don't change between when you read the configuration and when you update it, you can pass the revision ID from the output of `get-function-configuration` as a parameter to `update-function-configuration`.

To configure a function's encryption key, set the `KMSKeyARN` option.

```
aws lambda update-function-configuration --function-name my-function \
--kms-key-arn arn:aws:kms:us-east-2:123456789012:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

Example scenario for environment variables

You can use environment variables to customize function behavior in your test environment and production environment. For example, you can create two functions with the same code but different configurations. One function connects to a test database, and the other connects to a production database. In this situation, you use environment variables to tell the function the hostname and other connection details for the database.

The following example shows how to define the database host and database name as environment variables.

ENVIRONMENT	DEVELOPMENT	Remove
databaseHost	lambdadb	Remove
databaseName	rd1owwlydynnm5.cuovuayfg087	Remove
Key	Value	Remove

If you want your test environment to generate more debug information than the production environment, you could set an environment variable to configure your test environment to use more verbose logging or more detailed tracing.

Retrieve environment variables

To retrieve environment variables in your function code, use the standard method for your programming language.

Node.js

```
let region = process.env.AWS_REGION
```

Python

```
import os
region = os.environ['AWS_REGION']
```

Note

In some cases, you may need to use the following format:

```
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 stores environment variables securely by encrypting them at rest. You can [configure Lambda to use a different encryption key \(p. 80\)](#), encrypt environment variable values on the client side, or set environment variables in an AWS CloudFormation template with AWS Secrets Manager.

Defined runtime environment variables

Lambda [runtimes \(p. 37\)](#) set several environment variables during initialization. Most of the environment variables provide information about the function or runtime. The keys for these environment variables are *reserved* and cannot be set in your function configuration.

Reserved environment variables

- `_HANDLER` – The handler location configured on the function.
- `_X_AMZN_TRACE_ID` – The [X-Ray tracing header \(p. 807\)](#). This environment variable is not defined for custom runtimes (for example, runtimes that use the provided or provided.al2 identifiers). You can set `_X_AMZN_TRACE_ID` for custom runtimes using the Lambda-Runtime-Trace-Id response header from the [Next invocation \(p. 54\)](#).
- `AWS_DEFAULT_REGION` – The default AWS Region where the Lambda function is executed.
- `AWS_REGION` – The AWS Region where the Lambda function is executed. If defined, this value overrides the `AWS_DEFAULT_REGION`.
- `AWS_EXECUTION_ENV` – The [runtime identifier \(p. 37\)](#), prefixed by `AWS_Lambda_` (for example, `AWS_Lambda_java8`). This environment variable is not defined for custom runtimes (for example, runtimes that use the provided or provided.al2 identifiers).
- `AWS_LAMBDA_FUNCTION_NAME` – The name of the function.
- `AWS_LAMBDA_FUNCTION_MEMORY_SIZE` – The amount of memory available to the function in MB.
- `AWS_LAMBDA_FUNCTION_VERSION` – The version of the function being executed.
- `AWS_LAMBDA_INITIALIZATION_TYPE` – The initialization type of the function, which is on-demand, provisioned-concurrency, or snap-start. For information, see [Configuring provisioned concurrency \(p. 213\)](#) or [Improving startup performance with Lambda SnapStart \(p. 992\)](#).
- `AWS_LAMBDA_LOG_GROUP_NAME`, `AWS_LAMBDA_LOG_STREAM_NAME` – The name of the Amazon CloudWatch Logs group and stream for the function. The `AWS_LAMBDA_LOG_GROUP_NAME` and `AWS_LAMBDA_LOG_STREAM_NAME` [environment variables \(p. 79\)](#) are not available in Lambda SnapStart functions.
- `AWS_ACCESS_KEY`, `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_SESSION_TOKEN` – The access keys obtained from the function's [execution role \(p. 816\)](#).
- `AWS_LAMBDA_RUNTIME_API` – ([Custom runtime \(p. 59\)](#)) The host and port of the [runtime API \(p. 54\)](#).
- `LAMBDA_TASK_ROOT` – The path to your Lambda function code.
- `LAMBDA_RUNTIME_DIR` – The path to runtime libraries.

The following additional environment variables aren't reserved and can be extended in your function configuration.

Unreserved environment variables

- `LANG` – The locale of the runtime (`en_US.UTF-8`).
- `PATH` – The execution path (`/usr/local/bin:/usr/bin/:/bin:/opt/bin`).

- LD_LIBRARY_PATH – The system library path (/lib64:/usr/lib64:\$LAMBDA_RUNTIME_DIR:\$LAMBDA_RUNTIME_DIR/lib:\$LAMBDA_TASK_ROOT:\$LAMBDA_TASK_ROOT/lib:/opt/lib).
- NODE_PATH – ([Node.js \(p. 254\)](#)) The Node.js library path (/opt/nodejs/node12/node_modules:/opt/nodejs/node_modules:\$LAMBDA_RUNTIME_DIR/node_modules).
- PYTHONPATH – ([Python 2.7, 3.6, 3.8 \(p. 318\)](#)) The Python library path (\$LAMBDA_RUNTIME_DIR).
- GEM_PATH – ([Ruby \(p. 361\)](#)) The Ruby library path (\$LAMBDA_TASK_ROOT/vendor/bundle/ruby/2.5.0:/opt/ruby/gems/2.5.0).
- AWS_XRAY_CONTEXT_MISSING – For X-Ray tracing, Lambda sets this to LOG_ERROR to avoid throwing runtime errors from the X-Ray SDK.
- AWS_XRAY_DAEMON_ADDRESS – For X-Ray tracing, the IP address and port of the X-Ray daemon.
- AWS_LAMBDA_DOTNET_PREJIT – For the .NET 3.1 runtime, set this variable to enable or disable .NET 3.1 specific runtime optimizations. Values include always, never, and provisioned-concurrency. For information, see [Configuring provisioned concurrency \(p. 213\)](#).
- TZ – The environment's time zone (UTC). The execution environment uses NTP to synchronize the system clock.

The sample values shown reflect the latest runtimes. The presence of specific variables or their values can vary on earlier runtimes.

Securing environment variables

For securing your environment variables, you can use server-side encryption to protect your data at rest and client-side encryption to protect your data in transit.

Note

To increase database security, we recommend that you use AWS Secrets Manager instead of environment variables to store database credentials. For more information, see [Configuring database access for a Lambda function](#).

Security at rest

Lambda always provides server-side encryption at rest with an AWS KMS key. By default, Lambda uses an AWS managed key. If this default behavior suits your workflow, you don't need to set anything else up. Lambda creates the AWS managed key in your account and manages permissions to it for you. AWS doesn't charge you to use this key.

If you prefer, you can provide an AWS KMS customer managed key instead. You might do this to have control over rotation of the KMS key or to meet the requirements of your organization for managing KMS keys. When you use a customer managed key, only users in your account with access to the KMS key can view or manage environment variables on the function.

Customer managed keys incur standard AWS KMS charges. For more information, see [AWS Key Management Service pricing](#).

Security in transit

For additional security, you can enable helpers for encryption in transit, which ensures that your environment variables are encrypted client-side for protection in transit.

To configure encryption for your environment variables

1. Use the AWS Key Management Service (AWS KMS) to create any customer managed keys for Lambda to use for server-side and client-side encryption. For more information, see [Creating keys](#) in the [AWS Key Management Service Developer Guide](#).

2. Using the Lambda console, navigate to the **Edit environment variables** page.
 - a. Open the [Functions page](#) of the Lambda console.
 - b. Choose a function.
 - c. Choose **Configuration**, then choose **Environment variables** from the left navigation bar.
 - d. In the **Environment variables** section, choose **Edit**.
 - e. Expand **Encryption configuration**.
3. Optionally, enable console encryption helpers to use client-side encryption to protect your data in transit.
 - a. Under **Encryption in transit**, choose **Enable helpers for encryption in transit**.
 - b. For each environment variable that you want to enable console encryption helpers for, choose **Encrypt** next to the environment variable.
 - c. Under AWS KMS key to encrypt in transit, choose a customer managed key that you created at the beginning of this procedure.
 - d. Choose **Execution role policy** and copy the policy. This policy grants permission to your function's execution role to decrypt the environment variables.

Save this policy to use in the last step of this procedure.

 - e. Add code to your function that decrypts the environment variables. Choose **Decrypt secrets snippet** to see an example.
4. Optionally, specify your customer managed key for encryption at rest.
 - a. Choose **Use a customer master key**.
 - b. Choose a customer managed key that you created at the beginning of this procedure.
5. Choose **Save**.
6. Set up permissions.

If you're using a customer managed key with server-side encryption, grant permissions to any users or roles that you want to be able to view or manage environment variables on the function. For more information, see [Managing permissions to your server-side encryption KMS key \(p. 81\)](#).

If you're enabling client-side encryption for security in transit, your function needs permission to call the `kms:Decrypt` API operation. Add the policy that you saved previously in this procedure to the function's [execution role \(p. 816\)](#).

Managing permissions to your server-side encryption KMS key

No AWS KMS permissions are required for your user or the function's execution role to use the default encryption key. To use a customer managed key, you need permission to use the key. Lambda uses your permissions to create a grant on the key. This allows Lambda to use it for encryption.

- `kms>ListAliases` – To view keys in the Lambda console.
- `kms>CreateGrant, kms:Encrypt` – To configure a customer managed key on a function.
- `kms:Decrypt` – To view and manage environment variables that are encrypted with a customer managed key.

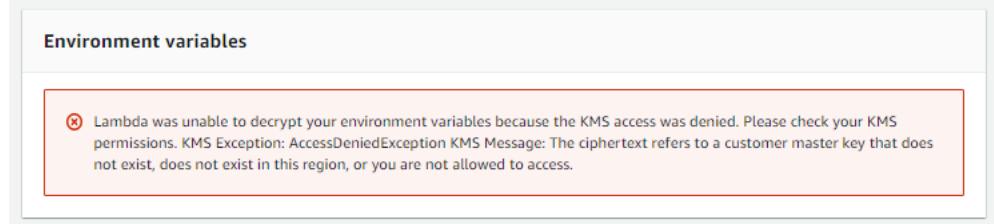
You can get these permissions from your AWS account or from a key's resource-based permissions policy. `ListAliases` is provided by the [managed policies for Lambda \(p. 823\)](#). Key policies grant the remaining permissions to users in the **Key users** group.

Users without `Decrypt` permissions can still manage functions, but they can't view environment variables or manage them in the Lambda console. To prevent a user from viewing environment variables,

add a statement to the user's permissions that denies access to the default key, a customer managed key, or all keys.

Example IAM policy – Deny access by key ARN

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "VisualEditor0",  
            "Effect": "Deny",  
            "Action": [  
                "kms:Decrypt"  
            ],  
            "Resource": "arn:aws:kms:us-east-2:123456789012:key/3be10e2d-xmpl-4be4-  
bc9d-0405a71945cc"  
        }  
    ]  
}
```



For details on managing key permissions, see [Using key policies in AWS KMS](#) in the AWS Key Management Service Developer Guide.

Sample code and templates

Sample applications in this guide's GitHub repository demonstrate the use of environment variables in function code and AWS CloudFormation templates.

Sample applications

- [Blank function \(p. 1010\)](#) – Create a basic function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests, and the AWS SDK.
- [RDS MySQL](#) – Create a VPC and an Amazon RDS DB instance in one template, with a password stored in Secrets Manager. In the application template, import database details from the VPC stack, read the password from Secrets Manager, and pass all connection configuration to the function in environment variables.

Lambda function versions

You can use versions to manage the deployment of your functions. For example, you can publish a new version of a function for beta testing without affecting users of the stable production version. Lambda creates a new version of your function each time that you publish the function. The new version is a copy of the unpublished version of the function.

Note

Lambda doesn't create a new version if the code in the unpublished version is the same as the previous published version. You need to deploy code changes in \$LATEST before you can create a new version.

A function version includes the following information:

- The function code and all associated dependencies.
- The Lambda runtime identifier and runtime version used by the function.
- All the function settings, including the environment variables.
- A unique Amazon Resource Name (ARN) to identify the specific version of the function.

When using runtime management controls with **Auto** mode, the runtime version used by the function version is updated automatically. When using **Function update** or **Manual** mode, the runtime version is not updated. For more information, see [the section called "Runtime updates" \(p. 41\)](#).

Sections

- [Creating function versions \(p. 83\)](#)
- [Using versions \(p. 84\)](#)
- [Granting permissions \(p. 84\)](#)

Creating function versions

You can change the function code and settings only on the unpublished version of a function. When you publish a version, Lambda locks the code and most of the settings to maintain a consistent experience for users of that version. For more information about configuring function settings, see [Configuring Lambda function options \(p. 71\)](#).

You can create a function version using the Lambda console.

To create a new function version

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function and then choose **Versions**.
3. On the versions configuration page, choose **Publish new version**.
4. (Optional) Enter a version description.
5. Choose **Publish**.

Alternatively, you can publish a version of a function using the [PublishVersion \(p. 1315\)](#) API operation.

The following AWS CLI command publishes a new version of a function. The response returns configuration information about the new version, including the version number and the function ARN with the version suffix.

```
aws lambda publish-version --function-name my-function
```

You should see the following output:

```
{  
  "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": "nodejs18.x",  
  ...  
}
```

Note

Lambda assigns monotonically increasing sequence numbers for versioning. Lambda never reuses version numbers, even after you delete and recreate a function.

Using versions

You can reference your Lambda function using either a qualified ARN or an unqualified ARN.

- **Qualified ARN** – The function ARN with a version suffix. The following example refers to version 42 of the helloworld function.

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:42
```

- **Unqualified ARN** – The function ARN without a version suffix.

```
arn:aws:lambda:aws-region:acct-id:function:helloworld
```

You can use a qualified or an unqualified ARN in all relevant API operations. However, you can't use an unqualified ARN to create an alias.

If you decide not to publish function versions, you can invoke the function using either the qualified or unqualified ARN in your [event source mapping \(p. 131\)](#). When you invoke a function using an unqualified ARN, Lambda implicitly invokes \$LATEST.

Lambda publishes a new function version only if the code has never been published or if the code has changed from the last published version. If there is no change, the function version remains at the last published version.

The qualified ARN for each Lambda function version is unique. After you publish a version, you can't change the ARN or the function code.

Granting permissions

You can use a [resource-based policy \(p. 832\)](#) or an [identity-based policy \(p. 823\)](#) to grant access to your function. The scope of the permission depends on whether you apply the policy to a function or to one version of a function. For more information about function resource names in policies, see [Resources and conditions for Lambda actions \(p. 838\)](#).

You can simplify the management of event sources and AWS Identity and Access Management (IAM) policies by using function aliases. For more information, see [Lambda function aliases \(p. 85\)](#).

Lambda function aliases

You can create one or more aliases for your Lambda function. A Lambda alias is like a pointer to a specific function version. Users can access the function version using the alias Amazon Resource Name (ARN).

Sections

- [Creating a function alias \(Console\) \(p. 85\)](#)
- [Managing aliases with the Lambda API \(p. 85\)](#)
- [Managing aliases with AWS SAM and AWS CloudFormation \(p. 86\)](#)
- [Using aliases \(p. 86\)](#)
- [Resource policies \(p. 86\)](#)
- [Alias routing configuration \(p. 86\)](#)

Creating a function alias (Console)

You can create a function alias using the Lambda console.

To create an alias

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Aliases** and then choose **Create alias**.
4. On the **Create alias** page, do the following:
 - a. Enter a **Name** for the alias.
 - b. (Optional) Enter a **Description** for the alias.
 - c. For **Version**, choose a function version that you want the alias to point to.
 - d. (Optional) To configure routing on the alias, expand **Weighted alias**. For more information, see [Alias routing configuration \(p. 86\)](#).
 - e. Choose **Save**.

Managing aliases with the Lambda API

To create an alias using the AWS Command Line Interface (AWS CLI), use the [create-alias](#) command.

```
aws lambda create-alias --function-name my-function --name alias-name --function-version version-number --description " "
```

To change an alias to point a new version of the function, use the [update-alias](#) command.

```
aws lambda update-alias --function-name my-function --name alias-name --function-version version-number
```

To delete an alias, use the [delete-alias](#) command.

```
aws lambda delete-alias --function-name my-function --name alias-name
```

The AWS CLI commands in the preceding steps correspond to the following Lambda API operations:

- [CreateAlias \(p. 1146\)](#)
- [UpdateAlias \(p. 1349\)](#)
- [DeleteAlias \(p. 1182\)](#)

Managing aliases with AWS SAM and AWS CloudFormation

You can create and manage function aliases using the AWS Serverless Application Model (AWS SAM) and AWS CloudFormation.

To see how to declare a function alias in an AWS SAM template, refer to the [AWS::Serverless::Function](#) page in the AWS SAM Developer Guide. For information on creating and configuring aliases using AWS CloudFormation, see [AWS::Lambda::Alias](#) in the AWS CloudFormation User Guide.

Using aliases

Each alias has a unique ARN. An alias can point only to a function version, not to another alias. You can update an alias to point to a new version of the function.

Event sources such as Amazon Simple Storage Service (Amazon S3) invoke your Lambda function. These event sources maintain a mapping that identifies the function to invoke when events occur. If you specify a Lambda function alias in the mapping configuration, you don't need to update the mapping when the function version changes. For more information, see [Lambda event source mappings \(p. 131\)](#).

In a resource policy, you can grant permissions for event sources to use your Lambda function. If you specify an alias ARN in the policy, you don't need to update the policy when the function version changes.

Resource policies

You can use a [resource-based policy \(p. 832\)](#) to give a service, resource, or account access to your function. The scope of that permission depends on whether you apply it to an alias, a version, or the entire function. For example, if you use an alias name (such as helloworld:PROD), the permission allows you to invoke the helloworld function using the alias ARN (helloworld:PROD).

If you attempt to invoke the function without an alias or a specific version, then you get a permission error. This permission error still occurs even if you attempt to directly invoke the function version associated with the alias.

For example, the following AWS CLI command grants Amazon S3 permissions to invoke the PROD alias of the helloworld function when Amazon S3 is acting on behalf of examplebucket.

```
aws lambda add-permission --function-name helloworld \
--qualifier PROD --statement-id 1 --principal s3.amazonaws.com --action
lambda:InvokeFunction \
--source-arn arn:aws:s3:::examplebucket --source-account 123456789012
```

For more information about using resource names in policies, see [Resources and conditions for Lambda actions \(p. 838\)](#).

Alias routing configuration

Use routing configuration on an alias to send a portion of traffic to a second function version. For example, you can reduce the risk of deploying a new version by configuring the alias to send most of the traffic to the existing version, and only a small percentage of traffic to the new version.

Note that Lambda uses a simple probabilistic model to distribute the traffic between the two function versions. At low traffic levels, you might see a high variance between the configured and actual percentage of traffic on each version. If your function uses provisioned concurrency, you can avoid [spillover invocations \(p. 871\)](#) by configuring a higher number of provisioned concurrency instances during the time that alias routing is active.

You can point an alias to a maximum of two Lambda function versions. The versions must meet the following criteria:

- Both versions must have the same [execution role \(p. 816\)](#).
- Both versions must have the same [dead-letter queue \(p. 129\)](#) configuration, or no dead-letter queue configuration.
- Both versions must be published. The alias cannot point to \$LATEST.

To configure routing on an alias

Note

Verify that the function has at least two published versions. To create additional versions, follow the instructions in [Lambda function versions \(p. 83\)](#).

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Aliases** and then choose **Create alias**.
4. On the **Create alias** page, do the following:
 - a. Enter a **Name** for the alias.
 - b. (Optional) Enter a **Description** for the alias.
 - c. For **Version**, choose the first function version that you want the alias to point to.
 - d. Expand **Weighted alias**.
 - e. For **Additional version**, choose the second function version that you want the alias to point to.
 - f. For **Weight (%)**, enter a weight value for the function. *Weight* is the percentage of traffic that is assigned to that version when the alias is invoked. The first version receives the residual weight. For example, if you specify 10 percent to **Additional version**, the first version is assigned 90 percent automatically.
 - g. Choose **Save**.

Configuring alias routing using CLI

Use the `create-alias` and `update-alias` AWS CLI commands to configure the traffic weights between two function versions. When you create or update the alias, you specify the traffic weight in the `routing-config` parameter.

The following example creates a Lambda function alias named **routing-alias** that points to version 1 of the function. Version 2 of the function receives 3 percent of the traffic. The remaining 97 percent of traffic is routed to version 1.

```
aws lambda create-alias --name routing-alias --function-name my-function --function-version
1 \
--routing-config AdditionalVersionWeights={"2":0.03}
```

Use the `update-alias` command to increase the percentage of incoming traffic to version 2. In the following example, you increase the traffic to 5 percent.

```
aws lambda update-alias --name routing-alias --function-name my-function \
```

```
--routing-config AdditionalVersionWeights={"2":0.05}
```

To route all traffic to version 2, use the update-alias command to change the function-version property to point the alias to version 2. The command also resets the routing configuration.

```
aws lambda update-alias --name routing-alias --function-name my-function \
--function-version 2 --routing-config AdditionalVersionWeights={}
```

The AWS CLI commands in the preceding steps correspond to the following Lambda API operations:

- [CreateAlias \(p. 1146\)](#)
- [UpdateAlias \(p. 1349\)](#)

Determining which version has been invoked

When you configure traffic weights between two function versions, there are two ways to determine the Lambda function version that has been invoked:

- **CloudWatch Logs** – Lambda automatically emits a START log entry that contains the invoked version ID to Amazon CloudWatch Logs for every function invocation. The following is an example:

```
19:44:37 START RequestId: request id Version: $version
```

For alias invocations, Lambda uses the Executed Version dimension to filter the metric data by the invoked version. For more information, see [Working with Lambda function metrics \(p. 870\)](#).

- **Response payload (synchronous invocations)** – Responses to synchronous function invocations include an x-amz-executed-version header to indicate which function version has been invoked.

VPC networking for Lambda

Amazon Virtual Private Cloud (Amazon VPC) is a virtual network in the AWS cloud, dedicated to your AWS account. You can use Amazon VPC to create a private network for resources such as databases, cache instances, or internal services. For more information about Amazon VPC, see [What is Amazon VPC?](#)

A Lambda function always runs inside a VPC owned by the Lambda service. Lambda applies network access and security rules to this VPC and Lambda maintains and monitors the VPC automatically. If your Lambda function needs to access the resources in your account VPC, [configure the function to access the VPC \(p. 222\)](#). Lambda provides managed resources named Hyperplane ENIs, which your Lambda function uses to connect from the Lambda VPC to an ENI (Elastic network interface) in your account VPC.

There's no additional charge for using a VPC or a Hyperplane ENI. There are charges for some VPC components, such as NAT gateways. For more information, see [Amazon VPC Pricing](#).

Topics

- [VPC network elements \(p. 89\)](#)
- [Connecting Lambda functions to your VPC \(p. 90\)](#)
- [Lambda Hyperplane ENIs \(p. 90\)](#)
- [Connections \(p. 91\)](#)
- [Security \(p. 91\)](#)
- [Observability \(p. 92\)](#)

VPC network elements

Amazon VPC networks includes the following network elements:

- Elastic network interface – [elastic network interface](#) is a logical networking component in a VPC that represents a virtual network card.
- Subnet – A range of IP addresses in your VPC. You can add AWS resources to a specified subnet. Use a public subnet for resources that must connect to the internet, and a private subnet for resources that don't connect to the internet.
- Security group – use security groups to control access to the AWS resources in each subnet.
- Access control list (ACL) – use a network ACL to provide additional security in a subnet. The default subnet ACL allows all inbound and outbound traffic.
- Route table – contains a set of routes that AWS uses to direct the network traffic for your VPC. You can explicitly associate a subnet with a particular route table. By default, the subnet is associated with the main route table.
- Route – each route in a route table specifies a range of IP addresses and the destination where Lambda sends the traffic for that range. The route also specifies a target, which is the gateway, network interface, or connection through which to send the traffic.
- NAT gateway – An AWS Network Address Translation (NAT) service that controls access from a private VPC private subnet to the Internet.
- VPC endpoints – You can use an Amazon VPC endpoint to create private connectivity to services hosted in AWS, without requiring access over the internet or through a NAT device, VPN connection, or AWS Direct Connect connection. For more information, see [AWS PrivateLink and VPC endpoints](#).

For more information about Amazon VPC networking definitions, see [How Amazon VPC works](#) in the Amazon VPC Developer Guide and the [Amazon VPC FAQs](#).

Connecting Lambda functions to your VPC

A Lambda function always runs inside a VPC owned by the Lambda service. By default, a Lambda function isn't connected to VPCs in your account. When you connect a function to a VPC in your account, the function can't access the internet unless your VPC provides access.

Lambda accesses resources in your VPC using a Hyperplane ENI. Hyperplane ENIs provide NAT capabilities from the Lambda VPC to your account VPC using VPC-to-VPC NAT (V2N). V2N provides connectivity from the Lambda VPC to your account VPC, but not in the other direction.

When you create a Lambda function (or update its VPC settings), Lambda allocates a Hyperplane ENI for each subnet in your function's VPC configuration. Multiple Lambda functions can share a network interface, if the functions share the same subnet and security group.

To connect to another AWS service, you can use [VPC endpoints](#) for private communications between your VPC and supported AWS services. An alternative approach is to use a [NAT gateway](#) to route outbound traffic to another AWS service.

To give your function access to the internet, route outbound traffic to a NAT gateway in a public subnet. The NAT gateway has a public IP address and can connect to the internet through the VPC's internet gateway.

For information about how to configure Lambda VPC networking, see [Lambda networking \(p. 222\)](#).

Lambda Hyperplane ENIs

The Hyperplane ENI is a managed network resource that the Lambda service creates and manages. Multiple execution environments in the Lambda VPC can use a Hyperplane ENI to securely access resources inside of VPCs in your account. Hyperplane ENIs provide NAT capabilities from the Lambda VPC to your account VPC. For more information about Hyperplane ENIs, see [Improved VPC networking for AWS Lambda functions](#) in the AWS compute blog.

Each unique security group and subnet combination in your account requires a different network interface. Functions in the account that share the same security group and subnet combination use the same network interfaces.

Because the functions in your account share the ENI resources, the ENI lifecycle is more complex than other Lambda resources. The following sections describe the ENI lifecycle.

ENI lifecycle

- [Creating ENIs \(p. 90\)](#)
- [Managing ENIs \(p. 91\)](#)
- [Deleting ENIs \(p. 91\)](#)

Creating ENIs

Lambda may create Hyperplane ENI resources for a newly created VPC-enabled function or for a VPC configuration change to an existing function. The function remains in pending state while Lambda creates the required resources. When the Hyperplane ENI is ready, the function transitions to active state and the ENI becomes available for use. Lambda can require several minutes to create a Hyperplane ENI.

For a newly created VPC-enabled function, any invocations or other API actions that operate on the function fail until the function state transitions to active.

For a VPC configuration change to an existing function, any function invocations continue to use the Hyperplane ENI associated with the old subnet and security group configuration until the function state transitions to active.

If a Lambda function remains idle for consecutive weeks, Lambda reclaims the unused Hyperplane ENIs and sets the function state to idle. The next invocation causes Lambda to reactivate the idle function. The invocation fails, and the function enters pending state until Lambda completes the creation or allocation of a Hyperplane ENI.

For more information about function states, see [Lambda function states \(p. 159\)](#).

Managing ENIs

Lambda uses permissions in your function's execution role to create and manage network interfaces. Lambda creates a Hyperplane ENI when you define a unique subnet plus security group combination for a VPC-enabled function in an account. Lambda reuses the Hyperplane ENI for other VPC-enabled functions in your account that use the same subnet and security group combination.

There is no quota on the number of Lambda functions that can use the same Hyperplane ENI. However, each Hyperplane ENI supports up to 65,000 connections/ports. If the number of connections exceeds 65,000, Lambda creates a new Hyperplane ENI to provide additional connections.

When you update your function configuration to access a different VPC, Lambda terminates connectivity to the Hyperplane ENI in the previous VPC. The process to update the connectivity to a new VPC can take several minutes. During this time, invocations to the function continue to use the previous VPC. After the update is complete, new invocations start using the Hyperplane ENI in the new VPC. At this point, the Lambda function is no longer connected to the previous VPC.

Deleting ENIs

When you update a function to remove its VPC configuration, Lambda requires up to 20 minutes to delete the attached Hyperplane ENI. Lambda only deletes the ENI if no other function (or published function version) is using that Hyperplane ENI.

Lambda relies on permissions in the function [execution role \(p. 816\)](#) to delete the Hyperplane ENI. If you delete the execution role before Lambda deletes the Hyperplane ENI, Lambda won't be able to delete the Hyperplane ENI. You can manually perform the deletion.

Lambda doesn't delete network interfaces that are in use by functions or function versions in your account. You can use the [Lambda ENI Finder](#) to identify the functions or function versions that are using a Hyperplane ENI. For any functions or function versions that you no longer need, you can remove the VPC configuration so that Lambda deletes the Hyperplane ENI.

Connections

Lambda supports two types of connections: TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).

When you create a VPC, Lambda automatically creates a set of DHCP options and associates them with the VPC. You can configure your own DHCP options set for your VPC. For more details, refer to [Amazon VPC DHCP options](#).

Amazon provides a DNS server (the Amazon Route 53 resolver) for your VPC. For more information, see [DNS support for your VPC](#).

Security

AWS provides [security groups](#) and [network ACLs](#) to increase security in your VPC. Security groups control inbound and outbound traffic for your instances, and network ACLs control inbound and outbound traffic for your subnets. Security groups provide enough access control for most subnets. You can use network ACLs if you want an additional layer of security for your VPC. For more information, see [Internetwork](#)

[traffic privacy in Amazon VPC](#). Every subnet that you create is automatically associated with the VPC's default network ACL. You can change the association, and you can change the contents of the default network ACL.

For general security best practices, see [VPC security best practices](#). For details on how you can use IAM to manage access to the Lambda API and resources, see [AWS Lambda permissions \(p. 815\)](#).

You can use Lambda-specific condition keys for VPC settings to provide additional permission controls for your Lambda functions. For more information about VPC condition keys, see [Using IAM condition keys for VPC settings \(p. 225\)](#).

Observability

You can use [VPC Flow Logs](#) to capture information about the IP traffic going to and from network interfaces in your VPC. You can publish Flow log data to Amazon CloudWatch Logs or Amazon S3. After you've created a flow log, you can retrieve and view its data in the chosen destination.

Note: when you attach a function to a VPC, the CloudWatch log messages do not use the VPC routes. Lambda sends them using the regular routing for logs.

Creating and sharing Lambda layers

Lambda [layers \(p. 11\)](#) provide a convenient way to package libraries and other dependencies that you can use with your Lambda functions. Using layers reduces the size of uploaded deployment archives and makes it faster to deploy your code.

A layer is a .zip file archive that can contain additional code or data. A layer can contain libraries, a [custom runtime \(p. 59\)](#), data, or configuration files. Layers promote code sharing and separation of responsibilities so that you can iterate faster on writing business logic.

You can use layers only with Lambda functions [deployed as a .zip file archive \(p. 18\)](#). For functions [defined as a container image \(p. 881\)](#), you package your preferred runtime and all code dependencies when you create the container image. For more information, see [Working with Lambda layers and extensions in container images](#) on the AWS Compute Blog.

You can create layers using the Lambda console, the Lambda API, AWS CloudFormation, or the AWS Serverless Application Model (AWS SAM). For more information about creating layers with AWS SAM, see [Working with layers](#) in the *AWS Serverless Application Model Developer Guide*.

Note

For Node.js runtimes 16 and earlier, Lambda doesn't support ES module dependencies in layers. Lambda does support ES module dependencies for Node.js 18.

Sections

- [Creating layer content \(p. 93\)](#)
- [Compiling the .zip file archive for your layer \(p. 93\)](#)
- [Including library dependencies in a layer \(p. 94\)](#)
- [Language-specific instructions \(p. 95\)](#)
- [Creating a layer \(p. 95\)](#)
- [Deleting a layer version \(p. 97\)](#)
- [Configuring layer permissions \(p. 97\)](#)
- [Using AWS CloudFormation with layers \(p. 97\)](#)

Creating layer content

When you create a layer, you must bundle all its content into a .zip file archive. You upload the .zip file archive to your layer from Amazon Simple Storage Service (Amazon S3) or your local machine. Lambda extracts the layer contents into the /opt directory when setting up the execution environment for the function.

Compiling the .zip file archive for your layer

You build your layer code into a .zip file archive using the same procedure that you would use for a function deployment package. If your layer includes any native code libraries, you must compile and build these libraries using a Linux development machine so that the binaries are compatible with [Amazon Linux \(p. 37\)](#).

When you create a layer, you can specify whether the layer is compatible with one or both of the instruction set architectures. You may need to set specific compile flags to build a layer that is compatible with the arm64 architecture.

One way to ensure that you package libraries correctly for Lambda is to use [AWS Cloud9](#). For more information, see [Using Lambda layers to simplify your development process](#) on the AWS Compute Blog.

Including library dependencies in a layer

For each [Lambda runtime \(p. 37\)](#), the PATH variable includes specific folders in the /opt directory. If you define the same folder structure in your layer .zip file archive, your function code can access the layer content without the need to specify the path.

The following table lists the folder paths that each runtime supports.

Layer paths for each Lambda runtime

Runtime	Path
Node.js	nodejs/node_modules
	nodejs/node14/node_modules (NODE_PATH)
	nodejs/node16/node_modules (NODE_PATH)
	nodejs/node18/node_modules (NODE_PATH)
Python	python
	python/lib/python3.10/site-packages(site directories)
Java	java/lib (CLASSPATH)
Ruby	ruby/gems/2.7.0 (GEM_PATH)
	ruby/lib (RUBYLIB)
All runtimes	bin (PATH)
	lib (LD_LIBRARY_PATH)

The following examples show how you can structure the folders in your layer .zip archive.

Node.js

Example file structure for the AWS X-Ray SDK for Node.js

```
xray-sdk.zip  
# nodejs/node_modules/aws-xray-sdk
```

Python

Example file structure for the Pillow library

```
pillow.zip  
# python/PIL  
# python/Pillow-5.3.0.dist-info
```

Ruby

Example file structure for the JSON gem

```
json.zip  
# ruby/gems/2.7.0/
```

```
| build_info  
| cache  
| doc  
| extensions  
| gems  
| # json-2.1.0  
# specifications  
# json-2.1.0.gemspec
```

Java

Example file structure for the Jackson JAR file

```
jackson.zip  
# java/lib/jackson-core-2.2.3.jar
```

All

Example file structure for the jq library

```
jq.zip  
# bin/jq
```

For more information about path settings in the Lambda execution environment, see [Defined runtime environment variables \(p. 79\)](#).

Language-specific instructions

For language-specific instructions on how to create a .zip file archive, see the following topics.

Node.js

[Deploy Node.js Lambda functions with .zip file archives \(p. 263\)](#)

Python

[Deploy Python Lambda functions with .zip file archives \(p. 324\)](#)

Ruby

[Deploy Ruby Lambda functions with .zip file archives \(p. 365\)](#)

Java

[Deploy Java Lambda functions with .zip or JAR file archives \(p. 393\)](#)

Go

[Deploy Go Lambda functions with .zip file archives \(p. 460\)](#)

C#

[Deploy C# Lambda functions with .zip file archives \(p. 497\)](#)

PowerShell

[Deploy PowerShell Lambda functions with .zip file archives \(p. 530\)](#)

Creating a layer

You can create new layers using the Lambda console or the Lambda API.

Layers can have one or more version. When you create a layer, Lambda sets the layer version to version 1. You can configure permissions on an existing layer version, but to update the code or make other configuration changes, you must create a new version of the layer.

To create a layer (console)

1. Open the [Layers page](#) of the Lambda console.
2. Choose **Create layer**.
3. Under **Layer configuration**, for **Name**, enter a name for your layer.
4. (Optional) For **Description**, enter a description for your layer.
5. To upload your layer code, do one of the following:
 - To upload a .zip file from your computer, choose **Upload a .zip file**. Then, choose **Upload** to select your local .zip file.
 - To upload a file from Amazon S3, choose **Upload a file from Amazon S3**. Then, for **Amazon S3 link URL**, enter a link to the file.
6. (Optional) For **Compatible architectures**, choose one value or both values.
7. (Optional) For **Compatible runtimes**, choose up to 15 runtimes.
8. (Optional) For **License**, enter any necessary license information.
9. Choose **Create**.

To create a layer (API)

To create a layer, use the **publish-layer-version** command with a name, description, .zip file archive, a list of [runtimes \(p. 37\)](#) and a list of architectures that are compatible with the layer. The runtimes and architecture parameters are optional.

```
aws lambda publish-layer-version --layer-name my-layer \
--description "My layer" \
--license-info "MIT" \
--zip-file file://layer.zip \
--compatible-runtimes python3.7 python3.8 \
--compatible-architectures "arm64" "x86_64"
```

You should see output similar to the following:

```
{  
    "Content": {  
        "Location": "https://awslambda-us-east-2-layers.s3.us-east-2.amazonaws.com/  
snapshots/123456789012/my-layer-4aaa2fbb-ff77-4b0a-ad92-5b78a716a96a?  
versionId=27iWyA73cCAYqyH...",  
        "CodeSha256": "tv9jJ0+rPbXUUXuRKi7CwHzKtLDkDRJLB3cC3Z/ouXo=",  
        "CodeSize": 169  
    },  
    "LayerArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer",  
    "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer:1",  
    "Description": "My layer",  
    "CreatedDate": "2018-11-14T23:03:52.894+0000",  
    "Version": 1,  
    "CompatibleArchitectures": [  
        "arm64",  
        "x86_64"  
    ],  
    "LicenseInfo": "MIT",  
    "CompatibleRuntimes": [  
        "python3.7",  
        "python3.8"  
    ]  
}
```

}

Note

Each time that you call `publish-layer-version`, you create a new version of the layer.

Deleting a layer version

To delete a layer version, use the `delete-layer-version` command.

```
aws lambda delete-layer-version --layer-name my-layer --version-number 1
```

When you delete a layer version, you can no longer configure a Lambda function to use it. However, any function that already uses the version continues to have access to it. Version numbers are never reused for a layer name.

Configuring layer permissions

By default, a layer that you create is private to your AWS account. However, you can optionally share the layer with other accounts or make it public.

To grant layer-usage permission to another account, add a statement to the layer version's permissions policy using the `add-layer-version-permission` command. In each statement, you can grant permission to a single account, all accounts, or an organization.

```
aws lambda add-layer-version-permission --layer-name xray-sdk-nodejs --statement-id xaccount \
--action lambda:GetLayerVersion --principal 111122223333 --version-number 1 --output text
```

You should see output similar to the following:

```
e210ffdc-e901-43b0-824b-5fc0dd26d16 {"Sid":"xaccount","Effect":"Allow","Principal":{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:lambda:layer-east-2:123456789012:layer:xray-sdk-nodejs:1"}
```

Permissions apply only to a single layer version. Repeat the process each time that you create a new layer version.

For more examples, see [Granting layer access to other accounts \(p. 836\)](#).

Using AWS CloudFormation with layers

You can use AWS CloudFormation to create a layer and associate the layer with your Lambda function. The following example template creates a layer named `blank-nodejs-lib` and attaches the layer to the Lambda function using the `Layers` property.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: A Lambda application that calls the Lambda API.
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs14.x
      Layers:
        - !Ref blank-nodejs-lib
```

```
CodeUri: function/.  
Description: Call the Lambda API  
Timeout: 10  
# Function's execution role  
Policies:  
  - AWSLambdaBasicExecutionRole  
  - AWSLambda_ReadOnlyAccess  
  - AWSXrayWriteOnlyAccess  
Tracing: Active  
Layers:  
  - !Ref libs  
libs:  
  Type: AWS::Lambda::LayerVersion  
  Properties:  
    LayerName: blank-nodejs-lib  
    Description: Dependencies for the blank sample app.  
    Content:  
      S3Bucket: my-bucket-region-123456789012  
      S3Key: layer.zip  
    CompatibleRuntimes:  
      - nodejs14.x
```

Configuring a Lambda function to stream responses

You can configure your Lambda function URLs to stream response payloads back to clients. Response streaming can benefit latency sensitive applications by improving time to first byte (TTFB) performance. This is because you can send partial responses back to the client as they become available. Additionally, you can use response streaming to build functions that return larger payloads. Response stream payloads have a soft limit of 20 MB as compared to the 6 MB limit for buffered responses. There is a bandwidth limit of 16777216 bps for streaming functions.

Streaming responses incurs a cost. For more information, see [AWS Lambda Pricing](#).

Currently, Lambda supports response streaming only on Node.js 14.x, Node.js 16.x, and Node.js 18.x managed runtimes. You can also use a custom runtime with a custom Runtime API integration to stream responses. Responses may only be streamed through either a Function URL or through the AWS SDKs.

Note

When testing your function through the Lambda console, you'll always see responses as buffered.

Writing response streaming-enabled functions

Writing the handler for response streaming functions is different than typical handler patterns. When writing streaming functions, be sure to do the following:

- Wrap your function with the `awslambda.streamifyResponse()` decorator that the native Node.js runtimes provide.
- End the stream gracefully to ensure that all data processing is complete.

Configuring a handler function to stream responses

To indicate to the runtime that Lambda should stream your function's responses, you must wrap your function with the `streamifyResponse()` decorator. This tells the runtime to use the proper logic path for streaming responses and enables the function to stream responses.

The `streamifyResponse()` decorator accepts a function that accepts the following parameters:

- `event` – Provides information about the function URL's invocation event, such as the HTTP method, query parameters, and the request body.
- `responseStream` – Provides a writable stream.
- `context` – Provides methods and properties with information about the invocation, function, and execution environment.

The `responseStream` object is a [Node.js writableStream](#). As with any such stream, you should use the `pipeline()` method.

Example response streaming-enabled handler

```
const pipeline = require("util").promisify(require("stream").pipeline);
const { Readable } = require('stream');

exports.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);
});
```

While `responseStream` offers the `write()` method to write to the stream, we recommend that you use [pipeline\(\)](#) wherever possible. Using `pipeline()` ensures that the writable stream is not overwhelmed by a faster readable stream.

Ending the stream

Make sure that you properly end the stream before the handler returns. The `pipeline()` method handles this automatically.

For other use cases, call the `responseStream.end()` method to properly end a stream. This method signals that no more data should be written to the stream. This method isn't required if you write to the stream with `pipeline()` or `pipe()`.

Example Example ending a stream with pipeline()

```
const pipeline = require("util").promisify(require("stream").pipeline);

exports.handler = awslambda.streamifyResponse(async (event, responseStream, _context) => {
    await pipeline(requestStream, responseStream);
});
```

Example Example ending a stream without pipeline()

```
exports.handler = awslambda.streamifyResponse(async (event, responseStream, _context) => {
    responseStream.write("Hello ");
    responseStream.write("world ");
    responseStream.write("from ");
    responseStream.write("Lambda!");
    responseStream.end();
});
```

Invoking a response streaming enabled function using Lambda function URLs

Note

You must invoke your function using a function URL to stream the responses.

You can invoke response streaming enabled functions by changing the invoke mode of your function's URL. The invoke mode determines which API operation Lambda uses to invoke your function. The available invoke modes are:

- **BUFFERED** – This is the default option. Lambda invokes your function using the `Invoke` API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.
- **RESPONSE_STREAM** – Enables your function to stream payload results as they become available. Lambda invokes your function using the `InvokeWithResponseStream` API operation. The maximum response payload size is 20 MB. However, you can [request a quota increase](#).

You can still invoke your function without response streaming by directly calling the `Invoke` API operation. However, Lambda streams all response payloads for invocations that come through the function's URL until you change the invoke mode to **BUFFERED**.

To set the invoke mode of a function URL (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of the function that you want to set the invoke mode for.
3. Choose the **Configuration** tab, and then choose **Function URL**.
4. Choose **Edit**, then choose **Additional settings**.
5. Under **Invoke mode**, choose your desired invoke mode.
6. Choose **Save**.

To set the invoke mode of a function's URL (AWS CLI)

```
aws lambda update-function-url-config --function-name my-function --invoke-mode  
RESPONSE_STREAM
```

To set the invoke mode of a function's URL (AWS CloudFormation)

```
MyFunctionUrl:  
  Type: AWS::Lambda::Url  
  Properties:  
    AuthType: AWS_IAM  
    InvokeMode: RESPONSE_STREAM
```

For more information about configuring function URLs, see [Lambda function URLs \(p. 166\)](#).

Tutorial: Creating a response streaming Lambda function with a function URL

In this tutorial, you create a Lambda function defined as a .zip file archive with a public function URL endpoint that returns a response stream. For more information about configuring function URLs, see [Creating and managing function URLs \(p. 167\)](#).

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console \(p. 4\)](#) to create your first Lambda function.

To complete the following steps, you need the [AWS Command Line Interface \(AWS CLI\) version 2](#). Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.0.57 Python/3.7.4 Darwin/19.6.0 exe/x86_64
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#). Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

Create an execution role

Create the [execution role \(p. 816\)](#) that gives your Lambda function permission to access AWS resources.

To create an execution role

1. Open the [Roles page](#) of the AWS Identity and Access Management (IAM) console.
2. Choose **Create role**.
3. Create a role with the following properties:
 - **Trusted entity type – AWS service**
 - **Use case – Lambda**
 - **Permissions – AWSLambdaBasicExecutionRole**
 - **Role name – response-streaming-role**

The **AWSLambdaBasicExecutionRole** policy has the permissions that the function needs to write logs to Amazon CloudWatch Logs.

Create a response streaming function (AWS CLI)

Create a response streaming Lambda function with a function URL endpoint using the AWS Command Line Interface (AWS CLI).

To create a function that can stream responses

1. Copy the following code example into a file named `index.js`.

```
import util from 'util';
import stream from 'stream';
const { Readable } = stream;
const pipeline = util.promisify(stream.pipeline);

/* global awslambda */
export const handler = awslambda.streamifyResponse(async (event, responseStream,
  _context) => {
  const requestStream = Readable.from(Buffer.from(JSON.stringify(event)));
  await pipeline(requestStream, responseStream);
});
```

2. Create a deployment package.

```
zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command.

```
aws lambda create-function \
  --function-name my-streaming-function \
  --runtime nodejs14.x \
  --zip-file fileb://function.zip \
  --handler index.handler \
```

```
--role arn:aws:iam::123456789012:role/response-streaming-role
```

To create a function URL

1. Add a resource-based policy to your function to allow public access to your function URL.

```
aws lambda add-permission  
  --function-name my-streaming-function \  
  --action lambda:InvokeFunctionUrl \  
  --statement-id 12345 \  
  --principal "*" \  
  --function-url-auth-type AWS_IAM \  
  --statement-id url
```

2. Create a URL endpoint for the function with the `create-function-url-config` command.

```
aws lambda create-function-url-config \  
  --function-name my-streaming-function \  
  --auth-type AWS_IAM \  
  --invoke-mode RESPONSE_STREAM
```

Create a response streaming function (AWS CloudFormation)

You can also create a streaming function configured with a function URL using AWS CloudFormation.

```
Resources:  
  MyStreamingFunction:  
    Type: AWS::Lambda::Function  
    Properties:  
      Handler: index.handler  
      Runtime: nodejs14.x  
      Role: arn:aws:iam::123456789012:role/response-streaming-role  
      Code:  
        ZipFile: |  
          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 3 \n");  
              await new Promise(r => setTimeout(r, 1000));  
              responseStream.end();  
            }  
          );  
        }  
      
```

```
        await responseStream.finished();
    }
);
Description: Create a streaming function configured with a URL.
MyStreamingFunctionPermissions:
Type: AWS::Lambda::Permission
Properties:
  FunctionName: !Ref MyStreamingFunction
  Action: lambda:InvokeFunctionUrl
  Principal: "*"
  FunctionUrlAuthType: AWS_IAM
MyFunctionUrl:
Type: AWS::Lambda::Url
Properties:
  TargetFunctionArn: !Ref MyUrlFunction
  AuthType: AWS_IAM
  InvokeMode: RESPONSE_STREAM
```

Create a response streaming function (AWS SAM)

You can also create a streaming function configured with a function URL using AWS Serverless Application Model (AWS SAM).

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  StreamingFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: "<path-to-your-code>"
      Handler: index.handler
      Runtime: nodejs14.x
      AutoPublishAlias: live
  MyFunctionUrl:
    Type: AWS::Lambda::Url
    Properties:
      TargetFunctionArn: !Ref StreamingFunction
      AuthType: AWS_IAM
      InvokeMode: RESPONSE_STREAM
```

Test the function URL endpoint

Test your integration by invoking your function. You can open your function's URL in a browser, or you can use curl.

```
curl --request GET "<your function url here>" --user "<key:token>" --aws-sigv4 "aws:amz:eu-west-1:lambda" --no-buffer
```

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.

3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

Deploying Lambda functions

You can deploy code to your Lambda function by uploading a zip file archive, or by creating and uploading a container image.

.zip file archives

A .zip file archive includes your application code and its dependencies. When you author functions using the Lambda console or a toolkit, Lambda automatically creates a .zip file archive of your code.

When you create functions with the Lambda API, command line tools, or the AWS SDKs, you must create a deployment package. You also must create a deployment package if your function uses a compiled language, or to add dependencies to your function. To deploy your function's code, you upload the deployment package from Amazon Simple Storage Service (Amazon S3) or your local machine.

You can upload a .zip file as your deployment package using the Lambda console, AWS Command Line Interface (AWS CLI), or to an Amazon Simple Storage Service (Amazon S3) bucket.

Container images

You can package your code and dependencies as a container image using tools such as the Docker command line interface (CLI). You can then upload the image to your container registry hosted on Amazon Elastic Container Registry (Amazon ECR).

AWS provides a set of open-source base images that you can use to build the container image for your function code. You can also use alternative base images from other container registries. AWS also provides an open-source runtime client that you add to your alternative base image to make it compatible with the Lambda service.

Additionally, AWS provides a runtime interface emulator for you to test your functions locally using tools such as the Docker CLI.

Note

You create each container image to be compatible with one of the instruction set architectures that Lambda supports. Lambda provides base images for each of the instruction set architectures and Lambda also provides base images that support both architectures.

The image that you build for your function must target only one of the architectures.

There is no additional charge for packaging and deploying functions as container images. When a function deployed as a container image is invoked, you pay for invocation requests and execution duration. You do incur charges related to storing your container images in Amazon ECR. For more information, see [Amazon ECR pricing](#).

Topics

- [Deploying Lambda functions as .zip file archives \(p. 107\)](#)
- [Deploying Lambda functions as container images \(p. 111\)](#)

Deploying Lambda functions as .zip file archives

When you create a Lambda function, you package your function code into a deployment package. Lambda supports two types of deployment packages: [container images \(p. 18\)](#) and [.zip file archives \(p. 18\)](#). The workflow to create a function depends on the deployment package type. To configure a function defined as a container image, see [the section called "Container images" \(p. 111\)](#).

You can use the Lambda console and the Lambda API to create a function defined with a .zip file archive. You can also upload an updated .zip file to change the function code.

Note

You cannot convert an existing [container image function \(p. 111\)](#) to use a .zip file archive. You must create a new function.

Topics

- [Creating the function \(p. 107\)](#)
- [Using the console code editor \(p. 108\)](#)
- [Updating function code \(p. 108\)](#)
- [Changing the runtime \(p. 109\)](#)
- [Changing the architecture \(p. 109\)](#)
- [Using the Lambda API \(p. 73\)](#)
- [AWS CloudFormation \(p. 110\)](#)

Creating the function

When you create a function defined with a .zip file archive, you choose a code template, the language version, and the execution role for the function. You add your function code after Lambda creates the function.

To create the function

1. Open the [Functions page](#) of the Lambda console.
2. Choose **Create function**.
3. Choose **Author from scratch** or **Use a blueprint** to create your function.
4. Under **Basic information**, do the following:
 - a. For **Function name**, enter the function name. Function names are limited to 64 characters in length.
 - b. For **Runtime**, choose the language version to use for your function.
 - c. (Optional) For **Architecture**, choose the instruction set architecture to use for your function. The default architecture is x86_64. When you build the deployment package for your function, make sure that it is compatible with this [instruction set architecture \(p. 29\)](#).
5. (Optional) Under **Permissions**, expand **Change default execution role**. You can create a new **Execution role** or use an existing role.
6. (Optional) Expand **Advanced settings**. You can choose a **Code signing configuration** for the function. You can also configure an (Amazon VPC) for the function to access.
7. Choose **Create function**.

Lambda creates the new function. You can now use the console to add the function code and configure other function parameters and features. For code deployment instructions, see the handler page for the runtime your function uses.

Node.js

[Deploy Node.js Lambda functions with .zip file archives \(p. 263\)](#)

Python

[Deploy Python Lambda functions with .zip file archives \(p. 324\)](#)

Ruby

[Deploy Ruby Lambda functions with .zip file archives \(p. 365\)](#)

Java

[Deploy Java Lambda functions with .zip or JAR file archives \(p. 393\)](#)

Go

[Deploy Go Lambda functions with .zip file archives \(p. 460\)](#)

C#

[Deploy C# Lambda functions with .zip file archives \(p. 497\)](#)

PowerShell

[Deploy PowerShell Lambda functions with .zip file archives \(p. 530\)](#)

Using the console code editor

The console creates a Lambda function with a single source file. For scripting languages, you can edit this file and add more files using the built-in [code editor \(p. 21\)](#). To save your changes, choose **Save**. Then, to run your code, choose **Test**.

Note

The Lambda console uses AWS Cloud9 to provide an integrated development environment in the browser. You can also use AWS Cloud9 to develop Lambda functions in your own environment. For more information, see [Working with Lambda Functions](#) in the AWS Cloud9 user guide.

When you save your function code, the Lambda console creates a .zip file archive deployment package. When you develop your function code outside of the console (using an IDE) you need to [create a deployment package \(p. 263\)](#) to upload your code to the Lambda function.

Updating function code

For scripting languages (Node.js, Python, and Ruby), you can edit your function code in the embedded code [editor \(p. 21\)](#). If the code is larger than 3MB, or if you need to add libraries, or for languages that the editor doesn't support (Java, Go, C#), you must upload your function code as a .zip archive. If the .zip file archive is smaller than 50 MB, you can upload the .zip file archive from your local machine. If the file is larger than 50 MB, upload the file to the function from an Amazon S3 bucket.

To upload function code as a .zip archive

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to update and choose the **Code** tab.
3. Under **Code source**, choose **Upload from**.
4. Choose **.zip file**, and then choose **Upload**.
 - In the file chooser, select the new image version, choose **Open**, and then choose **Save**.

5. (Alternative to step 4) Choose **Amazon S3 location**.
 - In the text box, enter the S3 link URL of the .zip file archive, then choose **Save**.

Changing the runtime

If you update the function configuration to use a new runtime, you may need to update the function code to be compatible with the new runtime. If you update the function configuration to use a different runtime, you **must** provide new function code that is compatible with the runtime and architecture. For instructions on how to create a deployment package for the function code, see the handler page for the runtime that the function uses.

To change the runtime

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to update and choose the **Code** tab.
3. Scroll down to the **Runtime settings** section, which is under the code editor.
4. Choose **Edit**.
 - a. For **Runtime**, select the runtime identifier.
 - b. For **Handler**, specify file name and handler for your function.
 - c. For **Architecture**, choose the instruction set architecture to use for your function.
5. Choose **Save**.

Changing the architecture

Before you can change the instruction set architecture, you need to ensure that your function's code is compatible with the target architecture.

If you use Node.js, Python, or Ruby and you edit your function code in the embedded [editor \(p. 21\)](#), the existing code may run without modification.

However, if you provide your function code using a .zip file archive deployment package, you must prepare a new .zip file archive that is compiled and built correctly for the target runtime and instruction-set architecture. For instructions, see the handler page for your function runtime.

To change the instruction set architecture

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to update and choose the **Code** tab.
3. Under **Runtime settings**, choose **Edit**.
4. For **Architecture**, choose the instruction set architecture to use for your function.
5. Choose **Save**.

Using the Lambda API

To create and configure a function that uses a .zip file archive, use the following API operations:

- [CreateFunction \(p. 1165\)](#)
- [UpdateFunctionCode \(p. 1367\)](#)
- [UpdateFunctionConfiguration \(p. 1377\)](#)

AWS CloudFormation

You can use AWS CloudFormation to create a Lambda function that uses a .zip file archive. In your AWS CloudFormation template, the `AWS::Lambda::Function` resource specifies the Lambda function. For descriptions of the properties in the `AWS::Lambda::Function` resource, see [AWS::Lambda::Function](#) in the *AWS CloudFormation User Guide*.

In the `AWS::Lambda::Function` resource, set the following properties to create a function defined as a .zip file archive:

- `AWS::Lambda::Function`
- `PackageType` – Set to `Zip`.
- `Code` – Enter the Amazon S3 bucket name and .zip file name in the `S3Bucket` and `S3Key` fields. For Node.js or Python, you can provide inline source code of your Lambda function.
- `Runtime` – Set the runtime value.
- `Architecture` – Set the architecture value to `arm64` to use the AWS Graviton2 processor. By default, the architecture value is `x86_64`.

Deploying Lambda functions as container images

When you create a Lambda function, you package your function code into a deployment package. Lambda supports two types of deployment packages: [container images \(p. 18\)](#) and [.zip file archives \(p. 18\)](#). The workflow to create a function is different depending on the deployment package type. To configure a function defined as a .zip file archive, see [the section called ".zip file archives" \(p. 107\)](#).

You can use the Lambda console and the Lambda API to create a function defined as a container image, update and test the image code, and configure other function settings.

Note

You cannot convert an existing [.zip file archive function \(p. 107\)](#) to use a container image. You must create a new function.

When you select an image using an image tag, Lambda translates the tag to the underlying image digest. To retrieve the digest for your image, use the [GetFunctionConfiguration \(p. 1229\)](#) API operation. To update the function to a newer image version, you must use the Lambda console to [update the function code \(p. 115\)](#), or use the [UpdateFunctionCode \(p. 1367\)](#) API operation. Configuration operations such as [UpdateFunctionConfiguration \(p. 1377\)](#) do not update the function's container image.

Topics

- [Prerequisites \(p. 111\)](#)
- [Permissions \(p. 112\)](#)
- [Creating the function \(p. 113\)](#)
- [Testing the function \(p. 114\)](#)
- [Overriding container settings \(p. 115\)](#)
- [Updating function code \(p. 115\)](#)
- [Using the Lambda API \(p. 116\)](#)
- [AWS CloudFormation \(p. 116\)](#)

Prerequisites

To complete the following steps, you need the [AWS Command Line Interface \(AWS CLI\) version 2](#). Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.0.57 Python/3.7.4 Darwin/19.6.0 exe/x86_64
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#). Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

Before you create the function, you must [create a container image and upload it to Amazon ECR \(p. 881\)](#).

Permissions

Make sure that the permissions for the user or role that creates the function contain the AWS managed policies `GetRepositoryPolicy` and `SetRepositoryPolicy`.

For example, use the IAM console to create a role with the following policy:

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Sid": "VisualEditor0",  
      "Effect": "Allow",  
      "Action": ["ecr:SetRepositoryPolicy", "ecr:GetRepositoryPolicy"],  
      "Resource": "arn:aws:ecr:<region>:<account>:repository/<repo name>/"  
    }  
  ]  
}
```

Amazon ECR permissions

For a function in the same account as the container image in Amazon ECR, you can add `ecr:BatchGetImage` and `ecr:GetDownloadUrlForLayer` permissions to your Amazon ECR repository. The following example shows the minimum policy:

```
{  
  "Sid": "LambdaECRImageRetrievalPolicy",  
  "Effect": "Allow",  
  "Principal": {  
    "Service": "lambda.amazonaws.com"  
  },  
  "Action": [  
    "ecr:BatchGetImage",  
    "ecr:GetDownloadUrlForLayer"  
  ]  
}
```

For more information about Amazon ECR repository permissions, see [Repository policies](#) in the *Amazon Elastic Container Registry User Guide*.

If the Amazon ECR repository does not include these permissions, Lambda adds `ecr:BatchGetImage` and `ecr:GetDownloadUrlForLayer` to the container image repository permissions. Lambda can add these permissions only if the Principal calling Lambda has `ecr:getRepositoryPolicy` and `ecr:setRepositoryPolicy` permissions.

To view or edit your Amazon ECR repository permissions, follow the directions in [Setting a repository policy statement](#) in the *Amazon Elastic Container Registry User Guide*.

Amazon ECR cross-account permissions

A different account in the same region can create a function that uses a container image owned by your account. In the following example, your Amazon ECR repository permissions policy needs the following statements to grant access to account number 123456789012.

- **CrossAccountPermission** – Allows account 123456789012 to create and update Lambda functions that use images from this ECR repository.
- **LambdaECRImageCrossAccountRetrievalPolicy** – Lambda will eventually set a function's state to inactive if it is not invoked for an extended period. This statement is required so that Lambda

can retrieve the container image for optimization and caching on behalf of the function owned by 123456789012.

Example Add cross-account permission to your repository

```
{"Version": "2012-10-17",
 "Statement": [
   {
     "Sid": "CrossAccountPermission",
     "Effect": "Allow",
     "Action": [
       "ecr:BatchGetImage",
       "ecr:GetDownloadUrlForLayer"
     ],
     "Principal": {
       "AWS": "arn:aws:iam::123456789012:root"
     }
   },
   {
     "Sid": "LambdaECRImageCrossAccountRetrievalPolicy",
     "Effect": "Allow",
     "Action": [
       "ecr:BatchGetImage",
       "ecr:GetDownloadUrlForLayer"
     ],
     "Principal": {
       "Service": "lambda.amazonaws.com"
     },
     "Condition": {
       "StringLike": {
         "aws:sourceARN":
           "arn:aws:lambda:us-east-1:123456789012:function:/*"
       }
     }
   }
 ]}
```

To give access to multiple accounts, you add the account IDs to the Principal list in the CrossAccountPermission policy and to the Condition evaluation list in the LambdaECRImageCrossAccountRetrievalPolicy.

If you are working with multiple accounts in an AWS Organization, we recommend that you enumerate each account ID in the ECR permissions policy. This approach aligns with the AWS security best practice of setting narrow permissions in IAM policies.

Creating the function

To create a function defined as a container image, you must first [create the image \(p. 881\)](#) and then store the image in the Amazon ECR repository.

To create the function

1. Open the [Functions page](#) of the Lambda console.
2. Choose **Create function**.
3. Choose the **Container image** option.
4. Under **Basic information**, do the following:
 - a. For **Function name**, enter the function name.

- b. For **Container image URI**, provide a container image that is compatible with the instruction set architecture that you want for your function code.

You can enter the Amazon ECR image URI or browse for the Amazon ECR image.

- Enter the Amazon ECR image URI.
- Or, to browse an Amazon ECR repository for the image, choose **Browse images**. Select the Amazon ECR repository from the dropdown list, and then select the image.

5. (Optional) To override configuration settings that are included in the Dockerfile, expand **Container image overrides**. You can override any of the following settings:

- For **Entrypoint**, enter the full path of the runtime executable. The following example shows an entrypoint for a Node.js function:

```
"/usr/bin/npx", "aws-lambda-ric"
```

- For **Command**, enter additional parameters to pass in to the image with **Entrypoint**. The following example shows a command for a Node.js function:

```
"app.handler"
```

- For **Working directory**, enter the full path of the working directory for the function. The following example shows the working directory for an AWS base image for Lambda:

```
"/var/task"
```

Note

For the override settings, make sure that you enclose each string in quotation marks ("").

6. (Optional) For **Architecture**, choose the instruction set architecture for the function. The default architecture is x86_64. Note: when you build the container image for your function, make sure that it is compatible with this [instruction set architecture \(p. 29\)](#).
7. (Optional) Under **Permissions**, expand **Change default execution role**. Then, choose to create a new **Execution role**, or to use an existing role.
8. Choose **Create function**.

Lambda creates your function and an [execution role \(p. 816\)](#) that grants the function permission to upload logs. Lambda assumes the execution role when you invoke your function, and uses the execution role to create credentials for the AWS SDK and to read data from event sources.

When you deploy code as a container image to a Lambda function, the image undergoes an optimization process for running on Lambda. This process can take a few seconds, during which the function is in pending state. When the optimization process completes, the function enters the active state.

Testing the function

Invoke your Lambda function using the sample event data provided in the console.

To invoke a function

1. After selecting your function, choose the **Test** tab.
2. For **Test event action**, choose **Create new event**.
3. For **Event name**, enter a name for the test event.
4. For **Event sharing settings**, choose **Private**.
5. For **Template**, leave the default **hello-world** option.

6. Choose **Save changes**, and then choose **Test**. Lambda runs your function on your behalf. The function handler receives and then processes the sample event.
7. Review the **Execution result** and **Details**:
 - The **Summary** section shows the key information from the REPORT line in the invocation log.
 - The **Log output** section shows the complete invocation log. Lambda writes all invocation logs to Amazon CloudWatch.

Overriding container settings

You can use the Lambda console or the Lambda API to override the following container image settings:

- **ENTRYPOINT** – Specifies the absolute path of the entry point to the application.
- **CMD** – Specifies parameters that you want to pass in with ENTRYPOINT.
- **WORKDIR** – Specifies the absolute path of the working directory.
- **ENV** – Specifies an environment variable for the Lambda function.

Any values that you provide in the Lambda console or the Lambda API override the values [in the Dockerfile \(p. 888\)](#).

To override the configuration values in the container image

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to update.
3. Under **Image configuration**, choose **Edit**.
4. Enter new values for any of the override settings, and then choose **Save**.
5. (Optional) To add or override environment variables, under **Environment variables**, choose **Edit**.

For more information, see [the section called "Environment variables" \(p. 76\)](#).

Updating function code

After you deploy a container image to a function, the image is read-only. To update the function code, you must first deploy a new image version. [Create a new image version \(p. 881\)](#), and then store the image in the Amazon ECR repository.

To configure the function to use an updated container image

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to update.
3. Under **Image**, choose **Deploy new image**.
4. Choose **Browse images**.
5. In the **Select container image** dialog box, select the Amazon ECR repository from the dropdown list, and then select the new image version.
6. Choose **Save**.

Function version \$LATEST

When you publish a function version, the code and most of the configuration settings are locked to maintain a consistent experience for users of that version. You can change the code and many

configuration settings only on the unpublished version of the function. By default, the console displays configuration information for the unpublished version of the function. To view the versions of a function, choose **Qualifiers**. The unpublished version is named **\$LATEST**.

Note that Amazon Elastic Container Registry (Amazon ECR) also uses a *latest* tag to denote the latest version of the container image. Be careful not to confuse this tag with the **\$LATEST** function version.

For more information about managing versions, see [Lambda function versions \(p. 83\)](#).

Using the Lambda API

To manage functions defined as container images, use the following API operations:

- [CreateFunction \(p. 1165\)](#)
- [UpdateFunctionCode \(p. 1367\)](#)
- [UpdateFunctionConfiguration \(p. 1377\)](#)

To create a function defined as container image, use the `create-function` command. Set the `package-type` to `Image` and specify your container image URI using the `code` parameter.

When you create the function, you can specify the instruction set architecture. The default architecture is `x86-64`. Make sure that the code in your container image is compatible with the architecture.

You can create the function from the same account as the container registry or from a different account in the same region as the container registry in Amazon ECR. For cross-account access, adjust the [Amazon ECR permissions \(p. 112\)](#) for the image.

```
aws lambda create-function --region sa-east-1 --function-name my-function \
--package-type Image \
--code ImageUri=<ECR Image URI> \
--role arn:aws:iam::123456789012:role/lambda-ex
```

To update the function code, use the `update-function-code` command. Specify the container image location using the `image-uri` parameter.

Note

You cannot change the `package-type` of a function.

```
aws lambda update-function-code --region sa-east-1 --function-name my-function \
--image-uri <ECR Image URI> \
```

To update the function parameters, use the `update-function-configuration` operation. Specify `EntryPoint` and `Command` as arrays of strings, and `WorkingDirectory` as a string.

```
aws lambda update-function-configuration --function-name my-function \
--image-config '{"EntryPoint": ["/usr/bin/npx", "aws-lambda-ric"], \
"Command": ["app.handler"], \
"WorkingDirectory": "/var/task"}'
```

AWS CloudFormation

You can use AWS CloudFormation to create Lambda functions defined as container images. In your AWS CloudFormation template, the `AWS::Lambda::Function` resource specifies the Lambda function. For descriptions of the properties in the `AWS::Lambda::Function` resource, see [AWS::Lambda::Function](#) in the *AWS CloudFormation User Guide*.

In the AWS::Lambda::Function resource, set the following properties to create a function defined as a container image:

- AWS::Lambda::Function
 - PackageType – Set to Image.
 - Code – Enter your container image URI in the ImageUri field.
 - ImageConfig – (Optional) Override the container image configuration properties.

The ImageConfig property in AWS::Lambda::Function contains the following fields:

- Command – Specifies parameters that you want to pass in with EntryPoint.
- EntryPoint – Specifies the entry point to the application.
- WorkingDirectory – Specifies the working directory.

Note

If you declare an ImageConfig property in your AWS CloudFormation template, you must provide values for all three of the ImageConfig properties.

For more information, see [ImageConfig](#) in the *AWS CloudFormation User Guide*.

Invoking Lambda functions

You can invoke Lambda functions directly using [the Lambda console \(p. 114\)](#), a [function URL \(p. 166\)](#) HTTP(S) endpoint, the [Lambda API](#), an [AWS SDK](#), the [AWS Command Line Interface \(AWS CLI\)](#), and [AWS toolkits](#). You can also configure other AWS services to invoke your function in response to events or external requests, or on a schedule. For example, Amazon Simple Storage Service (Amazon S3) can invoke your function when an object is created in an S3 bucket, or Amazon EventBridge (CloudWatch Events) can invoke your function on a schedule. You can also configure Lambda to read items from a stream or a queue and invoke your function to process them.

When you invoke a function, you can choose to invoke it synchronously or asynchronously. With [synchronous invocation \(p. 120\)](#), you wait for the function to process the event and return a response. With [asynchronous invocation \(p. 123\)](#), Lambda queues the event for processing and returns a response immediately. For asynchronous invocation, Lambda handles retries and can send invocation records to a [destination \(p. 125\)](#).

For another AWS service to invoke your function directly, you need to create a trigger using the Lambda console. A trigger is a resource you configure to allow another AWS service to invoke your function when certain events or conditions occur. Your function can have multiple triggers. Each trigger acts as a client invoking your function independently, and each event that Lambda passes to your function has data from only one trigger.

To create a trigger, open the [functions page](#) of the Lambda console and choose the function you want to add a trigger to. In the **Function overview** pane, choose **add trigger**, select the AWS service you want to invoke your function, and follow the instructions to create a trigger.

Each service varies in the options you configure for the trigger and in the structure of the event that the service sends to your Lambda function to invoke it. For a full list of the AWS services that can invoke your Lambda function by using a trigger, and for more information about configuring triggers for different services, see [Using Lambda with other services. \(p. 556\)](#)

For your Lambda function to process items from a stream or a queue, such as an Amazon Kinesis stream or an Amazon Simple Queue Service (Amazon SQS) queue, you need to create an [event source mapping \(p. 131\)](#). An event source mapping is a resource in Lambda that reads items from a stream or a queue and creates events containing batches of items to send to your Lambda function. Each event that your function processes can contain hundreds or thousands of items.

You can create an event source mapping for your Lambda function using the Lambda console, the AWS CLI, the Lambda API, or an AWS SDK. To create an event source mapping in the Lambda console, follow the instructions to create a trigger, and select one of the AWS services that support event source mappings as your source. To create an event source mapping using the AWS CLI, Lambda API, or an AWS SDK, and to see a list of the AWS services which event source mappings can be used with, refer to [Lambda event source mappings \(p. 131\)](#).

Depending on how your function is invoked, scaling behavior and the types of errors that occur can vary. When you invoke a function synchronously, you receive errors in the response and can retry. When you invoke asynchronously, use an event source mapping, or configure another service to invoke your function, the retry requirements and the way that your function scales to handle large numbers of events will vary. For more information, see [Error handling and automatic retries in AWS Lambda \(p. 161\)](#).

Topics

- [Synchronous invocation \(p. 120\)](#)
- [Asynchronous invocation \(p. 123\)](#)
- [Lambda event source mappings \(p. 131\)](#)

- [Lambda event filtering \(p. 136\)](#)
- [Lambda function states \(p. 159\)](#)
- [Error handling and automatic retries in AWS Lambda \(p. 161\)](#)
- [Testing Lambda functions in the console \(p. 163\)](#)
- [Invoking functions defined as container images \(p. 165\)](#)
- [Lambda function URLs \(p. 166\)](#)

Synchronous invocation

When you invoke a function synchronously, Lambda runs the function and waits for a response. When the function completes, Lambda returns the response from the function's code with additional data, such as the version of the function that was invoked. To invoke a function synchronously with the AWS CLI, use the `invoke` command.

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --payload '{ "key": "value" }' response.json
```

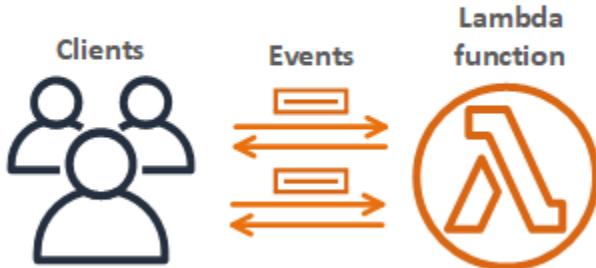
The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

You should see the following output:

```
{  
    "ExecutedVersion": "$LATEST",  
    "StatusCode": 200  
}
```

The following diagram shows clients invoking a Lambda function synchronously. Lambda sends the events directly to the function and sends the function's response back to the invoker.

Synchronous Invocation



The payload is a string that contains an event in JSON format. The name of the file where the AWS CLI writes the response from the function is `response.json`. If the function returns an object or error, the response is the object or error in JSON format. If the function exits without error, the response is null.

The output from the command, which is displayed in the terminal, includes information from headers in the response from Lambda. This includes the version that processed the event (useful when you use [aliases \(p. 85\)](#)), and the status code returned by Lambda. If Lambda was able to run the function, the status code is 200, even if the function returned an error.

Note

For functions with a long timeout, your client might be disconnected during synchronous invocation while it waits for a response. Configure your HTTP client, SDK, firewall, proxy, or operating system to allow for long connections with timeout or keep-alive settings.

If Lambda isn't able to run the function, the error is displayed in the output.

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --payload value response.json
```

You should see the following output:

```
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]
```

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

You can use the [AWS CLI](#) to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{  
    "StatusCode": 200,  
    "LogResult":  
        "U1RBULQgUmVxdWVzdElkOiaA4N2QwNDRiOC1mMTU0LTExZTgt0GNkYS0y0Tc0YzVlNGZiMjEgVmVyc21vb...",  
    "ExecutedVersion": "$LATEST"  
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \  
--query 'LogResult' --output text | base64 -d
```

You should see the following output:

```
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
```

The `base64` utility is available on Linux, macOS, and [Ubuntu on Windows](#). macOS users may need to use `base64 -D`.

For more information about the Invoke API, including a full list of parameters, headers, and errors, see [Invoke \(p. 1260\)](#).

When you invoke a function directly, you can check the response for errors and retry. The AWS CLI and AWS SDK also automatically retry on client timeouts, throttling, and service errors. For more information, see [Error handling and automatic retries in AWS Lambda \(p. 161\)](#).

Asynchronous invocation

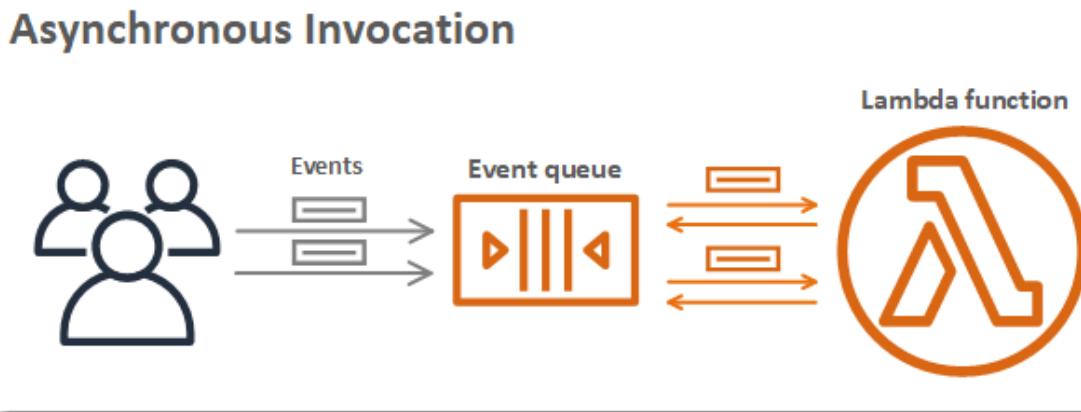
Several AWS services, such as Amazon Simple Storage Service (Amazon S3) and Amazon Simple Notification Service (Amazon SNS), invoke functions asynchronously to process events. When you invoke a function asynchronously, you don't wait for a response from the function code. You hand off the event to Lambda and Lambda handles the rest. You can configure how Lambda handles errors, and can send invocation records to a downstream resources such as Amazon Simple Queue Service (Amazon SQS) or Amazon EventBridge (EventBridge) to chain together components of your application.

Sections

- [How Lambda handles asynchronous invocations \(p. 123\)](#)
- [Configuring error handling for asynchronous invocation \(p. 125\)](#)
- [Configuring destinations for asynchronous invocation \(p. 125\)](#)
- [Asynchronous invocation configuration API \(p. 128\)](#)
- [Dead-letter queues \(p. 129\)](#)

How Lambda handles asynchronous invocations

The following diagram shows clients invoking a Lambda function asynchronously. Lambda queues the events before sending them to the function.



For asynchronous invocation, Lambda places the event in a queue and returns a success response without additional information. A separate process reads events from the queue and sends them to your function. To invoke a function asynchronously, set the invocation type parameter to Event.

```
aws lambda invoke \
--function-name my-function \
--invocation-type Event \
--cli-binary-format raw-in-base64-out \
--payload '{ "key": "value" }' response.json
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

```
{  
  "StatusCode": 202}
```

}

The output file (`response.json`) doesn't contain any information, but is still created when you run this command. If Lambda isn't able to add the event to the queue, the error message appears in the command output.

Lambda manages the function's asynchronous event queue and attempts to retry on errors. If the function returns an error, Lambda attempts to run it two more times, with a one-minute wait between the first two attempts, and two minutes between the second and third attempts. Function errors include errors returned by the function's code and errors returned by the function's runtime, such as timeouts.

If the function doesn't have enough concurrency available to process all events, additional requests are throttled. For throttling errors (429) and system errors (500-series), Lambda returns the event to the queue and attempts to run the function again for up to 6 hours. The retry interval increases exponentially from 1 second after the first attempt to a maximum of 5 minutes. If the queue contains many entries, Lambda increases the retry interval and reduces the rate at which it reads events from the queue.

Even if your function doesn't return an error, it's possible for it to receive the same event from Lambda multiple times because the queue itself is eventually consistent. If the function can't keep up with incoming events, events might also be deleted from the queue without being sent to the function. Ensure that your function code gracefully handles duplicate events, and that you have enough concurrency available to handle all invocations.

When the queue is very long, new events might age out before Lambda has a chance to send them to your function. When an event expires or fails all processing attempts, Lambda discards it. You can [configure error handling \(p. 125\)](#) for a function to reduce the number of retries that Lambda performs, or to discard unprocessed events more quickly.

You can also configure Lambda to send an invocation record to another service. Lambda supports the following [destinations \(p. 125\)](#) for asynchronous invocation. Note that SQS FIFO queues and SNS FIFO topics are not supported.

- **Amazon SQS** – A standard SQS queue.
- **Amazon SNS** – A standard SNS topic.
- **AWS Lambda** – A Lambda function.
- **Amazon EventBridge** – An EventBridge event bus.

The invocation record contains details about the request and response in JSON format. You can configure separate destinations for events that are processed successfully, and events that fail all processing attempts. Alternatively, you can configure a standard Amazon SQS queue or standard Amazon SNS topic as a [dead-letter queue \(p. 129\)](#) for discarded events. For dead-letter queues, Lambda only sends the content of the event, without details about the response.

If Lambda can't send a record to a destination you have configured, it sends a `DestinationDeliveryFailures` metric to Amazon CloudWatch. This can happen if your configuration includes an unsupported destination type, such as an Amazon SQS FIFO queue or an Amazon SNS FIFO topic. Delivery errors can also occur due to permissions errors and size limits. For more information on Lambda invocation metrics, see [Invocation metrics \(p. 871\)](#).

Note

To prevent a function from triggering, you can set the function's reserved concurrency to zero. When you set reserved concurrency to zero for an asynchronously-invoked function, Lambda begins sending new events to the configured [dead-letter queue \(p. 129\)](#) or the [on-failure event destination \(p. 125\)](#), without any retries. To process events that were sent while reserved concurrency was set to zero, you need to consume the events from the dead-letter queue or the on-failure event destination.

Configuring error handling for asynchronous invocation

Use the Lambda console to configure error handling settings on a function, a version, or an alias.

To configure error handling

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Asynchronous invocation**.
4. Under **Asynchronous invocation**, choose **Edit**.
5. Configure the following settings.
 - **Maximum age of event** – The maximum amount of time Lambda retains an event in the asynchronous event queue, up to 6 hours.
 - **Retry attempts** – The number of times Lambda retries when the function returns an error, between 0 and 2.
6. Choose **Save**.

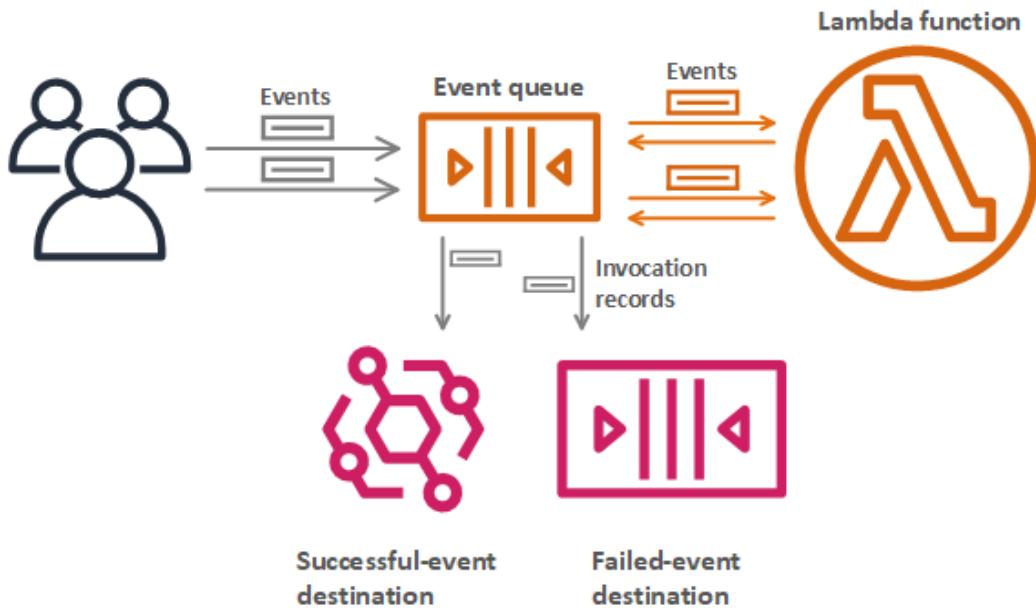
When an invocation event exceeds the maximum age or fails all retry attempts, Lambda discards it. To retain a copy of discarded events, configure a failed-event destination.

Configuring destinations for asynchronous invocation

To send records of asynchronous invocations to another service, add a destination to your function. You can configure separate destinations for events that fail processing and events that are successfully processed. Like error handling settings, you can configure destinations on a function, a version, or an alias.

The following example shows a function that is processing asynchronous invocations. When the function returns a success response or exits without throwing an error, Lambda sends a record of the invocation to an EventBridge event bus. When an event fails all processing attempts, Lambda sends an invocation record to a standard Amazon SQS queue.

Destinations for Asynchronous Invocation



To send events to a destination, your function needs additional permissions. Add a policy with the required permissions to your function's [execution role \(p. 816\)](#). Each destination service requires a different permission, as follows:

- **Amazon SQS** – [sq:SendMessage](#)
- **Amazon SNS** – [sns:Publish](#)
- **Lambda** – [InvokeFunction \(p. 1260\)](#)
- **EventBridge** – [events:PutEvents](#)

Add destinations to your function in the Lambda console's function visualization.

To configure a destination for asynchronous invocation records

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Under **Function overview**, choose **Add destination**.
4. For **Source**, choose **Asynchronous invocation**.
5. For **Condition**, choose from the following options:
 - **On failure** – Send a record when the event fails all processing attempts or exceeds the maximum age.
 - **On success** – Send a record when the function successfully processes an asynchronous invocation.
6. For **Destination type**, choose the type of resource that receives the invocation record.
7. For **Destination**, choose a resource.
8. Choose **Save**.

When an invocation matches the condition, Lambda sends a JSON document with details about the invocation to the destination.

Destination-specific JSON format

- For Amazon SQS and Amazon SNS (`SnsDestination` and `SqsDestination`), the invocation record is passed as the `Message` to the destination.
- For Lambda (`LambdaDestination`), the invocation record is passed as the payload to the function.
- For EventBridge (`EventBridgeDestination`), the invocation record is passed as the detail in the [PutEvents](#) call. The value for the `source` event field is `lambda`. The value for the `detail-type` event field is either *Lambda Function Invocation Result – Success* or *Lambda Function Invocation Result – Failure*. The `resource` event field contains the function and destination Amazon Resource Names (ARNs). For other event fields, see [Amazon EventBridge events](#).

The following example shows an invocation record for an event that failed three processing attempts due to a function error.

Example invocation record

```
{
    "version": "1.0",
    "timestamp": "2019-11-14T18:16:05.568Z",
    "requestContext": {
        "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012: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"
    }
}
```

The invocation record contains details about the event, the response, and the reason that the record was sent.

Tracing requests to destinations

You can use X-Ray to see a connected view of each request as it's queued, processed by a Lambda function, and passed to the destination service. When you activate X-Ray tracing for a function or a service that invokes a function, Lambda adds an X-Ray header to the request and passes the header to the destination service. Traces from upstream services are automatically linked to traces from downstream Lambda functions and destination services, creating an end-to-end view of the entire application. For more information about tracing, see [Using AWS Lambda with AWS X-Ray \(p. 807\)](#).

Asynchronous invocation configuration API

To manage asynchronous invocation settings with the AWS CLI or AWS SDK, use the following API operations.

- [PutFunctionEventInvokeConfig](#)
- [GetFunctionEventInvokeConfig](#)
- [UpdateFunctionEventInvokeConfig](#)
- [ListFunctionEventInvokeConfigs](#)
- [DeleteFunctionEventInvokeConfig](#)

To configure asynchronous invocation with the AWS CLI, use the `put-function-event-invoke-config` command. The following example configures a function with a maximum event age of 1 hour and no retries.

```
aws lambda put-function-event-invoke-config --function-name error \
--maximum-event-age-in-seconds 3600 --maximum-retry-attempts 0
```

You should see the following output:

```
{
    "LastModified": 1573686021.479,
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:error:$LATEST",
    "MaximumRetryAttempts": 0,
    "MaximumEventAgeInSeconds": 3600,
    "DestinationConfig": {
        "OnSuccess": {},
        "OnFailure": {}
    }
}
```

The `put-function-event-invoke-config` command overwrites any existing configuration on the function, version, or alias. To configure an option without resetting others, use `update-function-event-invoke-config`. The following example configures Lambda to send a record to a standard SQS queue named `destination` when an event can't be processed.

```
aws lambda update-function-event-invoke-config --function-name error \
--destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-\
east-2:123456789012:destination"}}'
```

You should see the following output:

```
{
    "LastModified": 1573687896.493,
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:error:$LATEST",
    "MaximumRetryAttempts": 0,
    "MaximumEventAgeInSeconds": 3600,
    "DestinationConfig": {
        "OnSuccess": {},
        "OnFailure": {
            "Destination": "arn:aws:sqs:us-east-2:123456789012:destination"
        }
    }
}
```

Dead-letter queues

As an alternative to an [on-failure destination \(p. 125\)](#), you can configure your function with a dead-letter queue to save discarded events for further processing. A dead-letter queue acts the same as an on-failure destination in that it is used when an event fails all processing attempts or expires without being processed. However, a dead-letter queue is part of a function's version-specific configuration, so it is locked in when you publish a version. On-failure destinations also support additional targets and include details about the function's response in the invocation record.

To reprocess events in a dead-letter queue, you can set it as an event source for your Lambda function. Alternatively, you can manually retrieve the events.

You can choose an Amazon SQS standard queue or Amazon SNS standard topic for your dead-letter queue. FIFO queues and Amazon SNS FIFO topics are not supported. If you don't have a queue or topic, create one. Choose the target type that matches your use case.

- [Amazon SQS queue](#) – A queue holds failed events until they're retrieved. Choose an Amazon SQS standard queue if you expect a single entity, such as a Lambda function or CloudWatch alarm, to process the failed event. For more information, see [Using Lambda with Amazon SQS \(p. 778\)](#).

Create a queue in the [Amazon SQS console](#).

- [Amazon SNS topic](#) – A topic relays failed events to one or more destinations. Choose an Amazon SNS standard topic if you expect multiple entities to act on a failed event. For example, you can configure a topic to send events to an email address, a Lambda function, and/or an HTTP endpoint. For more information, see [Using AWS Lambda with Amazon SNS \(p. 770\)](#).

Create a topic in the [Amazon SNS console](#).

To send events to a queue or topic, your function needs additional permissions. Add a policy with the required permissions to your function's [execution role \(p. 816\)](#).

- [Amazon SQS](#) – `sqs:SendMessage`
- [Amazon SNS](#) – `sns:Publish`

If the target queue or topic is encrypted with a customer managed key, the execution role must also be a user in the key's [resource-based policy](#).

After creating the target and updating your function's execution role, add the dead-letter queue to your function. You can configure multiple functions to send events to the same target.

To configure a dead-letter queue

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Asynchronous invocation**.
4. Under **Asynchronous invocation**, choose **Edit**.
5. Set **DLQ resource** to **Amazon SQS** or **Amazon SNS**.
6. Choose the target queue or topic.
7. Choose **Save**.

To configure a dead-letter queue with the AWS CLI, use the `update-function-configuration` command.

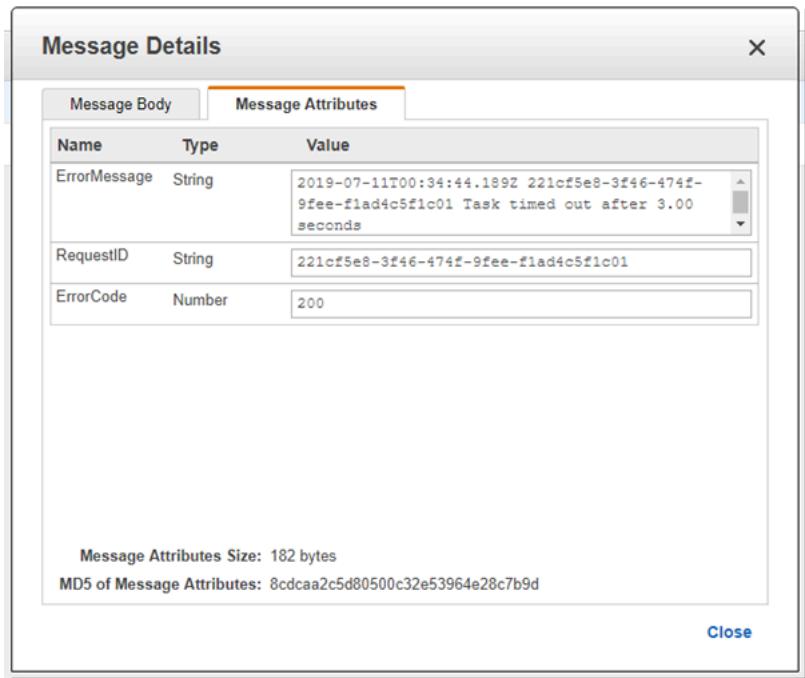
```
aws lambda update-function-configuration --function-name my-function \
```

```
--dead-letter-config TargetArn=arn:aws:sns:us-east-2:123456789012:my-topic
```

Lambda sends the event to the dead-letter queue as-is, with additional information in attributes. You can use this information to identify the error that the function returned, or to correlate the event with logs or an AWS X-Ray trace.

Dead-letter queue message attributes

- **RequestID** (String) – The ID of the invocation request. Request IDs appear in function logs. You can also use the X-Ray SDK to record the request ID on an attribute in the trace. You can then search for traces by request ID in the X-Ray console. For an example, see the [error processor sample \(p. 1015\)](#).
- **ErrorCode** (Number) – The HTTP status code.
- **ErrorMessage** (String) – The first 1 KB of the error message.



If Lambda can't send a message to the dead-letter queue, it deletes the event and emits the [DeadLetterErrors \(p. 870\)](#) metric. This can happen because of lack of permissions, or if the total size of the message exceeds the limit for the target queue or topic. For example, if an Amazon SNS notification with a body close to 256 KB triggers a function that results in an error, the additional event data added by Amazon SNS, combined with the attributes added by Lambda, can cause the message to exceed the maximum size allowed in the dead-letter queue.

If you're using Amazon SQS as an event source, configure a dead-letter queue on the Amazon SQS queue itself and not on the Lambda function. For more information, see [Using Lambda with Amazon SQS \(p. 778\)](#).

Lambda event source mappings

Note

If you want to send data to a target other than a Lambda function or enrich the data before sending it, see [Amazon EventBridge Pipes](#).

An event source mapping is a Lambda resource that reads from an event source and invokes a Lambda function. You can use event source mappings to process items from a stream or queue in services that don't invoke Lambda functions directly. This page describes the services that Lambda provides event source mappings and how-to fine tune batching behavior.

Services that Lambda reads events from

- [Amazon DynamoDB \(p. 635\)](#)
- [Amazon Kinesis \(p. 684\)](#)
- [Amazon MQ \(p. 708\)](#)
- [Amazon Managed Streaming for Apache Kafka \(Amazon MSK\) \(p. 716\)](#)
- [Self-managed Apache Kafka \(p. 671\)](#)
- [Amazon Simple Queue Service \(Amazon SQS\) \(p. 778\)](#)
- [Amazon DocumentDB \(with MongoDB compatibility\) \(Amazon DocumentDB\) \(p. 610\)](#)

An event source mapping uses permissions in the function's [execution role \(p. 816\)](#) to read and manage items in the event source. Permissions, event structure, settings, and polling behavior vary by event source. For more information, see the linked topic for the service that you use as an event source.

To manage an event source with the [AWS Command Line Interface \(AWS CLI\)](#) or an [AWS SDK](#), you can use the following API operations:

- [CreateEventSourceMapping \(p. 1153\)](#)
- [ListEventSourceMappings \(p. 1279\)](#)
- [GetEventSourceMapping \(p. 1214\)](#)
- [UpdateEventSourceMapping \(p. 1356\)](#)
- [DeleteEventSourceMapping \(p. 1186\)](#)

Note

When you update, disable, or delete an event source mapping for Amazon MQ, Amazon MSK, self-managed Apache Kafka, or Amazon DocumentDB, it can take up to 15 minutes for your changes to take effect. Before this period has elapsed, your event source mapping may continue to process events and invoke your function using your previous settings. This is true even when the status of the event source mapping displayed in the console indicates that your changes have been applied.

The following example uses the AWS CLI to map a function named my-function to a DynamoDB stream that its Amazon Resource Name (ARN) specifies, with a batch size of 500.

```
aws lambda create-event-source-mapping --function-name my-function --batch-size 500 --maximum-batching-window-in-seconds 5 --starting-position LATEST \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/stream/2019-06-10T19:26:16.525
```

You should see the following output:

```
{
```

```
"UUID": "14e0db71-5d35-4eb5-b481-8945cf9d10c2",
"BatchSize": 500,
"MaximumBatchingWindowInSeconds": 5,
"ParallelizationFactor": 1,
"EventSourceArn": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/stream/2019-06-10T19:26:16.525",
"FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
"LastModified": 1560209851.963,
"LastProcessingResult": "No records processed",
"State": "Creating",
"StateTransitionReason": "User action",
"DestinationConfig": {},
"MaximumRecordAgeInSeconds": 604800,
"BisectBatchOnFunctionError": false,
"MaximumRetryAttempts": 10000
}
```

Lambda event source mappings process events at least once due to the distributed nature of its pollers. As a result, your Lambda function may receive duplicate events in rare situations. Follow [Best practices for working with AWS Lambda functions \(p. 811\)](#) and build idempotent functions to avoid issues related to duplicate events.

Batching behavior

Event source mappings read items from a target event source. By default, an event source mapping batches records together into a single payload that Lambda sends to your function. To fine-tune batching behavior, you can configure a batching window (`MaximumBatchingWindowInSeconds`) and a batch size (`BatchSize`). A batching window is the maximum amount of time to gather records into a single payload. A batch size is the maximum number of records in a single batch. Lambda invokes your function when one of the following three criteria is met:

- **The batching window reaches its maximum value.** Batching window behavior varies depending on the specific event source.
- **For Kinesis, DynamoDB, and Amazon SQS event sources:** The default batching window is 0 seconds. This means that Lambda sends batches to your function as quickly as possible. If you configure a `MaximumBatchingWindowInSeconds`, the next batching window begins as soon as the previous function invocation completes.
- **For Amazon MSK, self-managed Apache Kafka, and Amazon MQ event sources:** The default batching window is 500 ms. You can configure `MaximumBatchingWindowInSeconds` to any value from 0 seconds to 300 seconds in increments of seconds. A batching window begins as soon as the first record arrives.

Note

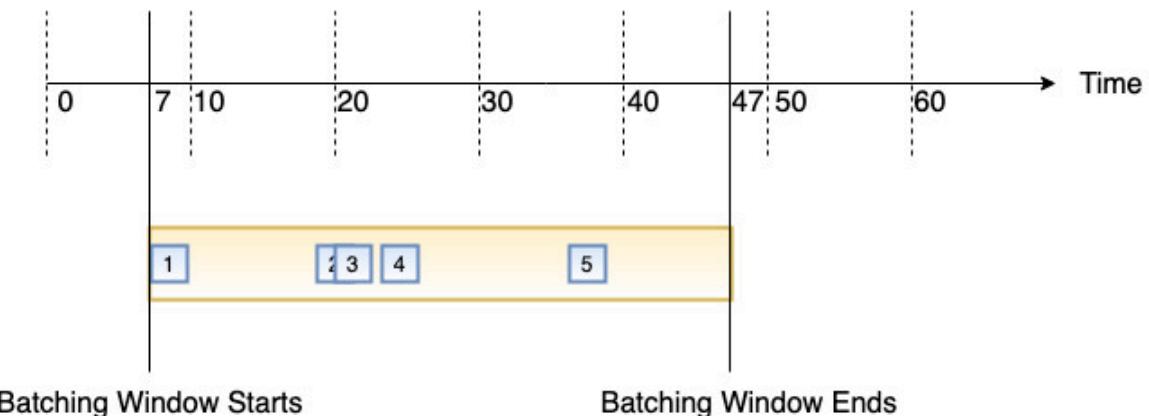
Because you can only change `MaximumBatchingWindowInSeconds` in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

- **The batch size is met.** The minimum batch size is 1. The default and maximum batch size depend on the event source. For details about these values, see the [BatchSize](#) specification for the `CreateEventSourceMapping` API operation.
- **The payload size reaches [6 MB](#).** You cannot modify this limit.

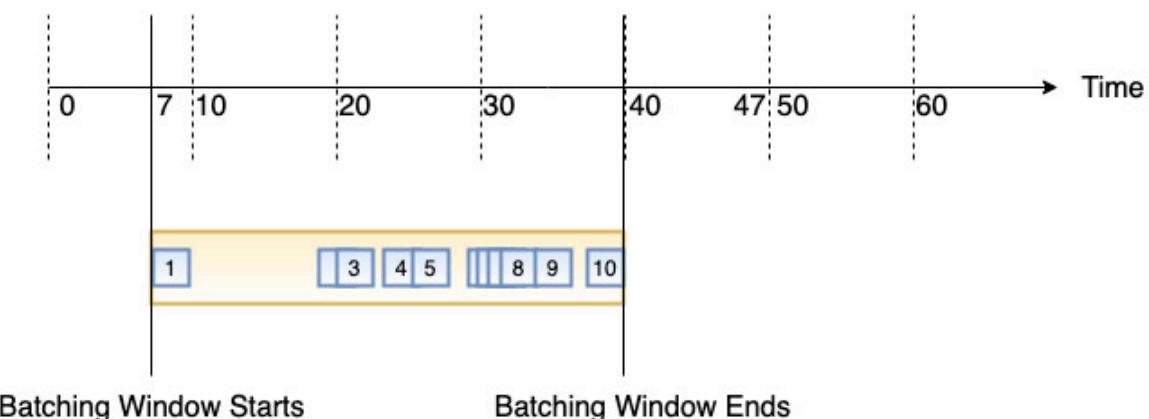
The following diagram illustrates these three conditions. Suppose a batching window begins at $t = 7$ seconds. In the first scenario, the batching window reaches its 40 second maximum at $t = 47$ seconds after accumulating 5 records. In the second scenario, the batch size reaches 10 before the batching window expires, so the batching window ends early. In the third scenario, the maximum payload size is reached before the batching window expires, so the batching window ends early.

Max Batching Window = 40 Seconds
Max Batch Size = 10
Max Batch Size in Bytes = 6 MB

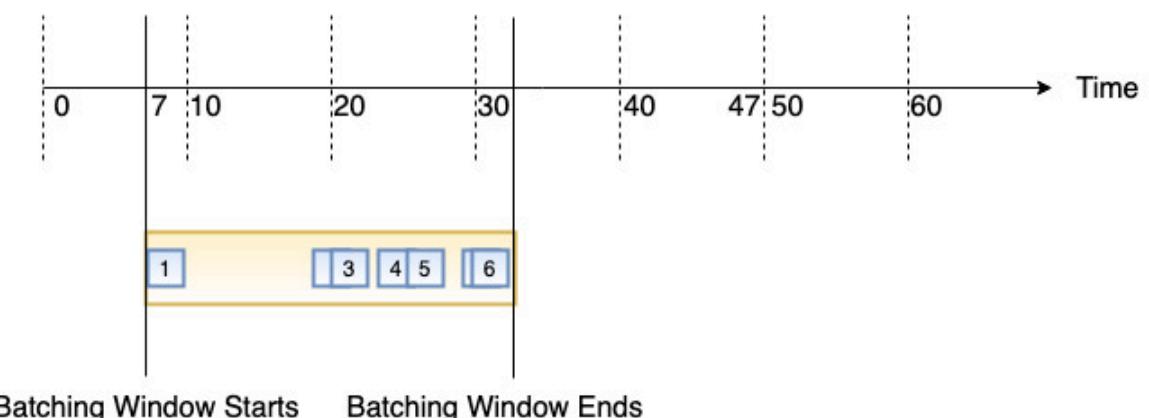
(1) Batching Window Expires



(2) Batching Size is reached

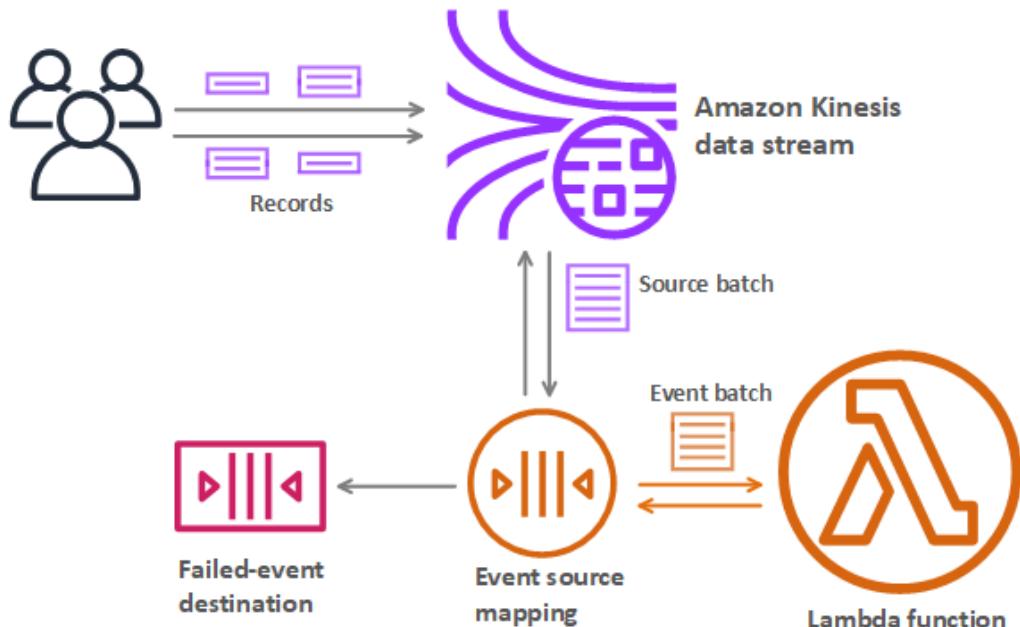


(3) Batch Size in bytes is reached



The following example shows an event source mapping that reads from a Kinesis stream. If a batch of events fails all processing attempts, the event source mapping sends details about the batch to an SQS queue.

Event Source Mapping with Kinesis Stream



The event batch is the event that Lambda sends to the function. It is a batch of records or messages compiled from the items that the event source mapping reads up until the current batching window expires.

For streams, an event source mapping creates an iterator for each shard in the stream and processes items in each shard in order. You can configure the event source mapping to read only new items that appear in the stream, or to start with older items. Processed items aren't removed from the stream, and other functions or consumers can process them.

By default, if your function returns an error, the event source mapping reprocesses the entire batch until the function succeeds, or the items in the batch expire. To ensure in-order processing, the event source mapping pauses processing for the affected shard until the error is resolved. You can configure the event source mapping to discard old events or process multiple batches in parallel. If you process multiple batches in parallel, in-order processing is still guaranteed for each partition key, but the event source mapping simultaneously processes multiple partition keys in the same shard.

For stream sources (DynamoDB and Kinesis), you can configure the maximum number of times that Lambda retries when your function returns an error. Service errors or throttles where the batch does not reach your function do not count toward retry attempts.

You can also configure the event source mapping to send an invocation record to another service when it discards an event batch. Lambda supports the following [destinations \(p. 125\)](#) for event source mappings.

- **Amazon SQS** – An SQS queue.
- **Amazon SNS** – An SNS topic.

The invocation record contains details about the failed event batch in JSON format.

The following example shows an invocation record for a Kinesis stream.

Example invocation record

```
{  
    "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"  
    }  
}
```

Lambda also supports in-order processing for [FIFO \(first-in, first-out\) queues \(p. 778\)](#), scaling up to the number of active message groups. For standard queues, items aren't necessarily processed in order. Lambda scales up to process a standard queue as quickly as possible. When an error occurs, Lambda returns batches to the queue as individual items and might process them in a different grouping than the original batch. Occasionally, the event source mapping might receive the same item from the queue twice, even if no function error occurred. Lambda deletes items from the queue after they're processed successfully. You can configure the source queue to send items to a dead-letter queue if Lambda can't process them.

For information about services that invoke Lambda functions directly, see [Using AWS Lambda with other services \(p. 556\)](#).

Lambda event filtering

You can use event filtering to control which records from a stream or queue Lambda sends to your function. For example, you can add a filter so that your function only processes Amazon SQS messages containing certain data parameters. Event filtering works with event source mappings. You can add filters to event source mappings for the following 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)

Lambda doesn't support event filtering for Amazon DocumentDB.

By default, you can define up to five different filters for a single event source mapping. Your filters are logically ORed together. If a record from your event source satisfies one or more of your filters, Lambda includes the record in the next event it sends to your function. If none of your filters are satisfied, Lambda discards the record.

Note

If you need to define more than five filters for an event source, you can request a quota increase up to 10 filters for each event source. If you attempt to add more filters than your current quota permits, Lambda will return an error when you try and create the event source.

Topics

- [Event filtering basics \(p. 136\)](#)
- [Handling records that don't meet filter criteria \(p. 138\)](#)
- [Filter rule syntax \(p. 138\)](#)
- [Attaching filter criteria to an event source mapping \(console\) \(p. 139\)](#)
- [Attaching filter criteria to an event source mapping \(AWS CLI\) \(p. 140\)](#)
- [Attaching filter criteria to an event source mapping \(AWS SAM\) \(p. 141\)](#)
- [Using filters with different AWS services \(p. 141\)](#)
- [Filtering with DynamoDB \(p. 142\)](#)
- [Filtering with Kinesis \(p. 145\)](#)
- [Filtering with Amazon MQ \(p. 147\)](#)
- [Filtering with Amazon MSK and self-managed Apache Kafka \(p. 152\)](#)
- [Filtering with Amazon SQS \(p. 155\)](#)

Event filtering basics

A filter criteria (`FilterCriteria`) object is a structure that consists of a list of filters (`Filters`). Each filter is a structure that defines an event filtering pattern (`Pattern`). A pattern is a string representation of a JSON filter rule. The structure of a `FilterCriteria` object is as follows.

```
{  
    "Filters": [  
        {  
            "Pattern": "{ \"Metadata1\": [ rule1 ], \"data\": { \"Data1\": [ rule2 ] }}"  
        }  
    ]  
}
```

```
        }
    ]
}
```

For added clarity, here is the value of the filter's Pattern expanded in plain JSON.

```
{
  "Metadata1": [ rule1 ],
  "data": {
    "Data1": [ rule2 ]
  }
}
```

Your filter pattern can include metadata properties, data properties, or both. The available metadata parameters and the format of the data parameters vary according to the AWS service which is acting as the event source. For example, suppose your event source mapping receives the following record from an Amazon SQS queue:

```
{
  "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": "AIDAENQZJ0L023YVJ4V0",
    "ApproximateFirstReceiveTimestamp": "1545082649185"
  },
  "messageAttributes": {},
  "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
  "eventSource": "aws:sqs",
  "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
  "awsRegion": "us-east-2"
}
```

- **Metadata properties** are the fields containing information about the event that created the record. In the example Amazon SQS record, the metadata properties include fields such as messageID, eventSourceArn, and awsRegion.
- **Data properties** are the fields of the record containing the data from your stream or queue. In the Amazon SQS event example, the key for the data field is body, and the data properties are the fields City, State, and Temperature.

Different types of event source use different key values for their data fields. To filter on data properties, make sure that you use the correct key in your filter's pattern. For a list of data filtering keys, and to see examples of filter patterns for each supported AWS service, refer to [Using filters with different AWS services \(p. 141\)](#).

Event filtering can handle multi-level JSON filtering. For example, consider the following fragment of a record from a DynamoDB stream:

```
"dynamodb": {
  "Keys": {
    "ID": {
      "S": "ABCD"
    }
    "Number": {
      "N": "1234"
    }
  },
  ...
}
```

```
}
```

Suppose you want to process only those records where the value of the sort key Number is 4567. In this case, your FilterCriteria object would look like this:

```
{
  "Filters": [
    {
      "Pattern": "{ \"dynamodb\": { \"Keys\": { \"Number\": { \"N\": [
        \"4567\" ] } } } }"
    }
  ]
}
```

For added clarity, here is the value of the filter's Pattern expanded in plain JSON.

```
{
  "dynamodb": {
    "Keys": {
      "Number": {
        "N": [ "4567" ]
      }
    }
  }
}
```

Handling records that don't meet filter criteria

The way in which records that don't meet your filter are handled depends on the event source.

- For **Amazon SQS**, if a message doesn't satisfy your filter criteria, Lambda automatically removes the message from the queue. You don't have to manually delete these messages in Amazon SQS.
- For **Kinesis** and **DynamoDB**, once your filter criteria processes a record, the streams iterator advances past this record. If the record doesn't satisfy your filter criteria, you don't have to manually delete the record from your event source. After the retention period, Kinesis and DynamoDB automatically delete these old records. If you want records to be deleted sooner, see [Changing the Data Retention Period](#).
- For **Amazon MSK**, **self-managed Apache Kafka**, and **Amazon MQ** messages, Lambda drops messages that don't match all fields included in the filter. For self-managed Apache Kafka, Lambda commits offsets for matched and unmatched messages after successfully invoking the function. For Amazon MQ, Lambda acknowledges matched messages after successfully invoking the function and acknowledges unmatched messages when filtering them.

Filter rule syntax

For filter rules, Lambda supports a subset of the Amazon EventBridge rules and uses the same syntax as EventBridge. For more information, see [Amazon EventBridge event patterns](#) in the *Amazon EventBridge User Guide*.

The following is a summary of all the comparison operators available for Lambda event filtering. Note that the **Equals (ignore case)**, **Or (multiple fields)**, and **Ends with** EventBridge comparison operators aren't supported.

Comparison operator	Example	Rule syntax
Null	UserID is null	"UserID": [null]

Comparison operator	Example	Rule syntax
Empty	Last Name is empty	"LastName": [""]
Equals	Name is "Alice"	"Name": ["Alice"]
And	Location is "New York" and Day is "Monday"	"Location": ["New York"], "Day": ["Monday"]
Or	PaymentType is "Credit" or "Debit"	"PaymentType": ["Credit", "Debit"]
Not	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] }]
Exists	ProductName exists	"ProductName": [{ "exists": true }]
Does not exist	ProductName does not exist	"ProductName": [{ "exists": false }]
Begins with	Region is in the US	"Region": [{ "prefix": "us-" }]

Note

Like EventBridge, for strings, Lambda uses exact character-by-character matching without case-folding or any other string normalization. For numbers, Lambda also uses string representation. For example, 300, 300.0, and 3.0e2 are not considered equal.

Attaching filter criteria to an event source mapping (console)

Follow these steps to create a new event source mapping with filter criteria using the Lambda console.

To create a new event source mapping with filter criteria (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of a function to create an event source mapping for.
3. Under **Function overview**, choose **Add trigger**.
4. For **Trigger configuration**, choose a trigger type that supports event filtering. For a list of supported services, refer to the list at the beginning of this page.
5. Expand **Additional settings**.
6. Under **Filter criteria**, choose **Add**, and then define and enter your filters. For example, you can enter the following.

```
{ "Metadata" : [ 1, 2 ] }
```

This instructs Lambda to process only the records where field Metadata is equal to 1 or 2. You can continue to select **Add** to add more filters up to the maximum allowed number.

-
- When you have finished adding your filters, choose **Save**.

When you enter filter criteria using the console, you enter only the filter pattern and don't need to provide the Pattern key or escape quotes. In step 6 of the preceding instructions, `{ "Metadata" : [1, 2] }` corresponds to the following FilterCriteria.

```
{  
  "Filters": [  
    {  
      "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"  
    }  
  ]  
}
```

After creating your event source mapping in the console, you can see the formatted FilterCriteria in the trigger details. For more examples of creating event filters using the console, see [Using filters with different AWS services \(p. 141\)](#).

Attaching filter criteria to an event source mapping (AWS CLI)

Suppose you want an event source mapping to have the following FilterCriteria:

```
{  
  "Filters": [  
    {  
      "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"  
    }  
  ]  
}
```

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
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 ] }"}]}'
```

This [create-event-source-mapping](#) command creates a new Amazon SQS event source mapping for function *my-function* with the specified FilterCriteria.

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \  
  --uuid "a1b2c3d4-5678-90ab-cdef-1111EXAMPLE" \  
  --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ] }"}]}'
```

Note that to update an event source mapping, you need its UUID. You can get the UUID from a [list-event-source-mappings](#) call. Lambda also returns the UUID in the [create-event-source-mapping](#) CLI response.

To remove filter criteria from an event source, you can run the following [update-event-source-mapping](#) command with an empty FilterCriteria object.

```
aws lambda update-event-source-mapping \
--uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
--filter-criteria "{}"
```

For more examples of creating event filters using the AWS CLI, see [Using filters with different AWS services \(p. 141\)](#).

Attaching filter criteria to an event source mapping (AWS SAM)

Suppose you want to configure an event source in AWS SAM to use the following filter criteria:

```
{
  "Filters": [
    {
      "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
    }
  ]
}
```

To add these filter criteria to your event source mapping, insert the following snippet into the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{\"Metadata\": [1, 2]}'
```

For more information on creating and configuring an AWS SAM template for an event source mapping, see the [EventSource](#) section of the AWS SAM Developer Guide. For more examples of creating event filters using AWS SAM templates, see [Using filters with different AWS services \(p. 141\)](#).

Using filters with different AWS services

Different types of event source use different key values for their data fields. To filter on data properties, make sure that you use the correct key in your filter's pattern. The following table gives the filtering keys for each supported AWS service.

AWS service	Filtering key
DynamoDB	dynamodb
Kinesis	data
Amazon MQ	data
Amazon MSK	value
Self-managed Apache Kafka	value
Amazon SQS	body

The following sections give examples of filter patterns for different types of event source. They also provide definitions of supported incoming data formats and filter pattern body formats for each supported service.

Filtering with DynamoDB

Suppose you have a DynamoDB table with the primary key `CustomerName` and attributes `AccountManager` and `PaymentTerms`. The following shows an example record from your DynamoDB table's stream.

```
{
  "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_AND_OLD_IMAGES"
      }
    }
  }
}
```

To filter based on the key and attribute values in your DynamoDB table, use the `dynamodb` key in the record. Suppose you want your function to process only those records where the primary key `CustomerName` is "AnyCompany Industries." the `FilterCriteria` object would be as follows.

```
{
  "Filters": [
    {
      "Pattern": "{ \"dynamodb\" : { \"Keys\" : { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"
    }
  ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
  "dynamodb": {
    "Keys": {
      "CustomerName": {
        "S": [ "AnyCompany Industries" ]
      }
    }
  }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping \(console\) \(p. 139\)](#) and enter the following string for the **Filter criteria**.

```
{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany Industries" ] } } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
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\" ] } } }"}}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
--uuid "a1b2c3d4-5678-90ab-caef-11111EXAMPLE" \
--filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" : { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } }"}}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany Industries" ] } } }'
```

With DynamoDB, you can also use the `NewImage` and `OldImage` keys to filter for attribute values. Suppose you want to filter records where the `AccountManager` attribute in the latest table image is "Pat Candella" or "Shirley Rodriguez." The `FilterCriteria` object would be as follows.

```
{
  "Filters": [
    {
      "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez\" ] } } }}"
    }
  ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
  "dynamodb": {
    "NewImage": {
      "AccountManager": {
        "S": [ "Pat Candella", "Shirley Rodriguez" ]
      }
    }
  }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping \(console\) \(p. 139\)](#) and enter the following string for the **Filter criteria**.

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella", "Shirley Rodriguez" ] } } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
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\" ] } } } }"}]}
```

To add these filter criteria to an existing event source mapping, run the following command.

```
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

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella", "Shirley Rodriguez" ] } } } }'
```

Note

DynamoDB event filtering doesn't support the use of numeric operators (numeric equals and numeric range). Even if items in your table are stored as numbers, these parameters are converted to strings in the JSON record object.

To properly filter events from DynamoDB sources, both the data field and your filter criteria for the data field (dynamodb) must be in valid JSON format. If either field isn't in a valid JSON format, Lambda drops the message or throws an exception. The following table summarizes the specific behavior:

Incoming data format	Filter pattern format for data properties	Resulting action
Valid JSON	Valid JSON	Lambda filters based on your filter criteria.
Valid JSON	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.

Incoming data format	Filter pattern format for data properties	Resulting action
Valid JSON	Non-JSON	Lambda throws an exception at the time of the event source mapping creation or update. The filter pattern for data properties must be in a valid JSON format.
Non-JSON	Valid JSON	Lambda drops the record.
Non-JSON	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Non-JSON	Non-JSON	Lambda throws an exception at the time of the event source mapping creation or update. The filter pattern for data properties must be in a valid JSON format.

Filtering with Kinesis

Suppose a producer is putting JSON formatted data into your Kinesis data stream. An example record would look like the following, with the JSON data converted to a Base64 encoded string in the data field.

```
{
  "kinesis": {
    "kinesisSchemaVersion": "1.0",
    "partitionKey": "1",
    "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
    "data": "eyJSZWNvcmROdW1iZXIiOiAiMDAwMSIsICJUaW1lU3RhXAiOiAieX15eS1tbS1kZFRoaDptbTpzcyIsICJSXF1ZXN0Q29kZSI6IyMjCg==",
    "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"
}
```

As long as the data the producer puts into the stream is valid JSON, you can use event filtering to filter records using the data key. Suppose a producer is putting records into your Kinesis stream in the following JSON format.

```
{
  "record": 12345,
  "order": [
    {
      "type": "buy",
      "stock": "ANYCO",
      "quantity": 1000
    }
}
```

To filter only those records where the order type is "buy," the `FilterCriteria` object would be as follows.

```
{  
  "Filters": [  
    {  
      "Pattern": "{ \"data\": { \"order\": { \"type\": [ \"buy\" ] } } }"  
    }  
  ]  
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{  
  "data": {  
    "order": {  
      "type": [ "buy" ]  
    }  
  }  
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping \(console\) \(p. 139\)](#) and enter the following string for the **Filter criteria**.

```
{ "data" : { "order" : { "type" : [ "buy" ] } } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
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\" ] } } }\"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \  
  --uuid a1b2c3d4-5678-90ab-cdef-11111EXAMPLE \  
  --filter-criteria '{\"Filters\": [{\"Pattern\": \"{ \"data\": { \"order\": { \"type\": [ \"buy\" ] } } }\"}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:  
  Filters:  
    - Pattern: '{ \"data\" : { \"order\" : { \"type\" : [ \"buy\" ] } } }'
```

To properly filter events from Kinesis sources, both the data field and your filter criteria for the data field must be in valid JSON format. If either field isn't in a valid JSON format, Lambda drops the message or throws an exception. The following table summarizes the specific behavior:

Incoming data format	Filter pattern format for data properties	Resulting action
Valid JSON	Valid JSON	Lambda filters based on your filter criteria.
Valid JSON	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	Non-JSON	Lambda throws an exception at the time of the event source mapping creation or update. The filter pattern for data properties must be in a valid JSON format.
Non-JSON	Valid JSON	Lambda drops the record.
Non-JSON	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Non-JSON	Non-JSON	Lambda throws an exception at the time of the event source mapping creation or update. The filter pattern for data properties must be in a valid JSON format.

Filtering Kinesis aggregated records

With Kinesis, you can aggregate multiple records into a single Kinesis Data Streams record to increase your data throughput. Lambda can only apply filter criteria to aggregated records when you use Kinesis [enhanced fan-out](#). Filtering aggregated records with standard Kinesis isn't supported. When using enhanced fan-out, you configure a Kinesis dedicated-throughput consumer to act as the trigger for your Lambda function. Lambda then filters the aggregated records and passes only those records that meet your filter criteria.

To learn more about Kinesis record aggregation, refer to the [Aggregation](#) section on the Kinesis Producer Library (KPL) Key Concepts page. To Learn more about using Lambda with Kinesis enhanced fan-out, see [Increasing real-time stream processing performance with Amazon Kinesis Data Streams enhanced fan-out and AWS Lambda](#) on the AWS compute blog.

Filtering with Amazon MQ

Note

When you update an event source mapping for Amazon MQ with new filter criteria, it can take up to 15 minutes for Lambda to apply your changes. Before this period has elapsed, previous filter settings can still be in effect. This is true even when the status of the event source mapping displayed in the console has changed from Enabling to Enabled.

Suppose your Amazon MQ message queue contains messages either in valid JSON format or as plain strings. An example record would look like the following, with the data converted to a Base64 encoded string in the data field.

ActiveMQ

```
{  
    "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-  
west-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": "QUJD0kFBQUE=",  
    "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  
    },  
    "data": "QUJD0kFBQUE=",  
    "timestamp": 1598827811958  
}
```

```

    "type": null,
    "userId": "AIDACKCEVSQ6C2EXAMPLE",
    "appId": null,
    "clusterId": null,
    "bodySize": 80
    },
    "redelivered": false,
    "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUiFQ=="
}

```

For both Active MQ and Rabbit MQ brokers, you can use event filtering to filter records using the data key. Suppose your Amazon MQ queue contains messages in the following JSON format.

```
{
  "timeout": 0,
  "IPAddress": "203.0.113.254"
}
```

To filter only those records where the `timeout` field is greater than 0, the `FilterCriteria` object would be as follows.

```
{
  "Filters": [
    {
      "Pattern": "{ \"data\" : { \"timeout\" : [ { \"numeric\" : [ \">>, 0 ] } ] } }"
    }
  ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
  "data": {
    "timeout": [ { "numeric": [ ">", 0 ] } ]
  }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

to add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping \(console\) \(p. 139\)](#) and enter the following string for the **Filter criteria**.

```
{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
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 ] } ] }"}]}]
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
--uuid "a1b2c3d4-5678-90ab-cdef-1111EXAMPLE" \
--filter-criteria '[{"Filters": [{"Pattern": "{ \"data\": { \"timeout\" : [ { \"numeric\": [ \">\", 0 ] } ] }"}]}]'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ \"data\" : { \"timeout\" : [ { \"numeric\": [ \">>, 0 ] } ] } }'
```

With Amazon MQ, you can also filter records where the message is a plain string. Suppose you want to process only records where the message begins with "Result: ". The `FilterCriteria` object would look as follows.

```
{
  "Filters": [
    {
      "Pattern": "{ \"data\" : [ { \"prefix\": \"Result: \" } ] }"
    }
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
  "data": [
    {
      "prefix": "Result: "
    }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping \(console\) \(p. 139\)](#) and enter the following string for the **Filter criteria**.

```
{ "data" : [ { "prefix": "Result " } ] }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
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: \
\" } ] }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
--uuid a1b2c3d4-5678-90ab-cdef-1111EXAMPLE \
--filter-criteria '{"Filters": [{"Pattern": "{ \"data\": [ { \"prefix\": \"Result: \
\" } ] }"}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : [ { "prefix": "Result" } ] }'
```

Amazon MQ messages must be UTF-8 encoded strings, either plain strings or in JSON format. That's because Lambda decodes Amazon MQ byte arrays into UTF-8 before applying filter criteria. If your messages use another encoding, such as UTF-16 or ASCII, or if the message format doesn't match the FilterCriteria format, Lambda processes metadata filters only. The following table summarizes the specific behavior:

Incoming message format	Filter pattern format for message properties	Resulting action
Plain string	Plain string	Lambda filters based on your filter criteria.
Plain string	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Plain string	Valid JSON	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	Plain string	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	Valid JSON	Lambda filters based on your filter criteria.
Non-UTF-8 encoded string	JSON, plain string, or no pattern	Lambda filters (on the other metadata properties only) based on your filter criteria.

Filtering with Amazon MSK and self-managed Apache Kafka

Note

When you update an event source mapping for Amazon MSK or self-managed Apache Kafka with new filter criteria, it can take up to 15 minutes for Lambda to apply your changes. Before this period has elapsed, previous filter settings can still be in effect. This is true even when the status of the event source mapping displayed in the console has changed from Enabling to Enabled.

Suppose a producer is writing messages to a topic in your Amazon MSK or self-managed Apache Kafka cluster, either in valid JSON format or as plain strings. An example record would look like the following, with the message converted to a Base64 encoded string in the value field.

```
{  
    "mytopic-0": [  
        {  
            "topic": "mytopic",  
            "partition": 0,  
            "offset": 15,  
            "timestamp": 1545084650987,  
            "timestampType": "CREATE_TIME",  
            "value": "SGVsbG8sIHRoaXMgYSB0ZXN0Lg==",  
            "headers": []  
        }  
    ]  
}
```

Suppose your Apache Kafka producer is writing messages to your topic in the following JSON format.

```
{  
    "device_ID": "AB1234",  
    "session": {  
        "start_time": "yyyy-mm-ddThh:mm:ss",  
        "duration": 162  
    }  
}
```

You can use the value key to filter records. Suppose you wanted to filter only those records where device_ID begins with the letters AB. The FilterCriteria object would be as follows.

```
{  
    "Filters": [  
        {  
            "Pattern": "{ \"value\" : { \"device_ID\" : [ { \"prefix\" : \"AB\" } ] } }"  
        }  
    ]  
}
```

For added clarity, here is the value of the filter's Pattern expanded in plain JSON.

```
{  
    "value": {  
        "device_ID": [ { "prefix": "AB" } ]  
    }  
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping \(console\) \(p. 139\)](#) and enter the following string for the **Filter criteria**.

```
{ "value" : { "device_ID" : [ { "prefix": "AB" } ] } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
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\" } ] } }"}]}
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
--uuid a1b2c3d4-5678-90ab-cdef-11111EXAMPLE \
--filter-criteria '{"Filters": [{"Pattern": "{ \"value\": { \"device_ID\": [ { \"prefix\": \"AB\" } ] } }"}]}
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : { "device_ID" : [ { "prefix": "AB" } ] } }'
```

With Amazon MSK and self-managed Apache Kafka, you can also filter records where the message is a plain string. Suppose you want to ignore those messages where the string is "error". The **FilterCriteria** object would look as follows.

```
{
  "Filters": [
    {
      "Pattern": "{ \"value\" : [ { \"anything-but\": [ \"error\" ] } ] }"
    }
  ]
}
```

For added clarity, here is the value of the filter's **Pattern** expanded in plain JSON.

```
{
  "value": [
    {
      "anything-but": [ "error" ]
    }
  ]
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping \(console\) \(p. 139\)](#) and enter the following string for the **Filter criteria**.

```
{ "value" : [ { "anything-but": [ "error" ] } ] }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
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\" ] } ] }"}]}
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
--uuid a1b2c3d4-5678-90ab-cdef-1111EXAMPLE \
--filter-criteria '{"Filters": [{"Pattern": "{ \"value\": [ { \"anything-but\": [ \"error\" ] } ] }"}]}
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : [ { "anything-but": [ "error" ] } ] }'
```

Amazon MSK and self-managed Apache Kafka messages must be UTF-8 encoded strings, either plain strings or in JSON format. That's because Lambda decodes Amazon MSK byte arrays into UTF-8 before applying filter criteria. If your messages use another encoding, such as UTF-16 or ASCII, or if the message format doesn't match the FilterCriteria format, Lambda processes metadata filters only. The following table summarizes the specific behavior:

Incoming message format	Filter pattern format for message properties	Resulting action
Plain string	Plain string	Lambda filters based on your filter criteria.
Plain string	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Plain string	Valid JSON	Lambda filters (on the other metadata properties only) based on your filter criteria.

Incoming message format	Filter pattern format for message properties	Resulting action
Valid JSON	Plain string	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	Valid JSON	Lambda filters based on your filter criteria.
Non-UTF-8 encoded string	JSON, plain string, or no pattern	Lambda filters (on the other metadata properties only) based on your filter criteria.

Filtering with Amazon SQS

Suppose your Amazon SQS queue contains messages in the following JSON format.

```
{
    "RecordNumber": 0000,
    "TimeStamp": "yyyy-mm-ddThh:mm:ss",
    "RequestCode": "AAAA"
}
```

An example record for this queue would look as follows.

```
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxLaS3SLy0a...",
    "body": "{\n        \"RecordNumber\": 0000,\n        \"TimeStamp\": \"yyyy-mm-ddThh:mm:ss\",\\n\n        \"RequestCode\": \"AAAA\"\n    }",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIEQQZJ0L023VYJ4V0",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
    },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-west-2:123456789012:my-queue",
    "awsRegion": "us-west-2"
}
```

To filter based on the contents of your Amazon SQS messages, use the body key in the Amazon SQS message record. Suppose you want to process only those records where the RequestCode in your Amazon SQS message is "BBBB." The FilterCriteria object would be as follows.

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RequestCode\" : [ \"BBBB\" ] } }"
        }
    ]
}
```

```
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
  "body": {
    "RequestCode": [ "BBBB" ]
  }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping \(console\) \(p. 139\)](#) and enter the following string for the **Filter criteria**.

```
{ "body" : { "RequestCode" : [ "BBBB" ] } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
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\" ] } }"}]}
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
--uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
--filter-criteria '{"Filters": [{"Pattern": "{ \"body\": { \"RequestCode\": [ \"BBBB\" ] } }"}]}
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RequestCode" : [ "BBBB" ] } }'
```

Suppose you want your function to process only those records where `RecordNumber` is greater than 9999. The `FilterCriteria` object would be as follows.

```
{
  "Filters": [
    {
      "Pattern": "{ \"body\": { \"RecordNumber\": [ { \"numeric\": [ \">>\", 9999 ] } ] } }"
    }
]
```

```
}
```

For added clarity, here is the value of the filter's Pattern expanded in plain JSON.

```
{
  "body": {
    "RecordNumber": [
      {
        "numeric": [ ">", 9999 ]
      }
    ]
  }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in [Attaching filter criteria to an event source mapping \(console\) \(p. 139\)](#) and enter the following string for the **Filter criteria**.

```
{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
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 ] } ] }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
--uuid a1b2c3d4-5678-90ab-cdef-11111EXAMPLE \
--filter-criteria '{"Filters": [{"Pattern": "{ \"body\": { \"RecordNumber\": [ { \"numeric\": [ \">\", 9999 ] } ] }"}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ \"body\" : { \"RecordNumber\" : [ { \"numeric\": [ \">\", 9999 ] } ] } }'
```

For Amazon SQS, the message body can be any string. However, this can be problematic if your FilterCriteria expect body to be in a valid JSON format. The reverse scenario is also true—if the incoming message body is in JSON format but your filter criteria expects body to be a plain string, this can lead to unintended behavior.

To avoid this issue, ensure that the format of body in your FilterCriteria matches the expected format of body in messages that you receive from your queue. Before filtering your messages, Lambda

automatically evaluates the format of the incoming message body and of your filter pattern for body. If there is a mismatch, Lambda drops the message. The following table summarizes this evaluation:

Incoming message body format	Filter pattern body format	Resulting action
Plain string	Plain string	Lambda filters based on your filter criteria.
Plain string	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Plain string	Valid JSON	Lambda drops the message.
Valid JSON	Plain string	Lambda drops the message.
Valid JSON	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	Valid JSON	Lambda filters based on your filter criteria.

Lambda function states

Lambda includes a state field in the function configuration for all functions to indicate when your function is ready to invoke. State provides information about the current status of the function, including whether you can successfully invoke the function. Function states do not change the behavior of function invocations or how your function runs the code. Function states include:

- Pending – After Lambda creates the function, it sets the state to pending. While in pending state, Lambda attempts to create or configure resources for the function, such as VPC or EFS resources. Lambda does not invoke a function during pending state. Any invocations or other API actions that operate on the function will fail.
- Active – Your function transitions to active state after Lambda completes resource configuration and provisioning. Functions can only be successfully invoked while active.
- Failed – Indicates that resource configuration or provisioning encountered an error.
- Inactive – A function becomes inactive when it has been idle long enough for Lambda to reclaim the external resources that were configured for it. When you try to invoke a function that is inactive, the invocation fails and Lambda sets the function to pending state until the function resources are recreated. If Lambda fails to recreate the resources, the function is set to the inactive state.

If you are using SDK-based automation workflows or calling Lambda's service APIs directly, ensure that you check a function's state before invocation to verify that it is active. You can do this with the Lambda API action [GetFunction \(p. 1220\)](#), or by configuring a waiter using the [AWS SDK for Java 2.0](#).

```
aws lambda get-function --function-name my-function --query 'Configuration.[State, LastUpdateStatus]'
```

You should see the following output:

```
[  
  "Active",  
  "Successful"  
]
```

Functions have two other attributes, StateReason and StateReasonCode. These provide information and context about the function's state when it is not active for troubleshooting issues.

The following operations fail while function creation is pending:

- [Invoke \(p. 1260\)](#)
- [UpdateFunctionCode \(p. 1367\)](#)
- [UpdateFunctionConfiguration \(p. 1377\)](#)
- [PublishVersion \(p. 1315\)](#)

Function states while updating

Lambda provides additional context for functions undergoing updates with the LastUpdateStatus attribute, which can have the following statuses:

- InProgress – An update is happening on an existing function. While a function update is in progress, invocations go to the function's previous code and configuration.
- Successful – The update has completed. Once Lambda finishes the update, this stays set until a further update.

- Failed – The function update has failed. Lambda aborts the update and the function's previous code and configuration remain available.

Example

The following is the result of `get-function-configuration` on a function undergoing an update.

```
{  
    "FunctionName": "my-function",  
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",  
    "Runtime": "nodejs18.x",  
    "VpcConfig": {  
        "SubnetIds": [  
            "subnet-071f712345678e7c8",  
            "subnet-07fd123456788a036",  
            "subnet-0804f77612345cacf"  
        ],  
        "SecurityGroupIds": [  
            "sg-085912345678492fb"  
        ],  
        "VpcId": "vpc-08e1234569e011e83"  
    },  
    "State": "Active",  
    "LastUpdateStatus": "InProgress",  
    ...  
}
```

[FunctionConfiguration \(p. 1430\)](#) has two other attributes, `LastUpdateStatusReason` and `LastUpdateStatusReasonCode`, to help troubleshoot issues with updating.

The following operations fail while an asynchronous update is in progress:

- [UpdateFunctionCode \(p. 1367\)](#)
- [UpdateFunctionConfiguration \(p. 1377\)](#)
- [PublishVersion \(p. 1315\)](#)
- [TagResource \(p. 1345\)](#)

Error handling and automatic retries in AWS Lambda

When you invoke a function, two types of error can occur. Invocation errors occur when the invocation request is rejected before your function receives it. Function errors occur when your function's code or [runtime \(p. 37\)](#) returns an error. Depending on the type of error, the type of invocation, and the client or service that invokes the function, the retry behavior and the strategy for managing errors varies.

Issues with the request, caller, or account can cause invocation errors. Invocation errors include an error type and status code in the response that indicate the cause of the error.

Common invocation errors

- **Request** – The request event is too large or isn't valid JSON, the function doesn't exist, or a parameter value is the wrong type.
- **Caller** – The user or service doesn't have permission to invoke the function.
- **Account** – The maximum number of function instances are already running, or requests are being made too quickly.

Clients such as the AWS CLI and the AWS SDK retry on client timeouts, throttling errors (429), and other errors that aren't caused by a bad request. For a full list of invocation errors, see [Invoke \(p. 1260\)](#).

Function errors occur when your function code or the runtime that it uses return an error.

Common function errors

- **Function** – Your function's code throws an exception or returns an error object.
- **Runtime** – The runtime terminated your function because it ran out of time, detected a syntax error, or failed to marshal the response object into JSON. The function exited with an error code.

Unlike invocation errors, function errors don't cause Lambda to return a 400-series or 500-series status code. If the function returns an error, Lambda indicates this by including a header named X-Amz-Function-Error, and a JSON-formatted response with the error message and other details. For examples of function errors in each language, see the following topics.

- [AWS Lambda function errors in Node.js \(p. 278\)](#)
- [AWS Lambda function errors in Python \(p. 346\)](#)
- [AWS Lambda function errors in Ruby \(p. 377\)](#)
- [AWS Lambda function errors in Java \(p. 429\)](#)
- [AWS Lambda function errors in Go \(p. 478\)](#)
- [AWS Lambda function errors in C# \(p. 514\)](#)
- [AWS Lambda function errors in PowerShell \(p. 539\)](#)

When you invoke a function directly, you determine the strategy for handling errors related to your function code. Lambda does not automatically retry these types of errors on your behalf. To retry, you can manually re-invoke your function, send the failed event to a queue for debugging, or ignore the error. Your function's code might have run completely, partially, or not at all. If you retry, ensure that your function's code can handle the same event multiple times without causing duplicate transactions or other unwanted side effects.

When you invoke a function indirectly, you need to be aware of the retry behavior of the invoker and any service that the request encounters along the way. This includes the following scenarios.

- **Asynchronous invocation** – Lambda retries function errors twice. If the function doesn't have enough capacity to handle all incoming requests, events might wait in the queue for hours or days to be sent to the function. You can configure a dead-letter queue on the function to capture events that weren't successfully processed. For more information, see [Asynchronous invocation \(p. 123\)](#).
- **Event source mappings** – Event source mappings that read from streams retry the entire batch of items. Repeated errors block processing of the affected shard until the error is resolved or the items expire. To detect stalled shards, you can monitor the [Iterator Age \(p. 870\)](#) metric.

For event source mappings that read from a queue, you determine the length of time between retries and destination for failed events by configuring the visibility timeout and redrive policy on the source queue. For more information, see [Lambda event source mappings \(p. 131\)](#) and the service-specific topics under [Using AWS Lambda with other services \(p. 556\)](#).

- **AWS services** – AWS services can invoke your function [synchronously \(p. 120\)](#) or asynchronously. For synchronous invocation, the service decides whether to retry. For example, Amazon S3 batch operations retries the operation if the Lambda function returns a `TemporaryFailure` response code. Services that proxy requests from an upstream user or client may have a retry strategy or may relay the error response back to the requestor. For example, API Gateway always relays the error response back to the requestor.

For asynchronous invocation, the behavior is the same as when you invoke the function synchronously. For more information, see the service-specific topics under [Using AWS Lambda with other services \(p. 556\)](#) and the invoking service's documentation.

- **Other accounts and clients** – When you grant access to other accounts, you can use [resource-based policies \(p. 832\)](#) to restrict the services or resources they can configure to invoke your function. To protect your function from being overloaded, consider putting an API layer in front of your function with [Amazon API Gateway \(p. 562\)](#).

To help you deal with errors in Lambda applications, Lambda integrates with services like Amazon CloudWatch and AWS X-Ray. You can use a combination of logs, metrics, alarms, and tracing to quickly detect and identify issues in your function code, API, or other resources that support your application. For more information, see [Monitoring and troubleshooting Lambda functions \(p. 860\)](#).

For a sample application that uses a CloudWatch Logs subscription, X-Ray tracing, and a Lambda function to detect and process errors, see [Error processor sample application for AWS Lambda \(p. 1015\)](#).

Testing Lambda functions in the console

You can test your Lambda function in the console by invoking your function with a *test event*. A *test event* is a JSON input to your function. If your function doesn't require input, the event can be an empty document ({}).

Private test events

Private test events are available only to the event creator, and they require no additional permissions to use. You can create and save up to 10 private test events per function.

To create a private test event

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of the function that you want to test.
3. Choose the **Test** tab.
4. Under **Test event**, do the following:
 - a. Choose a **Template**.
 - b. Enter a **Name** for the test.
 - c. In the text entry box, enter the JSON test event.
 - d. Under **Event sharing settings**, choose **Private**.
5. Choose **Save changes**.

You can also create new test events on the **Code** tab. From there, choose **Test, Configure test event**.

Shareable test events

Shareable test events are test events that you can share with other users in the same AWS account. You can edit other users' shareable test events and invoke your function with them.

Lambda saves shareable test events as schemas in an [Amazon EventBridge \(CloudWatch Events\) schema registry](#) named lambda-testevent-schemas. As Lambda utilizes this registry to store and call shareable test events you create, we recommend that you do not edit this registry or create a registry using the lambda-testevent-schemas name.

To see, share, and edit shareable test events, you must have permissions for all of the following [EventBridge \(CloudWatch Events\) schema registry API operations](#):

- [schemas.CreateRegistry](#)
- [schemas.CreateSchema](#)
- [schemas.DeleteSchema](#)
- [schemas.DeleteSchemaVersion](#)
- [schemas.DescribeRegistry](#)
- [schemas.DescribeSchema](#)
- [schemas.GetDiscoveredSchema](#)
- [schemas.ListSchemaVersions](#)
- [schemas.UpdateSchema](#)

Note that saving edits made to a shareable test event overwrites that event.

If you cannot create, edit, or see shareable test events, check that your account has the required permissions for these operations. If you have the required permissions but still cannot access shareable test events, check for any [resource-based policies \(p. 832\)](#) that might limit access to the EventBridge (CloudWatch Events) registry.

To create a shareable test event

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of the function that you want to test.
3. Choose the **Test** tab.
4. Under **Test event**, do the following:
 - a. Choose a **Template**.
 - b. Enter a **Name** for the test.
 - c. In the text entry box, enter the JSON test event.
 - d. Under **Event sharing settings**, choose **Shareable**.
5. Choose **Save changes**.

Invoking functions with test events

When you run a test event in the console, Lambda synchronously invokes your function with the test event. The function runtime converts the JSON document into an object and passes it to your code's handler method for processing.

To test a function

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of the function that you want to test.
3. Choose the **Test** tab.
4. Under **Test event**, choose **Saved event**, and then choose the saved event that you want to use.
5. Choose **Test**.
6. To review the test results, under **Execution result**, expand **Details**.

To invoke your function without saving your test event, choose **Test** before saving. This creates an unsaved test event that Lambda preserves for the duration of the session.

You can also access your saved and unsaved test events on the **Code** tab. From there, choose **Test**, and then choose your test event.

Deleting shareable test event schemas

When you delete shareable test events, Lambda removes them from the `lambda-testevent-schemas` registry. If you remove the last shareable test event from the registry, Lambda deletes the registry.

If you delete the function, Lambda does not delete any associated shareable test event schemas. You must clean up these resources manually from the [EventBridge \(CloudWatch Events\) console](#).

Invoking functions defined as container images

For a Lambda function defined as a container image, function behavior during invocation is very similar to a function defined as a .zip file archive. The following sections highlight the similarities and differences.

Topics

- [Function lifecycle \(p. 165\)](#)
- [Invoking the function \(p. 165\)](#)
- [Image security \(p. 165\)](#)

Function lifecycle

After you upload a new or updated container image, Lambda optimizes the image before the function can process invocations. The optimization process can take a few seconds. The function remains in the Pending state until the process completes. The function then transitions to the Active state. While the state is Pending, you can invoke the function, but other operations on the function fail. Invocations that occur while an image update is in progress run the code from the previous image.

If a function is not invoked for multiple weeks, Lambda reclaims its optimized version, and the function transitions to the Inactive state. To reactivate the function, you must invoke it. Lambda rejects the first invocation and the function enters the Pending state until Lambda re-optimizes the image. The function then returns to the Active state.

Lambda periodically fetches the associated container image from the Amazon Elastic Container Registry (Amazon ECR) repository. If the corresponding container image no longer exists on Amazon ECR or permissions are revoked, the function enters the Failed state, and Lambda returns a failure for any function invocations.

You can use the Lambda API to get information about a function's state. For more information, see [Lambda function states \(p. 159\)](#).

Invoking the function

When you invoke the function, Lambda deploys the container image to an execution environment. Lambda initializes any [extensions \(p. 896\)](#) and then runs the function's initialization code (the code outside the main handler). Note that function initialization duration is included in billed execution time.

Lambda then runs the function by calling the code entry point specified in the function configuration (the ENTRYPPOINT and CMD [container image settings \(p. 888\)](#)).

Image security

When Lambda first downloads the container image from its original source (Amazon ECR), the container image is optimized, encrypted, and stored using authenticated convergent encryption methods. All keys that are required to decrypt customer data are protected using AWS KMS customer managed keys. To track and audit Lambda's usage of customer managed keys, you can view the [AWS CloudTrail logs \(p. 585\)](#).

Lambda function URLs

A function URL is a dedicated HTTP(S) endpoint for your Lambda function. You can create and configure a function URL through the Lambda console or the Lambda API. When you create a function URL, Lambda automatically generates a unique URL endpoint for you. Once you create a function URL, its URL endpoint never changes. Function URL endpoints have the following format:

```
https://<url-id>.lambda-url.<region>.on.aws
```

Function URLs are dual stack-enabled, supporting IPv4 and IPv6. After you configure a function URL for your function, you can invoke your function through its HTTP(S) endpoint via a web browser, curl, Postman, or any HTTP client.

Note

You can access your function URL through the public Internet only. While Lambda functions do support AWS PrivateLink, function URLs do not.

Lambda function URLs use [resource-based policies \(p. 832\)](#) for security and access control. Function URLs also support cross-origin resource sharing (CORS) configuration options.

You can apply function URLs to any function alias, or to the \$LATEST unpublished function version. You can't add a function URL to any other function version.

Topics

- [Creating and managing Lambda function URLs \(p. 167\)](#)
- [Security and auth model for Lambda function URLs \(p. 172\)](#)
- [Invoking Lambda function URLs \(p. 177\)](#)
- [Monitoring Lambda function URLs \(p. 184\)](#)
- [Tutorial: Creating a Lambda function with a function URL \(p. 186\)](#)

Creating and managing Lambda function URLs

A function URL is a dedicated HTTP(S) endpoint for your Lambda function. You can create and configure a function URL through the Lambda console or the Lambda API. When you create a function URL, Lambda automatically generates a unique URL endpoint for you. Once you create a function URL, its URL endpoint never changes. Function URL endpoints have the following format:

```
https://<url-id>.lambda-url.<region>.on.aws
```

The following section show how to create and manage a function URL using the Lambda console, AWS CLI, and AWS CloudFormation template

Topics

- [Creating a function URL \(console\) \(p. 167\)](#)
- [Creating a function URL \(AWS CLI\) \(p. 168\)](#)
- [Adding a function URL to a CloudFormation template \(p. 169\)](#)
- [Cross-origin resource sharing \(CORS\) \(p. 170\)](#)
- [Throttling function URLs \(p. 170\)](#)
- [Deactivating function URLs \(p. 171\)](#)
- [Deleting function URLs \(p. 171\)](#)

Creating a function URL (console)

Follow these steps to create a function URL using the console.

To create a function URL for an existing function (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of the function that you want to create the function URL for.
3. Choose the **Configuration** tab, and then choose **Function URL**.
4. Choose **Create function URL**.
5. For **Auth type**, choose **AWS_IAM** or **NONE**. For more information about function URL authentication, see [Security and auth model \(p. 172\)](#).
6. (Optional) Select **Configure cross-origin resource sharing (CORS)**, and then configure the CORS settings for your function URL. For more information about CORS, see [Cross-origin resource sharing \(CORS\) \(p. 170\)](#).
7. Choose **Save**.

This creates a function URL for the \$LATEST unpublished version of your function. The function URL appears in the **Function overview** section of the console.

To create a function URL for an existing alias (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of the function with the alias that you want to create the function URL for.
3. Choose the **Aliases** tab, and then choose the name of the alias that you want to create the function URL for.
4. Choose the **Configuration** tab, and then choose **Function URL**.

5. Choose **Create function URL**.
6. For **Auth type**, choose **AWS_IAM** or **NONE**. For more information about function URL authentication, see [Security and auth model \(p. 172\)](#).
7. (Optional) Select **Configure cross-origin resource sharing (CORS)**, and then configure the CORS settings for your function URL. For more information about CORS, see [Cross-origin resource sharing \(CORS\) \(p. 170\)](#).
8. Choose **Save**.

This creates a function URL for your function alias. The function URL appears in the console's **Function overview** section for your alias.

To create a new function with a function URL (console)

To create a new function with a function URL (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose **Create function**.
3. Under **Basic information**, do the following:
 - a. For **Function name**, enter a name for your function, such as **my-function**.
 - b. For **Runtime**, choose the language runtime that you prefer, such as **Node.js 18.x**.
 - c. For **Architecture**, choose either **x86_64** or **arm64**.
 - d. Expand **Permissions**, then choose whether to create a new execution role or use an existing one.
4. Expand **Advanced settings**, and then select **Function URL**.
5. For **Auth type**, choose **AWS_IAM** or **NONE**. For more information about function URL authentication, see [Security and auth model \(p. 172\)](#).
6. (Optional) Select **Configure cross-origin resource sharing (CORS)**. By selecting this option during function creation, your function URL allows requests from all origins by default. You can edit the CORS settings for your function URL after creating the function. For more information about CORS, see [Cross-origin resource sharing \(CORS\) \(p. 170\)](#).
7. Choose **Create function**.

This creates a new function with a function URL for the \$LATEST unpublished version of the function. The function URL appears in the **Function overview** section of the console.

Creating a function URL (AWS CLI)

To create a function URL for an existing Lambda function using the AWS Command Line Interface (AWS CLI), run the following command:

```
aws lambda create-function-url-config \
--function-name my-function \
--qualifier prod \ // optional
--auth-type AWS_IAM
--cors-config {AllowOrigins="https://example.com"} // optional
```

This adds a function URL to the **prod** qualifier for the function **my-function**. For more information about these configuration parameters, see [CreateFunctionUrlConfig](#) in the API reference.

Note

To create a function URL via the AWS CLI, the function must already exist.

Adding a function URL to a CloudFormation template

To add an AWS::Lambda::Url resource to your AWS CloudFormation template, use the following syntax:

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

- (Required) AuthType – Defines the type of authentication for your function URL. Possible values are either AWS_IAM or NONE. To restrict access to authenticated users only, set to AWS_IAM. To bypass IAM authentication and allow any user to make requests to your function, set to NONE.
- (Optional) Cors – Defines the [CORS settings \(p. 170\)](#) for your function URL. To add Cors to your AWS::Lambda::Url resource in CloudFormation, use the following syntax.

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 – The alias name.
- (Required) TargetFunctionArn – The name or Amazon Resource Name (ARN) of the Lambda function. Valid name formats include the following:
 - **Function name** – my-function
 - **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function
 - **Partial ARN** – 123456789012:function:my-function

Cross-origin resource sharing (CORS)

To define how different origins can access your function URL, use [cross-origin resource sharing \(CORS\)](#). We recommend configuring CORS if you intend to call your function URL from a different domain. Lambda supports the following CORS headers for function URLs.

CORS header	CORS configuration property	Example values
Access-Control-Allow-Origin	AllowOrigins	* (allow all origins) 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 (default), 300

When you configure CORS for a function URL using the Lambda console or the AWS CLI, Lambda automatically adds the CORS headers to all responses through the function URL. Alternatively, you can manually add CORS headers to your function response. If there are conflicting headers, the configured CORS headers on the function URL take precedence.

Throttling function URLs

Throttling limits the rate at which your function processes requests. This is useful in many situations, such as preventing your function from overloading downstream resources, or handling a sudden surge in requests.

You can throttle the rate of requests that your Lambda function processes through a function URL by configuring reserved concurrency. Reserved concurrency limits the number of maximum concurrent invocations for your function. Your function's maximum request rate per second (RPS) is equivalent to 10 times the configured reserved concurrency. For example, if you configure your function with a reserved concurrency of 100, then the maximum RPS is 1,000.

Whenever your function concurrency exceeds the reserved concurrency, your function URL returns an HTTP 429 status code. If your function receives a request that exceeds the 10x RPS maximum based on

your configured reserved concurrency, you also receive an HTTP 429 error. For more information about reserved concurrency, see [Configuring reserved concurrency \(p. 210\)](#).

Deactivating function URLs

In an emergency, you might want to reject all traffic to your function URL. To deactivate your function URL, set the reserved concurrency to zero. This throttles all requests to your function URL, resulting in HTTP 429 status responses. To reactivate your function URL, delete the reserved concurrency configuration, or set the configuration to an amount greater than zero.

Deleting function URLs

When you delete a function URL, you can't recover it. Creating a new function URL will result in a different URL address.

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of the function.
3. Choose the **Configuration** tab, and then choose **Function URL**.
4. Choose **Delete**.
5. Enter the word *delete* into the field to confirm the deletion.
6. Choose **Delete**.

Note

If you delete a function URL with auth type NONE, Lambda doesn't automatically delete the associated resource-based policy. If you want to delete this policy, you must manually do so.

Security and auth model for Lambda function URLs

You can control access to your Lambda function URLs using the AuthType parameter combined with [resource-based policies \(p. 832\)](#) attached to your specific function. The configuration of these two components determines who can invoke or perform other administrative actions on your function URL.

The AuthType parameter determines how Lambda authenticates or authorizes requests to your function URL. When you configure your function URL, you must specify one of the following AuthType options:

- AWS_IAM – Lambda uses AWS Identity and Access Management (IAM) to authenticate and authorize requests based on the IAM principal's identity policy and the function's resource-based policy. Choose this option if you want only authenticated users and roles to invoke your function via the function URL.
- NONE – Lambda doesn't perform any authentication before invoking your function. However, your function's resource-based policy is always in effect and must grant public access before your function URL can receive requests. Choose this option to allow public, unauthenticated access to your function URL.

In addition to AuthType, you can also use resource-based policies to grant permissions to other AWS accounts to invoke your function. For more information, see [Using resource-based policies for Lambda \(p. 832\)](#).

For additional insights into security, you can use AWS Identity and Access Management Access Analyzer to get a comprehensive analysis of external access to your function URL. IAM Access Analyzer also monitors for new or updated permissions on your Lambda functions to help you identify permissions that grant public and cross-account access. IAM Access Analyzer is free to use for any AWS customer. To get started with IAM Access Analyzer, see [Using AWS IAM Access Analyzer](#).

This page contains examples of resource-based policies for both auth types, and also how to create these policies using the [AddPermission \(p. 1141\)](#) API operation or the Lambda console. For information on how to invoke your function URL after you've set up permissions, see [Invoking Lambda function URLs \(p. 177\)](#).

Topics

- [Using the AWS_IAM auth type \(p. 172\)](#)
- [Using the NONE auth type \(p. 174\)](#)
- [Governance and access control \(p. 174\)](#)

Using the AWS_IAM auth type

If you choose the AWS_IAM auth type, users who need to invoke your Lambda function URL must have the `lambda:InvokeFunctionUrl` permission. Depending on who makes the invocation request, you may have to grant this permission using a resource-based policy.

If the principal making the request is in the same AWS account as the function URL, then the principal must **either** have `lambda:InvokeFunctionUrl` permissions in their [identity-based policy](#), **or** have permissions granted to them in the function's resource-based policy. In other words, a resource-based policy is optional if the user already has `lambda:InvokeFunctionUrl` permissions in their identity-based policy. Policy evaluation follows the rules outlined in [Determining whether a request is allowed or denied within an account](#).

If the principal making the request is in a different account, then the principal must have **both** an identity-based policy that gives them `lambda:InvokeFunctionUrl` permissions **and** permissions

granted to them in a resource-based policy on the function that they are trying to invoke. In these cross-account cases, policy evaluation follows the rules outlined in [Determining whether a cross-account request is allowed](#).

For an example cross-account interaction, the following resource-based policy allows the example role in AWS account 444455556666 to invoke the function URL associated with function my-function:

Example function URL cross-account invoke policy

```
{
    "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"
                }
            }
        }
    ]
}
```

You can create this policy statement through the console by following these steps:

To grant URL invocation permissions to another account (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of the function that you want to grant URL invocation permissions for.
3. Choose the **Configuration** tab, and then choose **Permissions**.
4. Under **Resource-based policy**, choose **Add permissions**.
5. Choose **Function URL**.
6. For **Auth type**, choose **AWS_IAM**.
7. (Optional) For **Statement ID**, enter a statement ID for your policy statement.
8. For **Principal**, enter the Amazon Resource Name (ARN) of the user or role that you want to grant permissions to. For example: `arn:aws:iam::444455556666:role/example`.
9. Choose **Save**.

Alternatively, you can create this policy statement using the following [add-permission](#) AWS Command Line Interface (AWS CLI) command:

```
aws lambda add-permission --function-name my-function \
--statement-id example0-cross-account-statement \
--action lambda:InvokeFunctionUrl \
--principal arn:aws:iam::444455556666:role/example \
--function-url-auth-type AWS_IAM
```

In the previous example, the `lambda:FunctionUrlAuthType` condition key value is `AWS_IAM`. This policy only allows access when your function URL's auth type is also `AWS_IAM`.

Using the NONE auth type

Important

When your function URL auth type is NONE and you have a resource-based policy that grants public access, any unauthenticated user with your function URL can invoke your function.

In some cases, you may want your function URL to be public. For example, you might want to serve requests made directly from a web browser. To allow public access to your function URL, choose the NONE auth type.

If you choose the NONE auth type, Lambda doesn't use IAM to authenticate requests to your function URL. However, users must still have `lambda:InvokeFunctionUrl` permissions in order to successfully invoke your function URL. You can grant `lambda:InvokeFunctionUrl` permissions using the following resource-based policy:

Example function URL invoke policy for all unauthenticated principals

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Principal": "*",  
            "Action": "lambda:InvokeFunctionUrl",  
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",  
            "Condition": {  
                "StringEquals": {  
                    "lambda:FunctionUrlAuthType": "NONE"  
                }  
            }  
        }  
    ]  
}
```

Note

When you create a function URL with auth type NONE via the console or AWS Serverless Application Model (AWS SAM), Lambda automatically creates the preceding resource-based policy statement for you. If the policy already exists, or the user or role creating the application doesn't have the appropriate permissions, then Lambda won't create it for you. If you're using the AWS CLI, AWS CloudFormation, or the Lambda API directly, you must add `lambda:InvokeFunctionUrl` permissions yourself. This makes your function public.

In addition, if you delete your function URL with auth type NONE, Lambda doesn't automatically delete the associated resource-based policy. If you want to delete this policy, you must manually do so.

In this statement, the `lambda:FunctionUrlAuthType` condition key value is NONE. This policy statement allows access only when your function URL's auth type is also NONE.

If a function's resource-based policy doesn't grant `lambda:InvokeFunctionUrl` permissions, then users will get a 403 Forbidden error code when they try to invoke your function URL, even if the function URL uses the NONE auth type.

Governance and access control

In addition to function URL invocation permissions, you can also control access on actions used to configure function URLs. Lambda supports the following IAM policy actions for function URLs:

- `lambda:InvokeFunctionUrl` – Invoke a Lambda function using the function URL.
- `lambda>CreateFunctionUrlConfig` – Create a function URL and set its AuthType.

- `lambda:UpdateFunctionUrlConfig` – Update a function URL configuration and its AuthType.
- `lambda:GetFunctionUrlConfig` – View the details of a function URL.
- `lambda>ListFunctionUrlConfigs` – List function URL configurations.
- `lambda>DeleteFunctionUrlConfig` – Delete a function URL.

Note

The Lambda console supports adding permissions only for `lambda:InvokeFunctionUrl`. For all other actions, you must add permissions using the Lambda API or AWS CLI.

To allow or deny function URL access to other AWS entities, include these actions in IAM policies. For example, the following policy grants the example role in AWS account 444455556666 permissions to update the function URL for function **my-function** in account 123456789012.

Example cross-account function URL policy

```
{
    "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"
        }
    ]
}
```

Condition keys

For fine-grained access control over your function URLs, use a condition key. Lambda supports one additional condition key for function URLs: `FunctionUrlAuthType`. The `FunctionUrlAuthType` key defines an enum value describing the auth type that your function URL uses. The value can be either `AWS_IAM` or `NONE`.

You can use this condition key in policies associated with your function. For example, you might want to restrict who can make configuration changes to your function URLs. To deny all `UpdateFunctionUrlConfig` requests to any function with URL auth type `NONE`, you can define the following policy:

Example function URL policy with explicit deny

```
{
    "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"
                }
            }
        }
    ]
}
```

```
    ]
}
```

To grant the example role in AWS account 444455556666 permissions to make CreateFunctionUrlConfig and UpdateFunctionUrlConfig requests on functions with URL auth type AWS_IAM, you can define the following policy:

Example function URL policy with explicit allow

```
{
  "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:*

```

You can also use this condition key in a [service control policy](#) (SCP). Use SCPs to manage permissions across an entire organization in AWS Organizations. For example, to deny users from creating or updating function URLs that use anything other than the AWS_IAM auth type, use the following service control policy:

Example function URL SCP with explicit deny

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "lambda>CreateFunctionUrlConfig",
        "lambda:UpdateFunctionUrlConfig"
      ],
      "Resource": "arn:aws:lambda:*:123456789012:function:*

```

Invoking Lambda function URLs

A function URL is a dedicated HTTP(S) endpoint for your Lambda function. You can create and configure a function URL through the Lambda console or the Lambda API. When you create a function URL, Lambda automatically generates a unique URL endpoint for you. Once you create a function URL, its URL endpoint never changes. Function URL endpoints have the following format:

```
https://<url-id>.lambda-url.<region>.on.aws
```

Function URLs are dual stack-enabled, supporting IPv4 and IPv6. After configuring your function URL, you can invoke your function through its HTTP(S) endpoint via a web browser, curl, Postman, or any HTTP client. To invoke a function URL, you must have `lambda:InvokeFunctionUrl` permissions. For more information, see [Security and auth model \(p. 172\)](#).

Topics

- [Function URL invocation basics \(p. 177\)](#)
- [Request and response payloads \(p. 177\)](#)

Function URL invocation basics

If your function URL uses the `AWS_IAM` auth type, you must sign each HTTP request using [AWS Signature Version 4 \(SigV4\)](#). Tools such as `awscurl`, [Postman](#), and [AWS SigV4 Proxy](#) offer built-in ways to sign your requests with SigV4.

If you don't use a tool to sign HTTP requests to your function URL, you must manually sign each request using SigV4. When your function URL receives a request, Lambda also calculates the SigV4 signature. Lambda processes the request only if the signatures match. For instructions on how to manually sign your requests with SigV4, see [Signing AWS requests with Signature Version 4](#) in the *Amazon Web Services General Reference Guide*.

If your function URL uses the `NONE` auth type, you don't have to sign your requests using SigV4. You can invoke your function using a web browser, curl, Postman, or any HTTP client.

To test simple GET requests to your function, use a web browser. For example, if your function URL is `https://abcdefg.lambda-url.us-east-1.on.aws`, and it takes in a string parameter `message`, your request URL could look like this:

```
https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld
```

To test other HTTP requests, such as a POST request, you can use a tool such as curl. For example, if you want to include some JSON data in a POST request to your function URL, you could use the following curl command:

```
curl -v -X POST \
  'https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld' \
  -H 'content-type: application/json' \
  -d '{ "example": "test" }'
```

Request and response payloads

When a client calls your function URL, Lambda maps the request to an event object before passing it to your function. Your function's response is then mapped to an HTTP response that Lambda sends back to the client through the function URL.

The request and response event formats follow the same schema as the [Amazon API Gateway payload format version 2.0](#).

Request payload format

A request payload has the following structure:

```
{
  "version": "2.0",
  "routeKey": "$default",
  "rawPath": "/my/path",
  "rawQueryString": "parameter1=value1&parameter1=value2&parameter2=value",
  "cookies": [
    "cookie1",
    "cookie2"
  ],
  "headers": {
    "header1": "value1",
    "header2": "value1,value2"
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "<url-id>",
    "authentication": null,
    "authorizer": {
      "iam": {
        "accessKey": "AKIA...",
        "accountId": "111122223333",
        "callerId": "AIDA...",
        "cognitoIdentity": null,
        "principalOrgId": null,
        "userArn": "arn:aws:iam::111122223333:user/example-user",
        "userId": "AIDA..."
      }
    },
    "domainName": "<url-id>.lambda-url.us-west-2.on.aws",
    "domainPrefix": "<url-id>",
    "http": {
      "method": "POST",
      "path": "/my/path",
      "protocol": "HTTP/1.1",
      "sourceIp": "123.123.123.123",
      "userAgent": "agent"
    },
    "requestId": "id",
    "routeKey": "$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	Example
version	The payload format version for this event. Lambda function URLs currently support payload format version 2.0 .	2.0
routeKey	Function URLs don't use this parameter. Lambda sets this to \$default as a placeholder.	\$default
rawPath	The request path. For example, if the request URL is https:// {url-id}.lambda-url. {region}.on.aws/example/test/demo, then the raw path value is /example/test/demo.	/example/test/demo
rawQueryString	The raw string containing the request's query string parameters.	"?parameter1=value1¶meter2=value2"
cookies	An array containing all cookies sent as part of the request.	["Cookie_1=Value_1", "Cookie_2=Value_2"]
headers	The list of request headers, presented as key-value pairs.	{"header1": "value1", "header2": "value2"}
queryStringParameters	The query parameters for the request. For example, if the request URL is https:// {url-id}.lambda-url. {region}.on.aws/example?name=Jane, then the queryStringParameters value is a JSON object with a key of name and a value of Jane.	{"name": "Jane"}
requestContext	An object that contains additional information about the request, such as the requestId, the time of the request, and the identity of the caller if authorized via AWS Identity and Access Management (IAM).	
requestContext.accountId	The AWS account ID of the function owner.	"123456789012"
requestContext.apiId	The ID of the function URL.	"33anwqw8fj"
requestContext.authenticator	Function URLs don't use this parameter. Lambda sets this to null.	null
requestContext.authorizer	An object that contains information about the caller identity, if the function URL uses the AWS_IAM auth type.	

Parameter	Description	Example
	Otherwise, Lambda sets this to null.	
requestContext.authorizer	The access key of the caller identity.	"AKIAIOSFODNN7EXAMPLE"
requestContext.authorizer	The AWS account ID of the caller identity.	"111122223333"
requestContext.authorizer	The ID (User ID) of the caller.	"AIDACKCEVSQ6C2EXAMPLE"
requestContext.authorizer	Function-level identity. This parameter. Lambda sets this to null or excludes this from the JSON.	null
requestContext.authorizer	The principal ID associated with the caller identity.	"AIDACKCEVSQORGEXAMPLE"
requestContext.authorizer	The user's Amazon Resource Name (ARN) of the caller identity.	"arn:aws:iam::111122223333:user/example-user"
requestContext.authorizer	The user ID of the caller identity.	"AIDACOSFODNN7EXAMPLE2"
requestContext.domainName	The domain name of the function URL.	"<url-id>.lambda-url.us-west-2.on.aws"
requestContext.domainPrefix	The domain prefix of the function URL.	"<url-id>"
requestContext.http	An object that contains details about the HTTP request.	
requestContext.http.method	The HTTP method used in this request. Valid values include GET, POST, PUT, HEAD, OPTIONS, PATCH, and DELETE.	GET
requestContext.http.path	The request path. For example, if the request URL is https://<url-id>.lambda-url.<region>.on.aws/example/test/demo, then the path value is /example/test/demo.	/example/test/demo
requestContext.http.protocol	The protocol of the request.	HTTP/1.1
requestContext.http.sourceIp	The source IP address of the immediate TCP connection making the request.	123.123.123.123
requestContext.http.userAgent	The User-Agent request header value.	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) Gecko/20100101 Firefox/42.0

Parameter	Description	Example
<code>requestContext.requestId</code>	The ID of the invocation request. You can use this ID to trace invocation logs related to your function.	e1506fd5-9e7b-434f-bd42-4f8fa224b599
<code>requestContext.routeKey</code>	Function URLs don't use this parameter. Lambda sets this to <code>\$default</code> as a placeholder.	<code>\$default</code>
<code>requestContext.stage</code>	Function URLs don't use this parameter. Lambda sets this to <code>\$default</code> as a placeholder.	<code>\$default</code>
<code>requestContext.time</code>	The timestamp of the request.	"07/Sep/2021:22:50:22 +0000"
<code>requestContext.timeEpoch</code>	The timestamp of the request, in Unix epoch time.	"1631055022677"
<code>body</code>	The body of the request. If the content type of the request is binary, the body is base64-encoded.	{"key1": "value1", "key2": "value2"}
<code>pathParameters</code>	Function URLs don't use this parameter. Lambda sets this to <code>null</code> or excludes this from the JSON.	<code>null</code>
<code>isBase64Encoded</code>	TRUE if the body is a binary payload and base64-encoded. FALSE otherwise.	FALSE
<code>stageVariables</code>	Function URLs don't use this parameter. Lambda sets this to <code>null</code> or excludes this from the JSON.	<code>null</code>

Response payload format

When your function returns a response, Lambda parses the response and converts it into an HTTP response. Function response payloads have the following 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 infers the response format for you. If your function returns valid JSON and doesn't return a `statusCode`, Lambda assumes the following:

- `statusCode` is `200`.
- `content-type` is `application/json`.
- `body` is the function response.
- `isBase64Encoded` is `false`.

The following examples show how the output of your Lambda function maps to the response payload, and how the response payload maps to the final HTTP response. When the client invokes your function URL, they see the HTTP response.

Example output for a string response

Lambda function output	Interpreted response output	HTTP response (what the client sees)
"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!"

Example output for a JSON response

Lambda function output	Interpreted response output	HTTP response (what the client sees)
{ "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!" }

Example output for a custom response

Lambda function output	Interpreted response output	HTTP response (what the client sees)
{ "statusCode": 201, "headers": { "Content-Type": "application/json", } }	{ "statusCode": 201, "headers": { "Content-Type": "application/json", } }	HTTP/2 201 date: Wed, 08 Sep 2021 18:02:24 GMT content-type: application/json

Lambda function output	Interpreted response output	HTTP response (what the client sees)
<pre> "My-Custom-Header": "Custom Value" }, "body": JSON.stringify({ "message": "Hello, world!" }), "isBase64Encoded": false } </pre>	<pre> "My-Custom-Header": "Custom Value" }, "body": JSON.stringify({ "message": "Hello, world!" }), "isBase64Encoded": false } </pre>	<pre> content-length: 27 my-custom-header: Custom Value { "message": "Hello, world!" } </pre>

Cookies

To return cookies from your function, don't manually add `set-cookie` headers. Instead, include the cookies in your response payload object. Lambda automatically interprets this and adds them as `set-cookie` headers in your HTTP response, as in the following example.

Example output for a response returning cookies

Lambda function output	HTTP response (what the client sees)
<pre> { "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 } </pre>	<pre> 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!" } </pre>

Monitoring Lambda function URLs

You can use AWS CloudTrail and Amazon CloudWatch to monitor your function URLs.

Topics

- [Monitoring function URLs with CloudTrail \(p. 184\)](#)
- [CloudWatch metrics for function URLs \(p. 184\)](#)

Monitoring function URLs with CloudTrail

For function URLs, Lambda automatically supports logging the following API operations as events in CloudTrail log files:

- [CreateFunctionUrlConfig](#)
- [UpdateFunctionUrlConfig](#)
- [DeleteFunctionUrlConfig](#)
- [GetFunctionUrlConfig](#)
- [ListFunctionUrlConfigs](#)

Each log entry contains information about the caller identity, when the request was made, and other details. You can see all events within the last 90 days by viewing your CloudTrail **Event history**. To retain records past 90 days, you can create a trail. For more information, see [Using AWS Lambda with AWS CloudTrail \(p. 583\)](#).

By default, CloudTrail doesn't log InvokeFunctionUrl requests, which are considered data events. However, you can turn on data event logging in CloudTrail. For more information, see [Logging data events for trails](#) in the *AWS CloudTrail User Guide*.

CloudWatch metrics for function URLs

Lambda sends aggregated metrics about function URL requests to CloudWatch. With these metrics, you can monitor your function URLs, build dashboards, and configure alarms in the CloudWatch console.

Function URLs support the following invocation metrics. We recommend viewing these metrics with the Sum statistic.

- `UrlRequestCount` – The number of requests made to this function URL.
- `Url4xxError` – The number of requests that returned a 4XX HTTP status code. 4XX series codes indicate client-side errors, such as bad requests.
- `Url5xxError` – The number of requests that returned a 5XX HTTP status code. 5XX series codes indicate server-side errors, such as function errors and timeouts.

Function URLs also support the following performance metric. We recommend viewing this metric with the Average or Max statistics.

- `UrlRequestLatency` – The time between when the function URL receives a request and when the function URL returns a response.

Each of these invocation and performance metrics supports the following dimensions:

- `FunctionName` – View aggregate metrics for function URLs assigned to a function's \$LATEST unpublished version, or to any of the function's aliases. For example, hello-world-function.

- **Resource** – View metrics for a specific function URL. This is defined by a function name, along with either the function's \$LATEST unpublished version or one of the function's aliases. For example, hello-world-function:\$LATEST.
- **ExecutedVersion** – View metrics for a specific function URL based on the executed version. You can use this dimension primarily to track the function URL assigned to the \$LATEST unpublished version.

Tutorial: Creating a Lambda function with a function URL

In this tutorial, you create a Lambda function defined as a .zip file archive with a **public** function URL endpoint that returns the product of two numbers. For more information about configuring function URLs, see [Creating and managing function URLs \(p. 167\)](#).

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console \(p. 4\)](#) to create your first Lambda function.

To complete the following steps, you need the [AWS Command Line Interface \(AWS CLI\) version 2](#). Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.0.57 Python/3.7.4 Darwin/19.6.0 exe/x86_64
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#). Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

Create an execution role

Create the [execution role \(p. 816\)](#) that gives your Lambda function permission to access AWS resources.

To create an execution role

1. Open the [Roles page](#) of the AWS Identity and Access Management (IAM) console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – AWS Lambda.**
 - **Permissions – AWSLambdaBasicExecutionRole.**
 - **Role name – lambda-url-role.**

The **AWSLambdaBasicExecutionRole** policy has the permissions that the function needs to write logs to Amazon CloudWatch Logs.

Create a Lambda function with a function URL (.zip file archive)

Create a Lambda function with a function URL endpoint using a .zip file archive.

To create the function

1. Copy the following code example into a file named index.js.

Example index.js

```
exports.handler = async (event) => {
  let body = JSON.parse(event.body)
  const product = body.num1 * body.num2;
  const response = {
    statusCode: 200,
    body: "The product of " + body.num1 + " and " + body.num2 + " is " + product,
  };
  return response;
};
```

2. Create a deployment package.

```
zip function.zip index.js
```

3. Create a Lambda function with the create-function command.

```
aws lambda create-function \
--function-name my-url-function \
--runtime nodejs14.x \
--zip-file fileb://function.zip \
--handler index.handler \
--role arn:aws:iam::123456789012:role/Lambda-url-role
```

4. Add a resource-based policy to your function granting permissions to allow public access to your function URL.

```
aws lambda add-permission \
--function-name my-url-function \
--action lambda:InvokeFunctionUrl \
--principal "*" \
--function-url-auth-type "NONE" \
--statement-id url
```

5. Create a URL endpoint for the function with the create-function-url-config command.

```
aws lambda create-function-url-config \
--function-name my-url-function \
--auth-type NONE
```

Test the function URL endpoint

Invoke your Lambda function by calling your function URL endpoint using an HTTP client such as curl or Postman.

```
curl -X POST \
'https://abcdefg.lambda-url.us-east-1.on.aws/' \
-H 'Content-Type: application/json' \
-d '{"num1": "10", "num2": "10"}'
```

You should see the following output:

```
The product of 10 and 10 is 100
```

Create a Lambda function with a function URL (CloudFormation)

You can also create a Lambda function with a function URL endpoint using the AWS CloudFormation type `AWS::Lambda::Url`.

```
Resources:
  MyUrlFunction:
    Type: AWS::Lambda::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs14.x
      Role: arn:aws:iam::123456789012:role/lambda-url-role
      Code:
        ZipFile: |
          exports.handler = async (event) => {
            let body = JSON.parse(event.body)
            const product = body.num1 * body.num2;
            const response = {
              statusCode: 200,
              body: "The product of " + body.num1 + " and " + body.num2 + " is " +
product,
            };
            return response;
          };
        Description: Create a function with a URL.
  MyUrlFunctionPermissions:
    Type: AWS::Lambda::Permission
    Properties:
      FunctionName: !Ref MyUrlFunction
      Action: lambda:InvokeFunctionUrl
      Principal: "*"
      FunctionUrlAuthType: NONE
  MyFunctionUrl:
    Type: AWS::Lambda::Url
    Properties:
      TargetFunctionArn: !Ref MyUrlFunction
      AuthType: NONE
```

Create a Lambda function with a function URL (AWS SAM)

You can also create a Lambda function configured with a function URL using AWS Serverless Application Model (AWS SAM).

```
ProductFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: function/
    Handler: index.handler
    Runtime: nodejs14.x
    AutoPublishAlias: live
    FunctionUrlConfig:
      AuthType: NONE
```

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

Managing AWS Lambda functions

Learn how to adjust and secure the resources associated with your Lambda function using the Lambda API or console.

[Using Lambda with the AWS CLI \(p. 191\)](#)

You can use the AWS Command Line Interface to manage functions and other AWS Lambda resources. The AWS CLI uses the AWS SDK for Python (Boto) to interact with the Lambda API. In this tutorial, you manage and invoke Lambda functions with the AWS CLI.

[Function Scaling \(p. 197\)](#)

You can configure two function-level concurrency controls: reserved concurrency and provisioned concurrency. Concurrency is the number of instances of your function that are active and can be configured to ensure critical functions avoid throttling.

[Network configuration \(p. 222\)](#)

You can use your Lambda function with AWS resources in an Amazon VPC. Connecting your function to a VPC lets you access resources in a private subnet such as relational databases and caches.

[Interface VPC endpoints \(p. 229\)](#)

You can use an interface VPC endpoint to invoke your Lambda functions without crossing the public internet.

[Database \(p. 232\)](#)

You can create a database proxy for MySQL and Aurora DB instances. A database proxy enables a function to reach high concurrency levels without exhausting database connections.

[File system \(p. 236\)](#)

You can use your Lambda function to mount a Amazon EFS to a local directory. A file system allows your function code to access and modify shared resources safely and at high concurrency.

[Code signing \(p. 241\)](#)

Code signing for Lambda provides trust and integrity controls that let you verify that only unaltered code that approved developers have published is deployed in your Lambda functions.

[Organize with tags \(p. 250\)](#)

You can tag Lambda functions to activate [attribute-based access control \(ABAC\) \(p. 828\)](#) and to organize them by owner, project, or department.

[Using layers \(p. 245\)](#)

You can apply previously created layers to reduce deployment package size and promote code sharing and separation of responsibilities so that you can iterate faster on writing business logic.

Using Lambda with the AWS CLI

You can use the AWS Command Line Interface to manage functions and other AWS Lambda resources. The AWS CLI uses the AWS SDK for Python (Boto) to interact with the Lambda API. You can use it to learn about the API, and apply that knowledge in building applications that use Lambda with the AWS SDK.

In this tutorial, you manage and invoke Lambda functions with the AWS CLI. For more information, see [What is the AWS CLI?](#) in the *AWS Command Line Interface User Guide*.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [the section called "Create a function" \(p. 4\)](#).

To complete the following steps, you need the [AWS Command Line Interface \(AWS CLI\) version 2](#). Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.0.57 Python/3.7.4 Darwin/19.6.0 exe/x86_64
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#). Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

Create the execution role

Create the [execution role \(p. 816\)](#) that gives your function permission to access AWS resources. To create an execution role with the AWS CLI, use the `create-role` command.

In the following example, you specify the trust policy inline. Requirements for escaping quotes in the JSON string vary depending on your shell.

```
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"}]}'
```

You can also define the [trust policy](#) for the role using a JSON file. In the following example, `trust-policy.json` is a file in the current directory. This trust policy allows Lambda to use the role's permissions by giving the service principal `lambda.amazonaws.com` permission to call the AWS Security Token Service (AWS STS) `AssumeRole` action.

Example `trust-policy.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
```

You should see the following output:

```
{
  "Role": {
    "Path": "/",
    "RoleName": "lambda-ex",
    "RoleId": "AROAQFOXMP6TZ6ITKWND",
    "Arn": "arn:aws:iam::123456789012:role/lambda-ex",
    "CreateDate": "2020-01-17T23:19:12Z",
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "lambda.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
  }
}
```

To add permissions to the role, use the **attach-policy-to-role** command. Start by adding the **AWSLambdaBasicExecutionRole** managed policy.

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

The **AWSLambdaBasicExecutionRole** policy has the permissions that the function needs to write logs to CloudWatch Logs.

Create the function

The following example logs the values of environment variables and the event object.

Example index.js

```
exports.handler = async function(event, context) {
  console.log("ENVIRONMENT VARIABLES\n" + JSON.stringify(process.env, null, 2))
  console.log("EVENT\n" + JSON.stringify(event, null, 2))
  return context.logStreamName
}
```

To create the function

1. Copy the sample code into a file named `index.js`.
2. Create a deployment package.

```
zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command. Replace the highlighted text in the role ARN with your account ID.

```
aws lambda create-function --function-name my-function \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--role arn:aws:iam::123456789012:role/lambda-ex
```

You should see the following output:

```
{  
    "FunctionName": "my-function",  
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",  
    "Runtime": "nodejs18.x",  
    "Role": "arn:aws:iam::123456789012:role/lambda-ex",  
    "Handler": "index.handler",  
    "CodeSha256": "FpFMvUhayLkOoVBpNuNiIVML/tuGv2iJQ7t0yWVTU8c=",  
    "Version": "$LATEST",  
    "TracingConfig": {  
        "Mode": "PassThrough"  
    },  
    "RevisionId": "88ebe1e1-bfdf-4dc3-84de-3017268fa1ff",  
    ...  
}
```

To get logs for an invocation from the command line, use the `--log-type` option. The response includes a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{  
    "StatusCode": 200,  
    "LogResult":  
    "U1RBULQgUmVxdWVzdElk0iaA4N2QwNDRI0C1mMTU0LTEzTgt0GNkYS0y0Tc0YzVlNGZiMjEgVmVyc21vb...",  
    "ExecutedVersion": "$LATEST"  
}
```

You can use the `base64` utility to decode the logs.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text | base64 -d
```

You should see the following output:

```
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
```

The base64 utility is available on Linux, macOS, and [Ubuntu on Windows](#). For macOS, the command is `base64 -D`.

To get full log events from the command line, you can include the log stream name in the output of your function, as shown in the preceding example. The following example script invokes a function named `my-function` and downloads the last five log events.

Example `get-logs.sh` Script

This example requires that `my-function` returns a log stream ID.

```
#!/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 $(cat
out) --limit 5
```

The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to be available. The output includes the response from Lambda and the output from the `get-log-events` command.

```
./get-logs.sh
```

You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
$LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r\t\t\"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\", \r\t\t...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r\t\t\"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"
}
```

Update the function

After you create a function, you can configure additional capabilities for the function, such as triggers, network access, and file system access. You can also adjust resources associated with the function, such as memory and concurrency. These configurations apply to functions defined as .zip file archives and to functions defined as container images.

Use the [update-function-configuration](#) command to configure functions. The following example sets the function memory to 256 MB.

Example update-function-configuration command

```
aws lambda update-function-configuration \
--function-name my-function \
--memory-size 256
```

List the Lambda functions in your account

Run the following AWS CLI list-functions command to retrieve a list of functions that you have created.

```
aws lambda list-functions --max-items 10
```

You should see the following output:

```
{
    "Functions": [
        {
            "FunctionName": "cli",
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
            "Runtime": "nodejs18.x",
            "Role": "arn:aws:iam::123456789012:role/lambda-ex",
            "Handler": "index.handler",
            ...
        },
        {
            "FunctionName": "random-error",
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:random-error",
            "Runtime": "nodejs18.x",
            "Role": "arn:aws:iam::123456789012:role/lambda-role",
            "Handler": "index.handler",
            ...
        },
        ...
    ],
    "NextToken": "eyJNYXJrZXIiOiBudWxsLCAiYm90b190cnVuY2F0ZV9hbW91bnQiOjAxMH0="
}
```

In response, Lambda returns a list of up to 10 functions. If there are more functions you can retrieve, NextToken provides a marker you can use in the next list-functions request. The following list-functions AWS CLI command is an example that shows the --starting-token parameter.

```
aws lambda list-functions --max-items 10 --starting-token eyJNYXJrZXIiOiBudWxsLCAiYm90b190cnVuY2F0ZV9hbW91bnQiOjAxMH0=
```

Retrieve a Lambda function

The Lambda CLI get-function command returns Lambda function metadata and a presigned URL that you can use to download the function's deployment package.

```
aws lambda get-function --function-name my-function
```

You should see the following output:

```
{  
    "Configuration": {  
        "FunctionName": "my-function",  
        "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",  
        "Runtime": "nodejs18.x",  
        "Role": "arn:aws:iam::123456789012:role/lambda-ex",  
        "CodeSha256": "FpFMvUhayLk0oVBpNuNiIVML/tuGv2iJQ7t0yWVTU8c=",  
        "Version": "$LATEST",  
        "TracingConfig": {  
            "Mode": "PassThrough"  
        },  
        "RevisionId": "88ebe1e1-bfdf-4dc3-84de-3017268fa1ff",  
        ...  
    },  
    "Code": {  
        "RepositoryType": "S3",  
        "Location": "https://awslambda-us-east-2-tasks.s3.us-east-2.amazonaws.com/  
snapshots/123456789012/my-function-4203078a-b7c9-4f35-..."  
    }  
}
```

For more information, see [GetFunction \(p. 1220\)](#).

Clean up

Run the following delete-function command to delete the my-function function.

```
aws lambda delete-function --function-name my-function
```

Delete the IAM role you created in the IAM console. For information about deleting a role, see [Deleting roles or instance profiles](#) in the *IAM User Guide*.

Lambda function scaling

Concurrency is the number of in-flight requests your AWS Lambda function is handling at the same time. For each concurrent request, Lambda provisions a separate instance of your execution environment. As your functions receive more requests, Lambda automatically handles scaling the number of execution environments until you reach your account's concurrency limit. By default, Lambda provides your account with a total concurrency limit of 1,000 across all functions in a region. To support your specific account needs, you can [request a quota increase](#) and configure function-level concurrency controls so that your critical functions don't experience throttling.

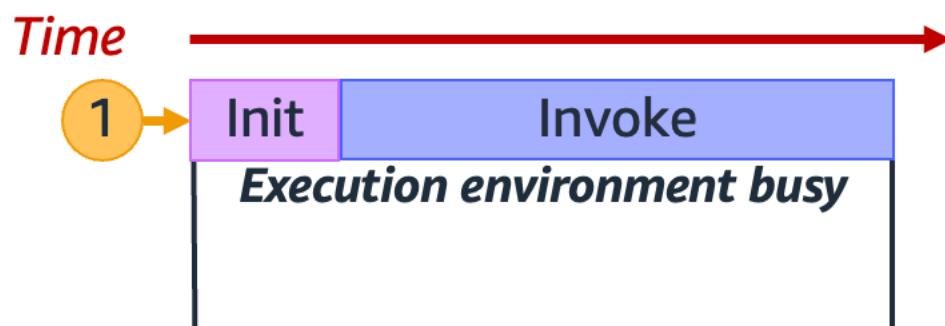
This topic explains concurrency and function scaling in Lambda. By the end of this topic, you'll be able to understand how to calculate concurrency, visualize the two main concurrency control options (reserved and provisioned), estimate appropriate concurrency control settings, and view metrics for further optimization.

Sections

- [Understanding and visualizing concurrency \(p. 197\)](#)
- [How to calculate concurrency \(p. 200\)](#)
- [Reserved concurrency and provisioned concurrency \(p. 201\)](#)
- [Concurrency quotas \(p. 207\)](#)
- [Accurately estimating required concurrency \(p. 208\)](#)
- [Concurrency metrics \(p. 208\)](#)
- [Configuring reserved concurrency \(p. 210\)](#)
- [Configuring provisioned concurrency \(p. 213\)](#)
- [Burst concurrency \(p. 220\)](#)

Understanding and visualizing concurrency

Lambda invokes your function in a secure and isolated [execution environment](#). To handle a request, Lambda must first initialize an execution environment (the [Init phase](#)), before using it to invoke your function (the [Invoke phase](#)):



Note

Actual Init and Invoke durations can vary depending on many factors, such as the runtime you choose and the Lambda function code. The previous diagram isn't meant to represent the exact proportions of Init and Invoke phase durations.

The previous diagram uses a rectangle to represent a single execution environment. When your function receives its very first request (represented by the yellow circle with label 1), Lambda creates a new

execution environment and runs the code outside your main handler during the Init phase. Then, Lambda runs your function's main handler code during the Invoke phase. During this entire process, this execution environment is busy and cannot process other requests.

When Lambda finishes processing the first request, this execution environment can then process additional requests for the same function. For subsequent requests, Lambda doesn't need to re-initialize the environment.

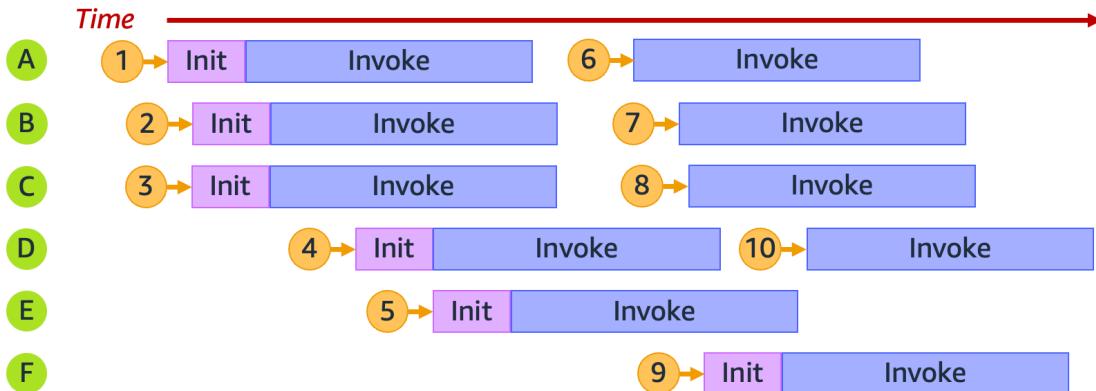


In the previous diagram, Lambda reuses the execution environment to handle the second request (represented by the yellow circle with label 2).

So far, we've focused on just a single instance of your execution environment (i.e. a concurrency of 1). In practice, Lambda may need to provision multiple execution environment instances in parallel to handle all incoming requests. When your function receives a new request, one of two things can happen:

- If a pre-initialized execution environment instance is available, Lambda uses it to process the request.
- Otherwise, Lambda creates a new execution environment instance to process the request.

For example, let's explore what happens when your function receives 10 requests:



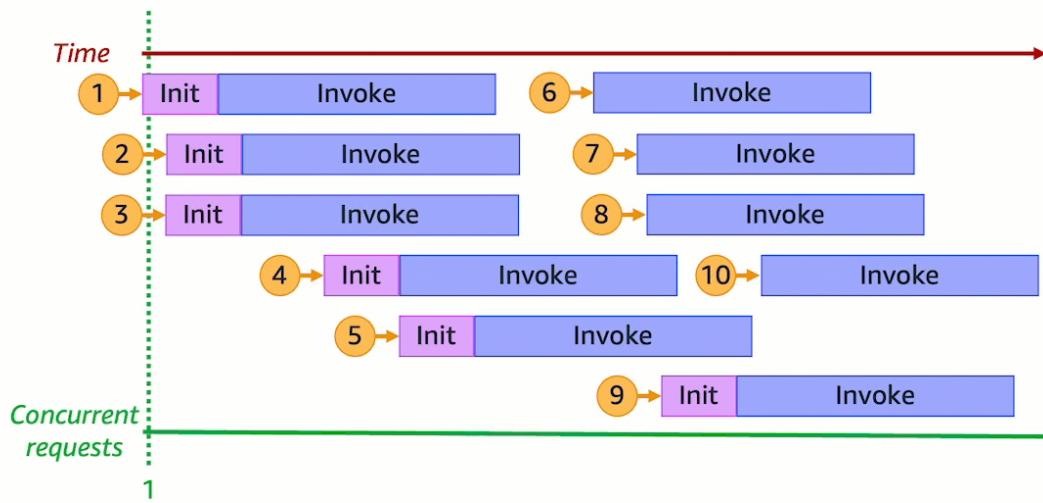
In the previous diagram, each horizontal plane represents a single execution environment instance (labeled from A through F). Here's how Lambda handles each request:

Lambda behavior for requests 1 through 10

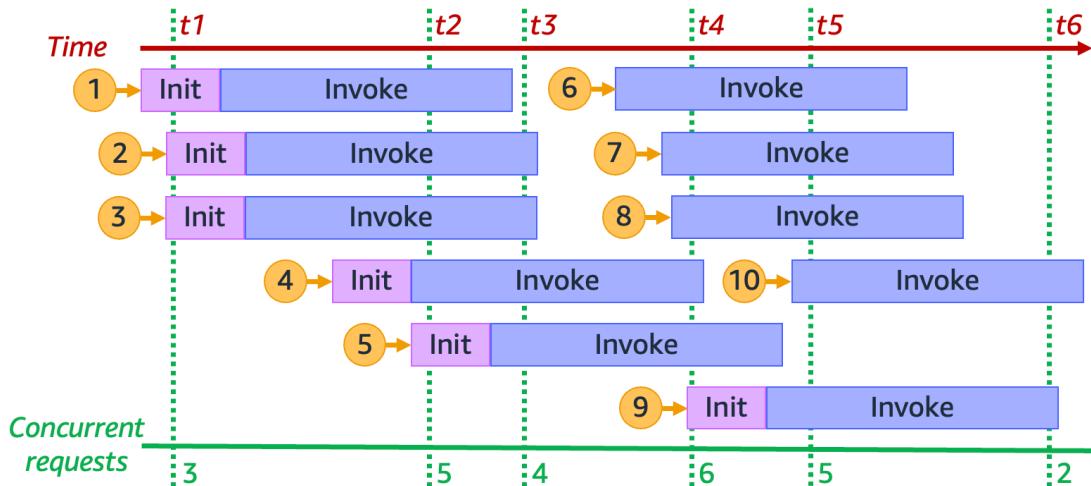
Request	Lambda behavior	Reasoning
1	Provisions new environment A	This is the first request; no execution environment instances available

Request	Lambda behavior	Reasoning
2	Provisions new environment B	Existing execution environment instance A is busy
3	Provisions new environment C	Existing execution environment instances A and B are both busy
4	Provisions new environment D	Existing execution environment instances A , B , and C are all busy
5	Provisions new environment E	Existing execution environment instances A , B , C , and D are all busy
6	Reuses environment A	Execution environment instance A has finished processing request 1 and is now available
7	Reuses environment B	Execution environment instance B has finished processing request 2 and is now available
8	Reuses environment C	Execution environment instance C has finished processing request 3 and is now available
9	Provisions new environment F	Existing execution environment instances A , B , C , D , and E are all busy
10	Reuses environment D	Execution environment instance D has finished processing request 4 and is now available

As your function receives more concurrent requests, Lambda scales up the number of execution environment instances in response. The following animation tracks the number of concurrent requests over time:



By freezing the previous animation at six distinct points in time, we get the following diagram:



In the previous diagram, we can draw a vertical line at any point in time and count the number of environments that intersect this line. This gives us the number of concurrent requests at that point in time. For example, at time t_1 , there are 3 active environments serving 3 concurrent requests. The maximum number of concurrent requests in this simulation occurs at time t_4 , when there are 6 active environments serving 6 concurrent requests.

To summarize, your function's concurrency is the number of concurrent requests that it's handling at the same time. In response to an increase in your function's concurrency, Lambda provisions more execution environment instances to meet request demand.

How to calculate concurrency

In general, concurrency of a system is the ability to process more than one task simultaneously. In Lambda, concurrency is the number of in-flight requests that your function is handling at the same

time. A quick and practical way of measuring concurrency of a Lambda function is to use the following formula:

$$\text{Concurrency} = (\text{average requests per second}) * (\text{average request duration in seconds})$$

Concurrency differs from requests per second. For example, suppose your function receives 100 requests per second on average. If the average request duration is 1 second, then it's true that the concurrency is also 100:

$$\text{Concurrency} = (100 \text{ requests/second}) * (1 \text{ second/request}) = 100$$

However, if the average request duration is 500 ms, the concurrency is 50:

$$\text{Concurrency} = (100 \text{ requests/second}) * (0.5 \text{ second/request}) = 50$$

What does a concurrency of 50 mean in practice? If the average request duration is 500 ms, you can think of an instance of your function as being able to handle 2 requests per second. Then, it takes 50 instances of your function to handle a load of 100 requests per second. A concurrency of 50 means that Lambda must provision 50 execution environment instances to efficiently handle this workload without any throttling. Here's how to express this in equation form:

$$\text{Concurrency} = (100 \text{ requests/second}) / (2 \text{ requests/second}) = 50$$

If your function receives double the number of requests (200 requests per second), but only requires half the time to process each request (250 ms), the concurrency is still 50:

$$\text{Concurrency} = (200 \text{ requests/second}) * (0.25 \text{ second/request}) = 50$$

Test your understanding of concurrency

Suppose you have a function that takes, on average, 20 ms to run. During peak load, you observe 5,000 asynchronous requests per second. What is the concurrency of your function during peak load?

Answer

The average function duration is 20 ms, or 0.020 seconds. Using the concurrency formula, you can plug in the numbers to get a concurrency of 100:

$$\text{Concurrency} = (5,000 \text{ requests/second}) * (0.020 \text{ seconds/request}) = 100$$

Alternatively, an average function duration of 20 ms means that your function can process 50 requests per second. To handle the 5,000 request per second workload, you need 100 execution environment instances. Thus, the concurrency is 100:

$$\text{Concurrency} = (5,000 \text{ requests/second}) / (50 \text{ requests/second}) = 100$$

Reserved concurrency and provisioned concurrency

By default, your account has a concurrency limit of 1,000 across all functions in a region. Your functions share this pool of 1,000 concurrency on an on-demand basis. Your function experiences throttling (i.e. it starts to drop requests) if you run out of available concurrency.

Some of your functions might be more critical than others. As a result, you might want to configure concurrency settings to ensure that critical functions get the concurrency they need. There are two types of concurrency controls available: reserved concurrency and provisioned concurrency.

- Use **reserved concurrency** to reserve a portion of your account's concurrency for a function. This is useful if you don't want other functions taking up all the available unreserved concurrency.
- Use **provisioned concurrency** to pre-initialize a number of environment instances for a function. This is useful for reducing cold start latencies.

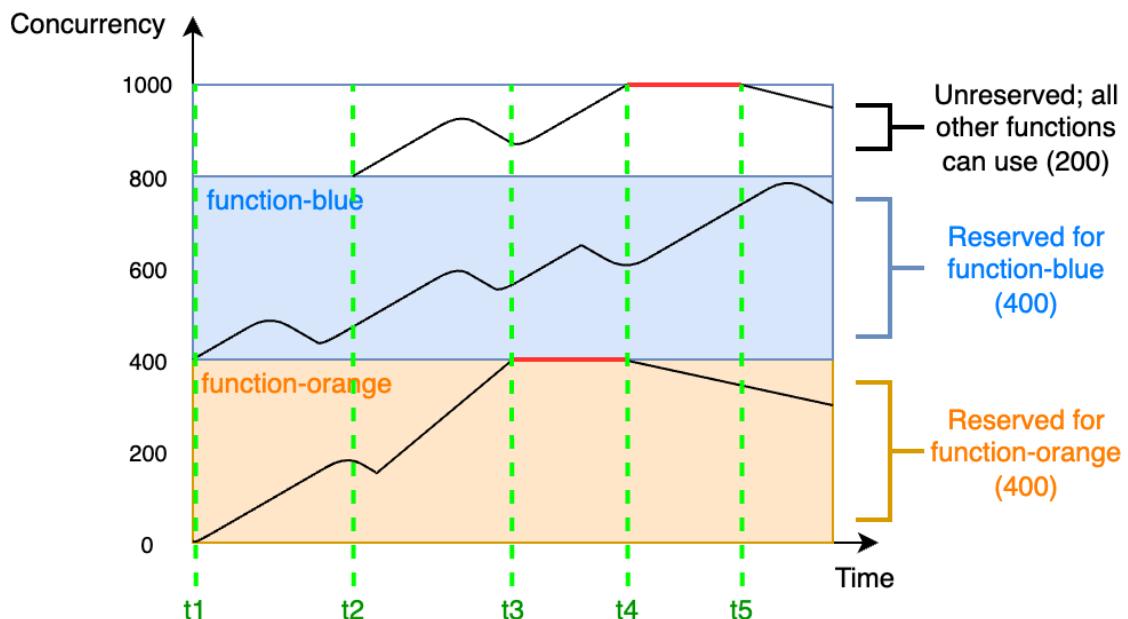
Reserved concurrency

If you want to guarantee that a certain amount of concurrency is available for your function at any time, use reserved concurrency.

Reserved concurrency is the maximum number of concurrent instances you want to allocate to your function. When you dedicate reserved concurrency to a function, no other function can use that concurrency. In other words, setting reserved concurrency can impact the concurrency pool that's available to other functions. Functions that don't have reserved concurrency share the remaining pool of unreserved concurrency.

Configuring reserved concurrency counts towards your overall account concurrency limit. There is no charge for configuring reserved concurrency for a function.

To better understand reserved concurrency, consider the following diagram:



In this diagram, your account concurrency limit for all the functions in this region is at the default limit of 1,000. Suppose you have two critical functions, function-blue and function-orange, that routinely expect to get high invocation volumes. You decide to give 400 units of reserved concurrency to function-blue, and 400 units of reserved concurrency to function-orange. In this example, all other functions in your account must share the remaining 200 units of unreserved concurrency.

The diagram has 5 points of interest:

- At t1, both function-orange and function-blue begin receiving requests. Each function begins to use up their allocated portion of reserved concurrency units.
- At t2, function-orange and function-blue are steadily receiving more requests. At the same time, you deploy some other Lambda functions, which begin receiving requests. You do not

allocate reserved concurrency to these other functions. They begin using the remaining 200 units of unreserved concurrency.

- At t3, function-orange hits the max concurrency of 400. Although there is unused concurrency elsewhere in your account, function-orange cannot access it. The red line indicates that function-orange is experiencing throttling, and Lambda may drop requests.
- At t4, function-orange starts to receive fewer requests and is no longer throttling. However, your other functions experience a spike in traffic and begin throttling. Although there is unused concurrency elsewhere in your account, these other functions cannot access it. The red line indicates that your other functions are experiencing throttling.
- At t5, other functions start to receive fewer requests and are no longer throttling.

From this example, notice that reserving concurrency has the following effects:

- **Your function can scale independently of other functions in your account.** All of your account's functions in the same region that don't have reserved concurrency share the pool of unreserved concurrency. Without reserved concurrency, other functions can potentially use up all of your available concurrency. This prevents critical functions from scaling up if needed.
- **Your function can't scale out of control.** Reserved concurrency puts a cap on your function's maximum concurrency. This means that your function can't use concurrency reserved for other functions, or concurrency from the unreserved pool. You can reserve concurrency to prevent your function from using all the available concurrency in your account, or from overloading downstream resources.
- **You may not be able to use all of your account's available concurrency.** Reserving concurrency counts towards your account concurrency limit, but this also means that other functions cannot use that chunk of reserved concurrency. If your function doesn't use up all of the concurrency that you reserve for it, you're effectively wasting that concurrency. This isn't an issue unless other functions in your account could benefit from the wasted concurrency.

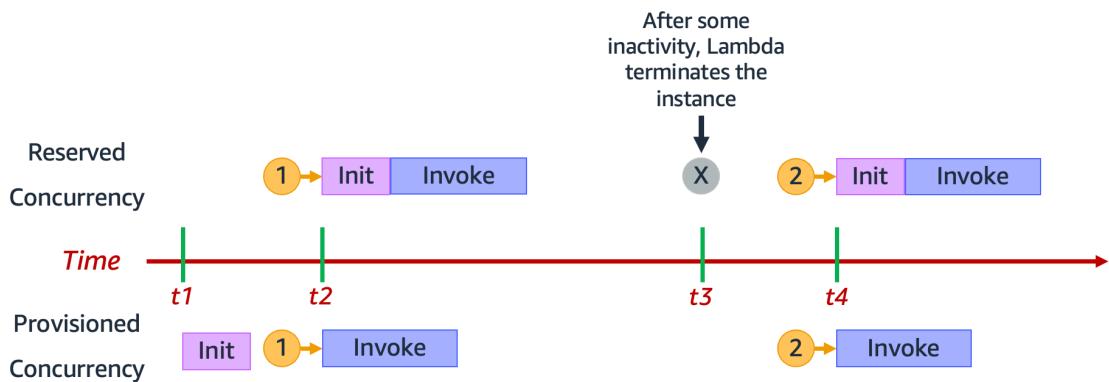
To manage reserved concurrency settings for your functions, see [Configuring reserved concurrency \(p. 210\)](#).

Provisioned concurrency

You use reserved concurrency to define the maximum number of execution environments reserved for a Lambda function. However, none of these environments come pre-initialized. As a result, your function invocations may take longer because Lambda must first initialize the new environment before being able to use it to invoke your function. When initialization takes longer than expected, this is known as a cold start. To mitigate cold starts, you can use provisioned concurrency.

Provisioned concurrency is the number of pre-initialized execution environments you want to allocate to your function. If you set provisioned concurrency on a function, Lambda initializes that number of execution environments so that they are prepared to respond immediately to function requests.

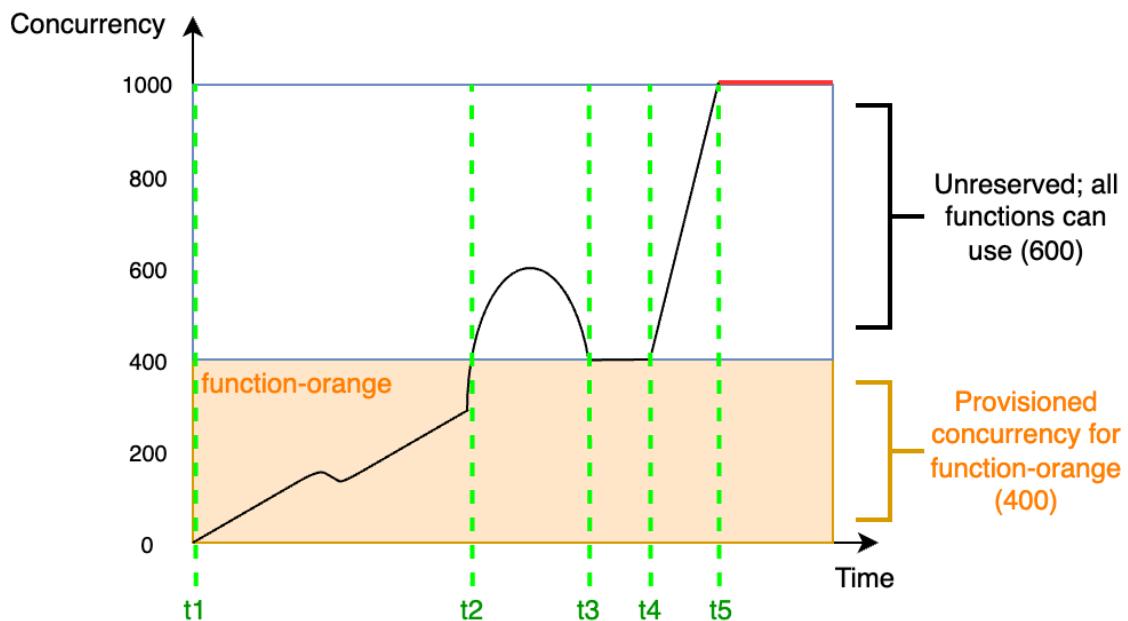
When using provisioned concurrency, Lambda still recycles execution environments in the background. However, at any given time, Lambda always ensures that the number of pre-initialized environments is equal to the value of your function's provisioned concurrency setting. This behavior differs from reserved concurrency, where Lambda may completely terminate an environment after a period of inactivity. The following diagram illustrates this by comparing the lifecycle of a single execution environment when you configure your function using reserved concurrency compared to provisioned concurrency.



The diagram has four points of interest:

Time	Reserved concurrency	Provisioned concurrency
t_1	Nothing happens.	Lambda pre-initializes an execution environment instance.
t_2	Request 1 comes in. Lambda must initialize a new execution environment instance.	Request 1 comes in. Lambda uses the pre-initialized environment instance.
t_3	After some inactivity, Lambda terminates the active environment instance.	Nothing happens.
t_4	Request 2 comes in. Lambda must initialize a new execution environment instance.	Request 2 comes in. Lambda uses the pre-initialized environment instance.

To better understand provisioned concurrency, consider the following diagram:

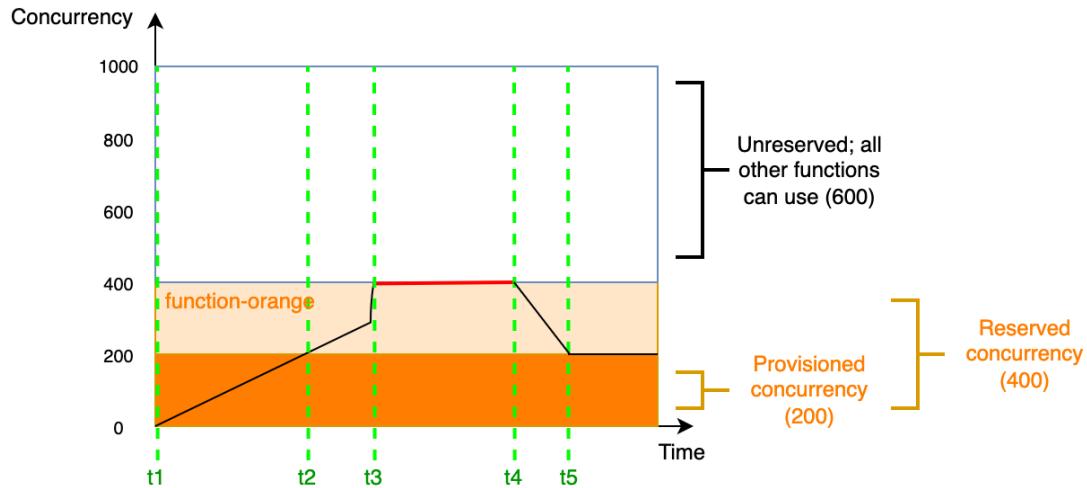


In this diagram, you have an account concurrency limit of 1,000. You decide to give 400 units of provisioned concurrency to `function-orange`. All functions in your account, **including** `function-orange`, can use the remaining 600 units of unreserved concurrency.

The diagram has 5 points of interest:

- At t_1 , `function-orange` begins receiving requests. Since Lambda has pre-initialized 400 execution environment instances, `function-orange` is ready for immediate invocation.
- At t_2 , `function-orange` reaches 400 concurrent requests. As a result, `function-orange` runs out of provisioned concurrency. However, since there's still unreserved concurrency available, Lambda can use this to handle additional requests to `function-orange` (there's no throttling). Lambda must create new instances to serve these requests, and your function may experience cold start latencies.
- At t_3 , `function-orange` returns to 400 concurrent requests after a brief spike in traffic. Lambda is again able to handle all requests without cold start latencies.
- At t_4 , functions in your account experience a burst in traffic. This burst can come from `function-orange` or any other function in your account. Lambda uses unreserved concurrency to handle these requests.
- At t_5 , functions in your account reach the maximum concurrency limit of 1,000, and experience throttling.

The previous example only considered provisioned concurrency. In practice, you can set both provisioned concurrency and reserved concurrency on a function. You might do this if you had a function that handles a consistent load of invocations, but routinely sees spikes of traffic during the weekends. In this case, you could use provisioned concurrency to set a baseline amount of environments to handle request during weekdays, and use reserved concurrency to handle the weekend spikes. Consider the following diagram:



In this diagram, suppose that you configure 200 units of provisioned concurrency and 400 units of reserved concurrency for function-orange. Because you configured reserved concurrency, function-orange cannot use any of the 600 units of unreserved concurrency.

This diagram has 5 points of interest:

- At t1, function-orange begins receiving requests. Since Lambda has pre-initialized 200 execution environment instances, function-orange is ready for immediate invocation.
- At t2, function-orange uses up all its provisioned concurrency. Function-orange can continue serving requests using reserved concurrency, but these requests may experience cold start latencies.
- At t3, function-orange reaches 400 concurrent requests. As a result, function-orange uses up all its reserved concurrency. Since function-orange cannot use unreserved concurrency, requests begin to throttle.
- At t4, function-orange starts to receive fewer requests, and no longer throttles.
- At t5, function-orange drops down to 200 concurrent requests, so all requests are again able to use provisioned concurrency (i.e. no cold start latencies).

Both reserved concurrency and provisioned concurrency count towards your account concurrency limit and [Regional quotas](#). In other words, allocating reserved and provisioned concurrency can impact the concurrency pool that's available to other functions. Configuring provisioned concurrency incurs charges to your AWS account.

Note

If the amount of provisioned concurrency on a function's versions and aliases adds up to the function's reserved concurrency, all invocations run on provisioned concurrency. This configuration also has the effect of throttling the unpublished version of the function (\$LATEST), which prevents it from executing. You can't allocate more provisioned concurrency than reserved concurrency for a function.

To manage provisioned concurrency settings for your functions, see [configuring provisioned concurrency](#). To automate provisioned concurrency scaling based on a schedule or application utilization, see [managing provisioned concurrency with Application Auto Scaling](#).

Comparing reserved concurrency and provisioned concurrency

The following is a table summarizing and comparing reserved and provisioned concurrency.

Topic	Reserved concurrency	Provisioned concurrency
Definition	Maximum number of execution environment instances for your function.	Set number of pre-provisioned execution environment instances for your function.
Provisioning behavior	Lambda provisions new instances on an on-demand basis.	Lambda pre-provisions instances (i.e. before your function starts receiving requests).
Cold start behavior	Cold start latency possible, since Lambda must create new instances on-demand.	Cold start latency eliminated, since Lambda doesn't have to create instances on-demand.
Throttling behavior	Function throttled when reserved concurrency limit reached.	If reserved concurrency not set: function uses unreserved concurrency when provisioned concurrency limit reached. If reserved concurrency set: function throttled when reserved concurrency limit reached.
Default behavior if not set	Function uses unreserved concurrency available in your account.	Lambda doesn't pre-provision any instances. Instead, if reserved concurrency not set: function uses unreserved concurrency available in your account. If reserved concurrency set: function uses reserved concurrency.
Pricing	No additional charge.	Incurs additional charges.

Concurrency quotas

Lambda sets quotas for the total amount of concurrency you can use across all functions in a region. These quotas exist on two levels:

- **At the account level**, your functions can have up to 1,000 units of concurrency by default. To increase this limit, see [Requesting a quota increase](#) in the *Service Quotas User Guide*.
- **At the function level**, you can reserve up to 900 units of concurrency across all your functions by default. 100 units of concurrency are always reserved for functions that don't explicitly reserve concurrency. For example, if you increased your account concurrency limit to 2,000, you can reserve up to 1,900 units of concurrency at the function level.

For an initial burst of traffic, your functions' cumulative concurrency in a Region can reach an initial level of between 500 and 3000:

Burst concurrency quotas

- **3000** – US West (Oregon), US East (N. Virginia), Europe (Ireland)
- **1000** – Asia Pacific (Tokyo), Europe (Frankfurt), US East (Ohio)

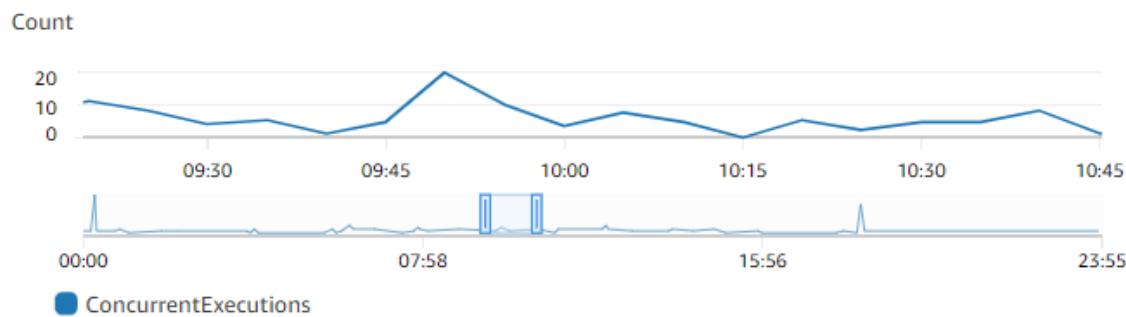
- **500 – Other Regions**

If your account concurrency limit is lower than the burst concurrency limit, Lambda limits your burst concurrency according to your account concurrency limit. For example, if your account limit is 1,000, then functions in US East (N. Virginia) get a burst concurrency of 1,000.

After the initial burst, your functions' concurrency can scale by an additional 500 instances per minute. For more information on burst concurrency, see [the section called "Burst concurrency" \(p. 220\)](#).

Accurately estimating required concurrency

If your function is currently serving traffic, you can easily view its concurrency metrics using [CloudWatch metrics](#). Specifically, the `ConcurrentExecutions` metric shows you the number of concurrent invocations for each function in your account.



From the graph, this function serves an average of 5 to 10 concurrent requests, and peaks at 20 requests on a typical day. Suppose that there are many other functions in your account. If this function is critical to your application and you don't want any dropped requests, you might use 20 as your reserved concurrency setting.

Recall that you can also calculate concurrency using the following formula:

`Concurrency = (average requests per second) * (average request duration in seconds)`

You can estimate average requests per second using the `Invocation` metric, and the average request duration in seconds using the `Duration` metric. See [Working with Lambda function metrics \(p. 870\)](#) for more details.

Note

If you choose provisioned concurrency, Lambda suggests including a 10% buffer on top of the amount of concurrency your function typically needs. Over-provisioning by 10% ensures that your function can always handle incoming requests using provisioned concurrency, even if you get slightly more traffic than expected. For example, if your function usually peaks at 200 concurrent requests, you might want to set your provisioned concurrency at 220 instead (200 concurrent requests + 10% = 220 provisioned concurrency).

Concurrency metrics

You can use the following metrics to monitor concurrency for your Lambda functions.

- `ConcurrentExecutions` – The number of currently active concurrent invocations.
- `UnreservedConcurrentExecutions` – The number of currently active concurrent invocations that are using unreserved concurrency.

- **ProvisionedConcurrentExecutions** – The number of execution environment instances that are processing events on provisioned concurrency. For each invocation of an alias or version with provisioned concurrency, Lambda emits the current count.
- **ProvisionedConcurrencyInvocations** – The number of times Lambda invokes your function code using provisioned concurrency.
- **ProvisionedConcurrencySpilloverInvocations** – The number of times Lambda invokes your function code on standard (reserved or unreserved) concurrency when all provisioned concurrency is in use.
- **ProvisionedConcurrencyUtilization** – For a version or alias, the value of ProvisionedConcurrentExecutions divided by the total amount of provisioned concurrency allocated. For example, .5 indicates that 50 percent of allocated provisioned concurrency is in use.

Configuring reserved concurrency

In Lambda, [concurrency \(p. 197\)](#) is the number of requests your function can handle at the same time. There are two types of concurrency controls available:

- Reserved concurrency – Reserved concurrency guarantees the maximum number of concurrent instances for the function. When a function has reserved concurrency, no other function can use that concurrency. There is no charge for configuring reserved concurrency for a function.
- Provisioned concurrency – Provisioned concurrency initializes a requested number of execution environments so that they are prepared to respond immediately to your function's invocations. Note that configuring provisioned concurrency incurs charges to your AWS account.

This topic details how to manage and configure reserved concurrency. If you want to decrease latency for your functions, use [provisioned concurrency \(p. 213\)](#).

Concurrency is the number of requests that your function is serving at any given time. When your function is invoked, Lambda allocates an instance of it to process the event. When the function code finishes running, it can handle another request. If the function is invoked again while a request is still being processed, another instance is allocated, which increases the function's concurrency. The total concurrency for all of the functions in your account is subject to a per-region quota.

Sections

- [Configuring reserved concurrency \(p. 210\)](#)
- [Configuring concurrency with the Lambda API \(p. 211\)](#)

Configuring reserved concurrency

To manage reserved concurrency settings for a function, use the Lambda console.

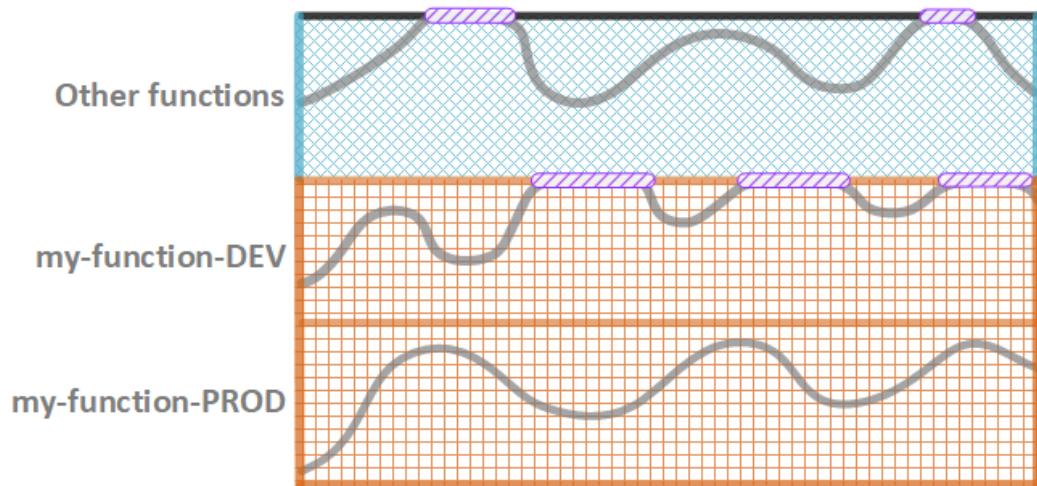
To reserve concurrency for a function

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Concurrency**.
4. Under **Concurrency**, choose **Edit**.
5. Choose **Reserve concurrency**. Enter the amount of concurrency to reserve for the function.
6. Choose **Save**.

You can reserve up to the **Unreserved account concurrency** value that is shown, minus 100 for functions that don't have reserved concurrency. To throttle a function, set the reserved concurrency to zero. This stops any events from being processed until you remove the limit.

The following example shows two functions with pools of reserved concurrency, and the unreserved concurrency pool used by other functions. Throttling errors occur when all of the concurrency in a pool is in use.

Reserved Concurrency



Legend

- Function concurrency
 -  Reserved concurrency
 -  Unreserved concurrency
 -  Throttling

Reserving concurrency has the following effects.

- **Other functions can't prevent your function from scaling** – All of your account's functions in the same Region without reserved concurrency share the pool of unreserved concurrency. Without reserved concurrency, other functions can use up all of the available concurrency. This prevents your function from scaling up when needed.
 - **Your function can't scale out of control** – Reserved concurrency also limits your function from using concurrency from the unreserved pool, which caps its maximum concurrency. You can reserve concurrency to prevent your function from using all the available concurrency in the Region, or from overloading downstream resources.

Setting per-function concurrency can impact the concurrency pool that is available to other functions. To avoid issues, limit the number of users who can use the `PutFunctionConcurrency` and `DeleteFunctionConcurrency` API operations.

Configuring concurrency with the Lambda API

To manage concurrency settings the AWS CLI or AWS SDK, use the following API operations.

- PutFunctionConcurrency (p. 1327)

- [GetFunctionConcurrency](#)
- [DeleteFunctionConcurrency \(p. 1197\)](#)
- [GetAccountSettings \(p. 1207\)](#)

To configure reserved concurrency with the AWS CLI, use the `put-function-concurrency` command. The following command reserves a concurrency of 100 for a function named `my-function`:

```
aws lambda put-function-concurrency --function-name my-function --reserved-concurrent-executions 100
```

You should see the following output:

```
{  
    "ReservedConcurrentExecutions": 100  
}
```

Configuring provisioned concurrency

In Lambda, [concurrency \(p. 197\)](#) is the number of requests your function can handle at the same time. There are two types of concurrency controls available:

- Reserved concurrency – Reserved concurrency guarantees the maximum number of concurrent instances for the function. When a function has reserved concurrency, no other function can use that concurrency. There is no charge for configuring reserved concurrency for a function.
- Provisioned concurrency – Provisioned concurrency initializes a requested number of execution environments so that they are prepared to respond immediately to your function's invocations. Note that configuring provisioned concurrency incurs charges to your AWS account.

This topic details how to manage and configure provisioned concurrency. If you want to configure reserved concurrency, see [Managing Lambda reserved concurrency \(p. 210\)](#).

When Lambda allocates an instance of your function, the runtime loads your function's code and runs initialization code that you define outside of the handler. If your code and dependencies are large, or you create SDK clients during initialization, this process can take some time. When your function has not been used for some time, needs to scale up, or when you update a function, Lambda creates new execution environments. This causes the portion of requests that are served by new instances to have higher latency than the rest, otherwise known as a cold start.

By allocating provisioned concurrency before an increase in invocations, you can ensure that all requests are served by initialized instances with low latency. Lambda functions configured with provisioned concurrency run with consistent start-up latency, making them ideal for building interactive mobile or web backends, latency sensitive microservices, and synchronously invoked APIs.

Note

Provisioned concurrency counts towards a function's reserved concurrency and [Regional quotas \(p. 1131\)](#). If the amount of provisioned concurrency on a function's versions and aliases adds up to the function's reserved concurrency, all invocations run on provisioned concurrency. This configuration also has the effect of throttling the unpublished version of the function (\$LATEST), which prevents it from executing. You can't allocate more provisioned concurrency than reserved concurrency for a function.

Lambda also integrates with Application Auto Scaling, allowing you to manage provisioned concurrency on a schedule or based on utilization.

Sections

- [Differences between Lambda SnapStart and provisioned concurrency \(p. 213\)](#)
- [Configuring provisioned concurrency \(p. 214\)](#)
- [Optimizing latency with provisioned concurrency \(p. 215\)](#)
- [Managing provisioned concurrency with Application Auto Scaling \(p. 217\)](#)

Differences between Lambda SnapStart and provisioned concurrency

[Lambda SnapStart \(p. 992\)](#) is a performance optimization that helps you improve startup performance for latency-sensitive applications by up to 10x at no extra cost. Provisioned concurrency keeps functions initialized and ready to respond in double-digit milliseconds. Configuring provisioned concurrency incurs charges to your AWS account. Use provisioned concurrency if your application has strict cold start latency requirements. You can't use both SnapStart and provisioned concurrency on the same function version.

Configuring provisioned concurrency

To manage provisioned concurrency settings for a version or alias, use the Lambda console. You can configure provisioned concurrency on a version of a function, or on an alias.

Each version of a function can only have one provisioned concurrency configuration. This can be directly on the version itself, or on an alias that points to the version. Two aliases can't allocate provisioned concurrency for the same version.

If you change the version that an alias points to, Lambda deallocates the provisioned concurrency from the old version and allocates it to the new version. You can add a routing configuration to an alias that has provisioned concurrency. For more information, see [Lambda function aliases \(p. 85\)](#). Note that you can't manage provisioned concurrency settings on the alias while the routing configuration is in place.

Note

Provisioned Concurrency is not supported on the unpublished version of the function (\$LATEST). Ensure your client application is not pointing to \$LATEST before configuring provisioned concurrency.

To allocate provisioned concurrency for an alias or version

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Concurrency**.
4. Under **Provisioned concurrency configurations**, choose **Add configuration**.
5. Choose an alias or version.
6. Enter the amount of provisioned concurrency to allocate.
7. Choose **Save**.

You can also configure provisioned concurrency using the Lambda API with the following operations:

- [PutProvisionedConcurrencyConfig](#)
- [GetProvisionedConcurrencyConfig](#)
- [ListProvisionedConcurrencyConfigs](#)
- [DeleteProvisionedConcurrencyConfig](#)

To allocate provisioned concurrency for a function, use `put-provisioned-concurrency-config`. The following command allocates a concurrency of 100 for the BLUE alias of a function named my-function:

```
aws lambda put-provisioned-concurrency-config --function-name my-function \
--qualifier BLUE --provisioned-concurrent-executions 100
```

You should see the following output:

```
{  
  "Requested_ProvisionedConcurrentExecutions": 100,  
  "Allocated_ProvisionedConcurrentExecutions": 0,  
  "Status": "IN_PROGRESS",  
  "LastModified": "2019-11-21T19:32:12+0000"  
}
```

To view your account's concurrency quotas in a Region, use `get-account-settings`.

```
aws lambda get-account-settings
```

You should see the following output:

```
{  
    "AccountLimit": {  
        "TotalCodeSize": 80530636800,  
        "CodeSizeUnzipped": 262144000,  
        "CodeSizeZipped": 52428800,  
        "ConcurrentExecutions": 1000,  
        "UnreservedConcurrentExecutions": 900  
    },  
    "AccountUsage": {  
        "TotalCodeSize": 174913095,  
        "FunctionCount": 52  
    }  
}
```

You can manage provisioned concurrency for all aliases and versions from the function configuration page. The list of provisioned concurrency configurations shows the allocation progress of each configuration. Provisioned concurrency settings are also available on the configuration page for each version and alias.

Lambda emits the following metrics for provisioned concurrency:

Provisioned concurrency metrics

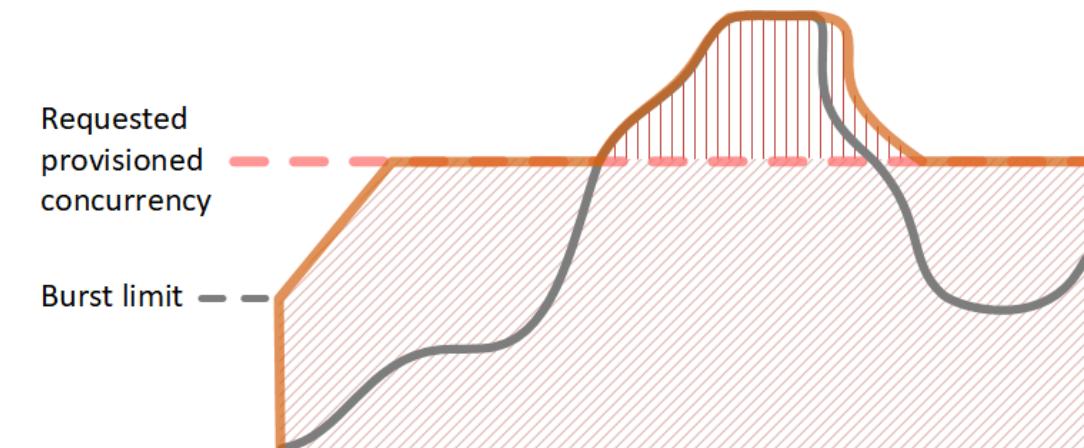
- `ProvisionedConcurrentExecutions` – The number of function instances that are processing events on provisioned concurrency. For each invocation of an alias or version with provisioned concurrency, Lambda emits the current count.
- `ProvisionedConcurrencyInvocations` – The number of times your function code is executed on provisioned concurrency.
- `ProvisionedConcurrencySpilloverInvocations` – The number of times your function code is executed on standard concurrency when all provisioned concurrency is in use.
- `ProvisionedConcurrencyUtilization` – For a version or alias, the value of `ProvisionedConcurrentExecutions` divided by the total amount of provisioned concurrency allocated. For example, .5 indicates that 50 percent of allocated provisioned concurrency is in use.

For more details, see [Working with Lambda function metrics \(p. 870\)](#).

Optimizing latency with provisioned concurrency

Provisioned concurrency does not come online immediately after you configure it. Lambda starts allocating provisioned concurrency after a minute or two of preparation. Similar to how functions [scale under load \(p. 220\)](#), up to 3000 instances of the function can be initialized at once, depending on the Region. After the initial burst, instances are allocated at a steady rate of 500 per minute until the request is fulfilled. When you request provisioned concurrency for multiple functions or versions of a function in the same Region, scaling quotas apply across all requests.

Function Scaling with Provisioned Concurrency



Legend

- Function instances
- Open requests
- Provisioned concurrency
- Standard concurrency

To optimize latency, you can customize the initialization behavior for functions that use provisioned concurrency. You can run initialization code for provisioned concurrency instances without impacting latency, because the initialization code runs at allocation time. However, the initialization code for an on-demand instance directly impacts the latency of the first invocation. For an on-demand instance, you may choose to defer initialization for a specific capability until the function needs that capability.

To determine the type of initialization, check the value of `AWS_LAMBDA_INITIALIZATION_TYPE`. Lambda sets this environment variable to provisioned-concurrency or on-demand. The value of `AWS_LAMBDA_INITIALIZATION_TYPE` is immutable and does not change over the lifetime of the execution environment.

If you use the .NET 3.1 runtime, you can configure the `AWS_LAMBDA_DOTNET_PREJIT` environment variable to improve the latency for functions that use provisioned concurrency. The .NET runtime lazily compiles and initializes each library that your code calls for the first time. As a result, the first invocation of a Lambda function can take longer than subsequent invocations. When you set `AWS_LAMBDA_DOTNET_PREJIT` to `ProvisionedConcurrency`, Lambda performs ahead-of-time JIT compilation for common system dependencies. Lambda performs this initialization optimization for provisioned concurrency instances only, which results in faster performance for the first invocation. If you set the environment variable to `Always`, Lambda performs ahead-of-time JIT compilation for every initialization. If you set the environment variable to `Never`, ahead-of-time JIT compilation is disabled. The default value for `AWS_LAMBDA_DOTNET_PREJIT` is `ProvisionedConcurrency`.

For provisioned concurrency instances, your function's [initialization code \(p. 13\)](#) runs during allocation and every few hours, as running instances of your function are recycled. You can see the initialization time in logs and [traces \(p. 807\)](#) after an instance processes a request. However, initialization is billed even if the instance never processes a request. Provisioned concurrency runs continually and is billed separately from initialization and invocation costs. For details, see [AWS Lambda pricing](#).

For more information on optimizing functions using provisioned concurrency, see the [Lambda Operator Guide](#).

Managing provisioned concurrency with Application Auto Scaling

Application Auto Scaling allows you to manage provisioned concurrency on a schedule or based on utilization. Use a target tracking scaling policy if want your function to maintain a specified utilization percentage, and scheduled scaling to increase provisioned concurrency in anticipation of peak traffic.

Target tracking

With target tracking, Application Auto Scaling creates and manages the CloudWatch alarms that trigger a scaling policy and calculates the scaling adjustment based on a metric and target value that you define. This is ideal for applications that don't have a scheduled time of increased traffic, but have certain traffic patterns.

To increase provisioned concurrency automatically as needed, use the `RegisterScalableTarget` and `PutScalingPolicy` Application Auto Scaling API operations to register a target and create a scaling policy:

1. Register a function's alias as a scaling target. The following example registers the BLUE alias of a function named `my-function`:

```
aws application-autoscaling register-scaling-target --service-namespace lambda \
--resource-id function:my-function:BLUE --min-capacity 1 --max-
capacity 100 \
--scalable-dimension lambda:function:ProvisionedConcurrency
```

2. Apply a scaling policy to the target. The following example configures Application Auto Scaling to adjust the provisioned concurrency configuration for an alias to keep utilization near 70 percent.

```
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" }}'
```

You should see the following output:

```
{
  "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"
```

```

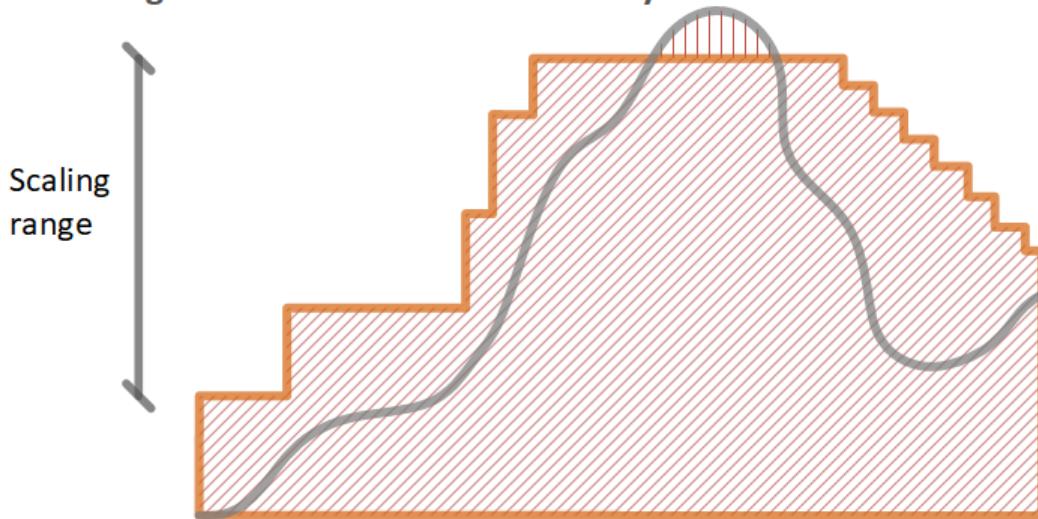
    },
    {
        "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66",
        "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66"
    }
]
}

```

Application Auto Scaling creates two alarms in CloudWatch. The first alarm triggers when the utilization of provisioned concurrency consistently exceeds 70 percent. When this happens, Application Auto Scaling allocates more provisioned concurrency to reduce utilization. The second alarm triggers when utilization is consistently less than 63 percent (90 percent of the 70 percent target). When this happens, Application Auto Scaling reduces the alias's provisioned concurrency.

In the following example, a function scales between a minimum and maximum amount of provisioned concurrency based on utilization. When the number of open requests increases, Application Auto Scaling increases provisioned concurrency in large steps until it reaches the configured maximum. The function continues to scale on standard concurrency until utilization starts to drop. When utilization is consistently low, Application Auto Scaling decreases provisioned concurrency in smaller periodic steps.

Autoscaling with Provisioned Concurrency



Legend

- Function instances
- Open requests
- Provisioned concurrency
- Standard concurrency

Both of these alarms use the *average* statistic by default. Functions that have traffic patterns of quick bursts may not trigger your provisioned concurrency to scale up. For example, if your Lambda function executes quickly (20–100 ms) and your traffic pattern comes in quick bursts, this may cause incoming requests to exceed your allocated provisioned concurrency during the burst, but if the burst doesn't last 3 minutes, auto scaling will not trigger. Additionally, if CloudWatch doesn't get three data points that hit the target average, the auto scaling policy will not trigger.

For more information on target tracking scaling policies, see [Target tracking scaling policies for Application Auto Scaling](#).

Scheduled scaling

Scaling based on a schedule allows you to set your own scaling schedule according to predictable load changes. For more information and examples, see [Scheduling AWS Lambda Provisioned Concurrency for recurring peak usage](#).

Burst concurrency

In Lambda, [concurrency \(p. 197\)](#) is the number of requests your function can handle at the same time. By default, Lambda provides your account with a total concurrency limit of 1,000 across all functions in a region.

For an initial burst of traffic, your functions' cumulative concurrency in a Region can reach an initial level of between 500 and 3000, which varies per Region. Note that the burst concurrency quota is not per-function; it applies to all your functions in the Region.

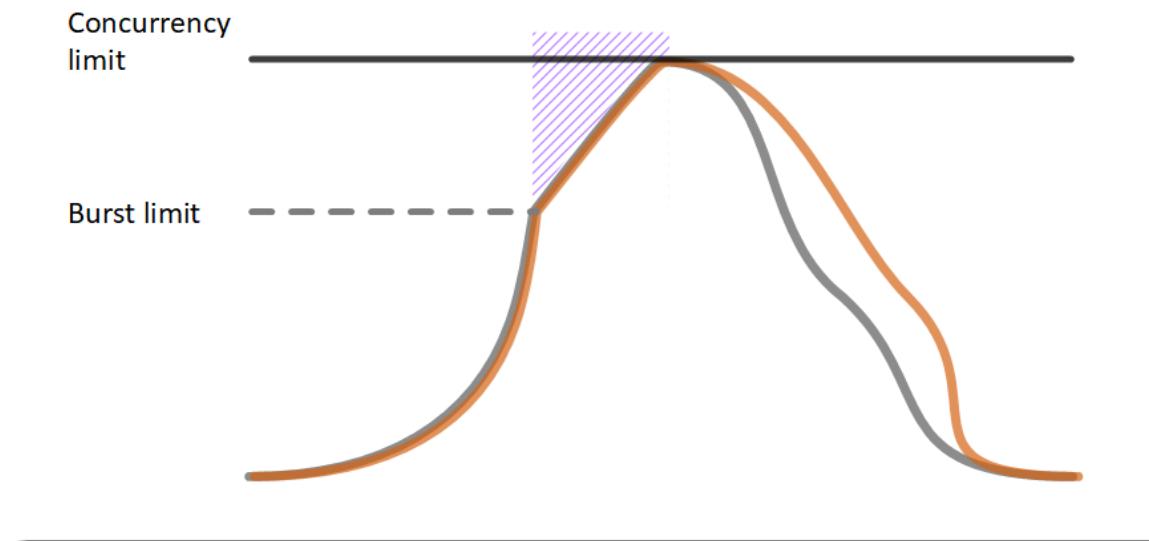
Burst concurrency quotas

- **3000** – US West (Oregon), US East (N. Virginia), Europe (Ireland)
- **1000** – Asia Pacific (Tokyo), Europe (Frankfurt), US East (Ohio)
- **500** – Other Regions

After the initial burst, your functions' concurrency can scale by an additional 500 instances each minute. This continues until there are enough instances to serve all requests, or until a concurrency limit is reached. When requests come in faster than your function can scale, or when your function is at maximum concurrency, additional requests fail with a throttling error (429 status code).

The following example shows a function processing a spike in traffic. As invocations increase exponentially, the function scales up. It initializes a new instance for any request that can't be routed to an available instance. When the burst concurrency limit is reached, the function starts to scale linearly. If this isn't enough concurrency to serve all requests, additional requests are throttled and should be retried.

Function Scaling with Concurrency Limit



Legend

- Function instances
- Open requests

-  Throttling possible

The function continues to scale until the account's concurrency limit for the function's Region is reached. The function catches up to demand, requests subside, and unused instances of the function are stopped after being idle for some time. Unused instances are frozen while they're waiting for requests and don't incur any charges.

Configuring a Lambda function to access resources in a VPC

You can configure a Lambda function to connect to private subnets in a virtual private cloud (VPC) in your AWS account. Use Amazon Virtual Private Cloud (Amazon VPC) to create a private network for resources such as databases, cache instances, or internal services. Connect your function to the VPC to access private resources while the function is running. This section provides a summary of Lambda VPC connections. For details about VPC networking in Lambda, see [the section called "Private networking" \(p. 89\)](#).

When you connect a function to a VPC, Lambda assigns your function to a Hyperplane ENI (elastic network interface) for each subnet in your function's VPC configuration. Lambda creates a Hyperplane ENI the first time a unique subnet and security group combination is defined for a VPC-enabled function in an account.

While Lambda creates a Hyperplane ENI, you can't perform additional operations that target the function, such as [creating versions \(p. 83\)](#) or updating the function's code. For new functions, you can't invoke the function until its state changes from Pending to Active. For existing functions, you can still invoke an earlier version while the update is in progress. For details about the Hyperplane ENI lifecycle, see [the section called "Lambda Hyperplane ENIs" \(p. 90\)](#).

Lambda functions can't connect directly to a VPC with [dedicated instance tenancy](#). To connect to resources in a dedicated VPC, [peer it to a second VPC with default tenancy](#).

Sections

- [Managing VPC connections \(p. 222\)](#)
- [Execution role and user permissions \(p. 223\)](#)
- [Configuring VPC access \(console\) \(p. 223\)](#)
- [Configuring VPC access \(API\) \(p. 224\)](#)
- [Using IAM condition keys for VPC settings \(p. 225\)](#)
- [Internet and service access for VPC-connected functions \(p. 228\)](#)
- [VPC tutorials \(p. 228\)](#)
- [Sample VPC configurations \(p. 228\)](#)

Managing VPC connections

Multiple functions can share a network interface, if the functions share the same subnet and security group. Connecting additional functions to the same VPC configuration (subnet and security group) that has an existing Lambda-managed network interface is much quicker than creating a new network interface.

If your functions aren't active for a long period of time, Lambda reclaims its network interfaces, and the functions become Idle. To reactivate an idle function, invoke it. This invocation fails, and the function enters a Pending state again until a network interface is available.

If you update your function to access a different VPC, it terminates connectivity from the Hyperplane ENI to the previous VPC. The process to update the connectivity to a new VPC can take several minutes. During this time, Lambda connects function invocations to the previous VPC. After the update is complete, new invocations start using the new VPC and the Lambda function is no longer connected to the older VPC.

For short-lived operations, such as DynamoDB queries, the latency overhead of setting up a TCP connection might be greater than the operation itself. To ensure connection reuse for short-lived/infrequently invoked functions, we recommend that you use *TCP keep-alive* for connections that were created during your function initialization, to avoid creating new connections for subsequent invokes. For more information on reusing connections using keep-alive, refer to [Lambda documentation on reusing connections](#).

Execution role and user permissions

Lambda uses your function's permissions to create and manage network interfaces. To connect to a VPC, your function's [execution role \(p. 816\)](#) must have the following permissions:

Execution role permissions

- **ec2:CreateNetworkInterface**
- **ec2:DescribeNetworkInterfaces** – This action only works if it's allowed on all resources ("Resource": "/*").
- **ec2:DeleteNetworkInterface** – If you don't specify a resource ID for **DeleteNetworkInterface** in the execution role, your function may not be able to access the VPC. Either specify a unique resource ID, or include all resource IDs, for example, "Resource": "arn:aws:ec2:us-west-2:123456789012:/*/*".

These permissions are included in the AWS managed policy **AWSLambdaVPCAccessExecutionRole**. Note that these permissions are required only to create ENIs, not to invoke your VPC function. In other words, you are still able to invoke your VPC function successfully even if you remove these permissions from your execution role. To completely disassociate your Lambda function from the VPC, update the function's VPC configuration settings using the console or the [UpdateFunctionConfiguration \(p. 1377\)](#) API.

When you configure VPC connectivity, Lambda uses your permissions to verify network resources. To configure a function to connect to a VPC, your user needs the following permissions:

User permissions

- **ec2:DescribeSecurityGroups**
- **ec2:DescribeSubnets**
- **ec2:DescribeVpcs**

Configuring VPC access (console)

If your [IAM permissions \(p. 225\)](#) allow you only to create Lambda functions that connect to your VPC, you must configure the VPC when you create the function. If your IAM permissions allow you to create functions that aren't connected to your VPC, you can add the VPC configuration after you create the function.

To configure a VPC when you create a function

1. Open the [Functions page](#) of the Lambda console.
2. Choose **Create function**.
3. Under **Basic information**, for **Function name**, enter a name for your function.
4. Expand **Advanced settings**.
5. Under **Network**, choose a **VPC** for your function to access.

6. Choose subnets and security groups. When you choose a security group, the console displays the inbound and outbound rules for that security group.

Note

To access private resources, connect your function to private subnets. If your function needs internet access, use [network address translation \(NAT\) \(p. 228\)](#). Connecting a function to a public subnet doesn't give it internet access or a public IP address.

7. Choose **Create function**.

To configure a VPC for an existing function

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **VPC**.
4. Under **VPC**, choose **Edit**.
5. Choose a VPC, subnets, and security groups.

Note

To access private resources, connect your function to private subnets. If your function needs internet access, use [network address translation \(NAT\) \(p. 228\)](#). Connecting a function to a public subnet doesn't give it internet access or a public IP address.

6. Choose **Save**.

Configuring VPC access (API)

To connect a Lambda function to a VPC, you can use the following API operations:

- [CreateFunction \(p. 1165\)](#)
- [UpdateFunctionConfiguration \(p. 1377\)](#)

To create a function and connect it to a VPC using the AWS Command Line Interface (AWS CLI), you can use the `create-function` command with the `vpc-config` option. The following example creates a function with a connection to a VPC with two subnets and one security group.

```
aws lambda create-function --function-name my-function \
--runtime nodejs18.x --handler index.js --zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/Lambda-role \
--vpc-config
SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a036,SecurityGroupIds=sg-085912345678492fb
```

To connect an existing function to a VPC, use the `update-function-configuration` command with the `vpc-config` option.

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config
SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a036,SecurityGroupIds=sg-085912345678492fb
```

To disconnect your function from a VPC, update the function configuration with an empty list of subnets and security groups.

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config SubnetIds=[],SecurityGroupIds=[]
```

Using IAM condition keys for VPC settings

You can use Lambda-specific condition keys for VPC settings to provide additional permission controls for your Lambda functions. For example, you can require that all functions in your organization are connected to a VPC. You can also specify the subnets and security groups that the function's users can and can't use.

Lambda supports the following condition keys in IAM policies:

- **lambda:VpcIds** – Allow or deny one or more VPCs.
- **lambda:SubnetIds** – Allow or deny one or more subnets.
- **lambda:SecurityGroupIds** – Allow or deny one or more security groups.

The Lambda API operations [CreateFunction \(p. 1165\)](#) and [UpdateFunctionConfiguration \(p. 1377\)](#) support these condition keys. For more information about using condition keys in IAM policies, see [IAM JSON Policy Elements: Condition](#) in the *IAM User Guide*.

Tip

If your function already includes a VPC configuration from a previous API request, you can send an `UpdateFunctionConfiguration` request without the VPC configuration.

Example policies with condition keys for VPC settings

The following examples demonstrate how to use condition keys for VPC settings. After you create a policy statement with the desired restrictions, append the policy statement for the target user or role.

Ensure that users deploy only VPC-connected functions

To ensure that all users deploy only VPC-connected functions, you can deny function create and update operations that don't include a valid VPC ID.

Note that VPC ID is not an input parameter to the `CreateFunction` or `UpdateFunctionConfiguration` request. Lambda retrieves the VPC ID value based on the subnet and security group parameters.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "EnforceVPCFunction",  
            "Action": [  
                "lambda:CreateFunction",  
                "lambda:UpdateFunctionConfiguration"  
            ],  
            "Effect": "Deny",  
            "Resource": "*",  
            "Condition": {  
                "Null": {  
                    "lambda:VpcIds": "true"  
                }  
            }  
        }  
    ]  
}
```

Deny users access to specific VPCs, subnets, or security groups

To deny users access to specific VPCs, use `StringEquals` to check the value of the `lambda:VpcIds` condition. The following example denies users access to `vpc-1` and `vpc-2`.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "EnforceOutOfVPC",  
            "Action": [  
                "lambda:CreateFunction",  
                "lambda:UpdateFunctionConfiguration"  
            ],  
            "Effect": "Deny",  
            "Resource": "*",  
            "Condition": {  
                "StringEquals": {  
                    "lambda:VpcIds": ["vpc-1", "vpc-2"]  
                }  
            }  
        }  
    ]  
}
```

To deny users access to specific subnets, use `StringEquals` to check the value of the `lambda:SubnetIds` condition. The following example denies users access to `subnet-1` and `subnet-2`.

```
{  
    "Sid": "EnforceOutOfSubnet",  
    "Action": [  
        "lambda:CreateFunction",  
        "lambda:UpdateFunctionConfiguration"  
    ],  
    "Effect": "Deny",  
    "Resource": "*",  
    "Condition": {  
        "ForAnyValue:StringEquals": {  
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]  
        }  
    }  
}
```

To deny users access to specific security groups, use `StringEquals` to check the value of the `lambda:SecurityGroupIds` condition. The following example denies users access to `sg-1` and `sg-2`.

```
{  
    "Sid": "EnforceOutOfSecurityGroups",  
    "Action": [  
        "lambda:CreateFunction",  
        "lambda:UpdateFunctionConfiguration"  
    ],  
    "Effect": "Deny",  
    "Resource": "*",  
    "Condition": {  
        "ForAnyValue:StringEquals": {  
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]  
        }  
    }  
}
```

Allow users to create and update functions with specific VPC settings

To allow users to access specific VPCs, use `StringEquals` to check the value of the `lambda:VpcIds` condition. The following example allows users to access `vpc-1` and `vpc-2`.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "EnforceStayInSpecificVpc",  
            "Action": [  
                "lambda>CreateFunction",  
                "lambda:UpdateFunctionConfiguration"  
            ],  
            "Effect": "Allow",  
            "Resource": "*",  
            "Condition": {  
                "StringEquals": {  
                    "lambda:VpcIds": ["vpc-1", "vpc-2"]  
                }  
            }  
        }  
    ]  
}
```

To allow users to access specific subnets, use `StringEquals` to check the value of the `lambda:SubnetIds` condition. The following example allows users to access `subnet-1` and `subnet-2`.

```
{  
    "Sid": "EnforceStayInSpecificSubnets",  
    "Action": [  
        "lambda>CreateFunction",  
        "lambda:UpdateFunctionConfiguration"  
    ],  
    "Effect": "Allow",  
    "Resource": "*",  
    "Condition": {  
        "ForAllValues:StringEquals": {  
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]  
        }  
    }  
}
```

To allow users to access specific security groups, use `StringEquals` to check the value of the `lambda:SecurityGroupIds` condition. The following example allows users to access `sg-1` and `sg-2`.

```
{  
    "Sid": "EnforceStayInSpecificSecurityGroup",  
    "Action": [  
        "lambda>CreateFunction",  
        "lambda:UpdateFunctionConfiguration"  
    ],  
    "Effect": "Allow",  
    "Resource": "*",  
    "Condition": {  
        "ForAllValues:StringEquals": {  
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]  
        }  
    }  
}
```

```
    ]  
}
```

Internet and service access for VPC-connected functions

By default, Lambda runs your functions in a secure VPC with access to AWS services and the internet. Lambda owns this VPC, which isn't connected to your account's [default VPC](#). When you connect a function to a VPC in your account, the function can't access the internet unless your VPC provides access.

Note

Several AWS services offer [VPC endpoints](#). You can use VPC endpoints to connect to AWS services from within a VPC without internet access.

Internet access from a private subnet requires network address translation (NAT). To give your function access to the internet, route outbound traffic to a [NAT gateway](#) in a public subnet. The NAT gateway has a public IP address and can connect to the internet through the VPC's internet gateway. An idle NAT gateway connection will [time out after 350 seconds](#). For more information, see [How do I give internet access to my Lambda function in a VPC?](#)

VPC tutorials

In the following tutorials, you connect a Lambda function to resources in your VPC.

- [Tutorial: Using a Lambda function to access Amazon RDS in an Amazon VPC \(p. 728\)](#)
- [Tutorial: Configuring a Lambda function to access Amazon ElastiCache in an Amazon VPC \(p. 660\)](#)

Sample VPC configurations

You can use the following sample AWS CloudFormation templates to create VPC configurations to use with Lambda functions. There are two templates available in this guide's GitHub repository:

- [vpc-private.yaml](#) – A VPC with two private subnets and VPC endpoints for Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB. Use this template to create a VPC for functions that don't need internet access. This configuration supports use of Amazon S3 and DynamoDB with the AWS SDKs, and access to database resources in the same VPC over a local network connection.
- [vpc-privatepublic.yaml](#) – A VPC with two private subnets, VPC endpoints, a public subnet with a NAT gateway, and an internet gateway. Internet-bound traffic from functions in the private subnets is routed to the NAT gateway using a route table.

To create a VPC using a template, on the AWS CloudFormation console [Stacks page](#), choose **Create stack**, and then follow the instructions in the **Create stack** wizard.

Configuring interface VPC endpoints for Lambda

If you use Amazon Virtual Private Cloud (Amazon VPC) to host your AWS resources, you can establish a connection between your VPC and Lambda. You can use this connection to invoke your Lambda function without crossing the public internet.

To establish a private connection between your VPC and Lambda, create an [interface VPC endpoint](#). Interface endpoints are powered by [AWS PrivateLink](#), which enables you to privately access Lambda APIs without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to communicate with Lambda APIs. Traffic between your VPC and Lambda does not leave the AWS network.

Each interface endpoint is represented by one or more [elastic network interfaces](#) in your subnets. A network interface provides a private IP address that serves as an entry point for traffic to Lambda.

Sections

- [Considerations for Lambda interface endpoints \(p. 229\)](#)
- [Creating an interface endpoint for Lambda \(p. 230\)](#)
- [Creating an interface endpoint policy for Lambda \(p. 230\)](#)

Considerations for Lambda interface endpoints

Before you set up an interface endpoint for Lambda, be sure to review [Interface endpoint properties and limitations](#) in the *Amazon VPC User Guide*.

You can call any of the Lambda API operations from your VPC. For example, you can invoke the Lambda function by calling the Invoke API from within your VPC. For the full list of Lambda APIs, see [Actions](#) in the Lambda API reference.

use1-az3 is a limited capacity Region for Lambda VPC functions. You shouldn't use subnets in this availability zone with your Lambda functions because this can result in reduced zonal redundancy in the event of an outage.

Keep-alive for persistent connections

Lambda purges idle connections over time, so you must use a keep-alive directive to maintain persistent connections. Attempting to reuse an idle connection when invoking a function results in a connection error. To maintain your persistent connection, use the keep-alive directive associated with your runtime. For an example, see [Reusing Connections with Keep-Alive in Node.js](#) in the *AWS SDK for JavaScript Developer Guide*.

Billing Considerations

There is no additional cost to access a Lambda function through an interface endpoint. For more Lambda pricing information, see [AWS Lambda Pricing](#).

Standard pricing for AWS PrivateLink applies to interface endpoints for Lambda. Your AWS account is billed for every hour an interface endpoint is provisioned in each Availability Zone and for data processed through the interface endpoint. For more interface endpoint pricing information, see [AWS PrivateLink pricing](#).

VPC Peering Considerations

You can connect other VPCs to the VPC with interface endpoints using [VPC peering](#). VPC peering is a networking connection between two VPCs. You can establish a VPC peering connection between

your own two VPCs, or with a VPC in another AWS account. The VPCs can also be in two different AWS Regions.

Traffic between peered VPCs stays on the AWS network and does not traverse the public internet. Once VPCs are peered, resources like Amazon Elastic Compute Cloud (Amazon EC2) instances, Amazon Relational Database Service (Amazon RDS) instances, or VPC-enabled Lambda functions in both VPCs can access the Lambda API through interface endpoints created in the one of the VPCs.

Creating an interface endpoint for Lambda

You can create an interface endpoint for Lambda using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see [Creating an interface endpoint](#) in the *Amazon VPC User Guide*.

To create an interface endpoint for Lambda (console)

1. Open the [Endpoints page](#) of the Amazon VPC console.
2. Choose **Create Endpoint**.
3. For **Service category**, verify that **AWS services** is selected.
4. For **Service Name**, choose **com.amazonaws.region.lambda**. Verify that the **Type** is **Interface**.
5. Choose a VPC and subnets.
6. To enable private DNS for the interface endpoint, select the **Enable DNS Name** check box.
7. For **Security group**, choose one or more security groups.
8. Choose **Create endpoint**.

To use the private DNS option, you must set the `enableDnsHostnames` and `enableDnsSupport` attributes of your VPC. For more information, see [Viewing and updating DNS support for your VPC](#) in the *Amazon VPC User Guide*. If you enable private DNS for the interface endpoint, you can make API requests to Lambda using its default DNS name for the Region, for example, `lambda.us-east-1.amazonaws.com`. For more service endpoints, see [Service endpoints and quotas](#) in the *AWS General Reference*.

For more information, see [Accessing a service through an interface endpoint](#) in the *Amazon VPC User Guide*.

For information about creating and configuring an endpoint using AWS CloudFormation, see the [AWS::EC2::VPCEndpoint](#) resource in the *AWS CloudFormation User Guide*.

To create an interface endpoint for Lambda (AWS CLI)

Use the `create-vpc-endpoint` command and specify the VPC ID, VPC endpoint type (interface), service name, subnets that will use the endpoint, and security groups to associate with the endpoint's network interfaces. For example:

```
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
```

Creating an interface endpoint policy for Lambda

To control who can use your interface endpoint and which Lambda functions the user can access, you can attach an endpoint policy to your endpoint. The policy specifies the following information:

- The principal that can perform actions.
- The actions that the principal can perform.
- The resources on which the principal can perform actions.

For more information, see [Controlling access to services with VPC endpoints](#) in the *Amazon VPC User Guide*.

Example: Interface endpoint policy for Lambda actions

The following is an example of an endpoint policy for Lambda. When attached to an endpoint, this policy allows user MyUser to invoke the function my-function.

Note

You need to include both the qualified and the unqualified function ARN in the resource.

```
{  
    "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:*"  
            ]  
        }  
    ]  
}
```

Configuring database access for a Lambda function

You can create an Amazon RDS Proxy database proxy for your function. A database proxy manages a pool of database connections and relays queries from a function. With a proxy, high [concurrency \(p. 11\)](#) levels can be achieved without exhausting database connections.

Sections

- [Creating a database proxy \(console\) \(p. 232\)](#)
- [Using the function's permissions for authentication \(p. 233\)](#)
- [Sample application \(p. 233\)](#)

Creating a database proxy (console)

You can use the Lambda console to create an Amazon RDS Proxy database proxy.

To create a database proxy

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Database proxies**.
4. Choose **Add database proxy**.
5. Configure the following options.
 - **Proxy identifier** – The name of the proxy.
 - **RDS DB instance** – A [supported MySQL or PostgreSQL](#) DB instance or cluster.
 - **Secret** – A Secrets Manager secret with the database user name and password.

Example secret

```
{  
    "username": "admin",  
    "password": "e2abcecxmpldc897"  
}
```

6. Choose **Add**.
- **IAM role** – An IAM role with permission to use the secret, and a trust policy that allows Amazon RDS to assume the role.
- **Authentication** – The authentication and authorization method for connecting to the proxy from your function code.

Pricing

Amazon RDS charges a hourly price for proxies that is determined by the instance size of your database. For details, see [RDS Proxy pricing](#).

Proxy creation takes a few minutes. When the proxy is available, configure your function to connect to the proxy endpoint instead of the database endpoint.

Standard [Amazon RDS Proxy pricing](#) applies. For more information, see [Managing connections with the Amazon RDS Proxy](#) in the Amazon Aurora User Guide.

Using the function's permissions for authentication

By default, you can connect to a proxy with the same username and password that it uses to connect to the database. The only difference in your function code is the endpoint that the database client connects to. The drawback of this method is that you must expose the password to your function code, either by configuring it in a secure environment variable or by retrieving it from Secrets Manager.

You can create a database proxy that uses the function's IAM credentials for authentication and authorization instead of a password. To use the function's permissions to connect to the proxy, set **Authentication to Execution role**.

The Lambda console adds the required permission (`rds-db:connect`) to the execution role. You can then use the AWS SDK to generate a token that allows it to connect to the proxy. The following example shows how to configure a database connection with the `mysql2` library in Node.js.

Example [dbadmin/index-iam.js](#) – AWS SDK signer

```
const signer = new AWS.RDS.Signer({
  region: region,
  hostname: host,
  port: sqlport,
  username: username
})

exports.handler = async (event) => {
  let connectionConfig = {
    host      : host,
    user      : username,
    database  : database,
    ssl: 'Amazon RDS',
    authPlugins: { mysql_clear_password: () => () => signer.getAuthToken() }
  }
  var connection = mysql.createConnection(connectionConfig)
  var query = event.query
  var result
  connection.connect()
}
```

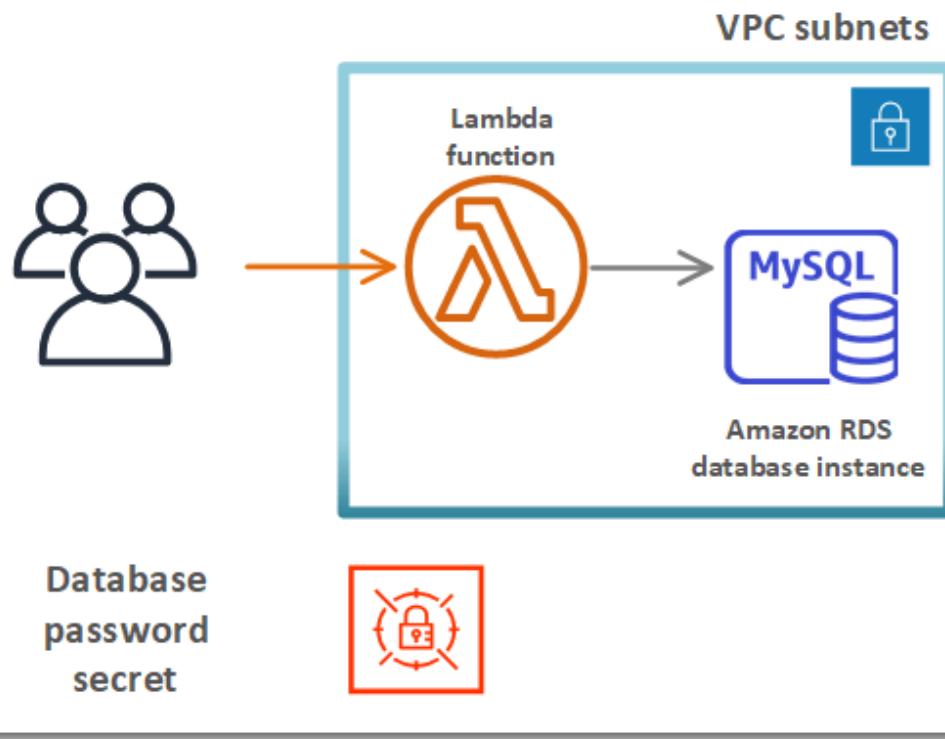
For more information, see [IAM database authentication](#) in the Amazon RDS User Guide.

Sample application

Sample applications that demonstrate the use of Lambda with an Amazon RDS database are available in this guide's GitHub repository. There are two applications:

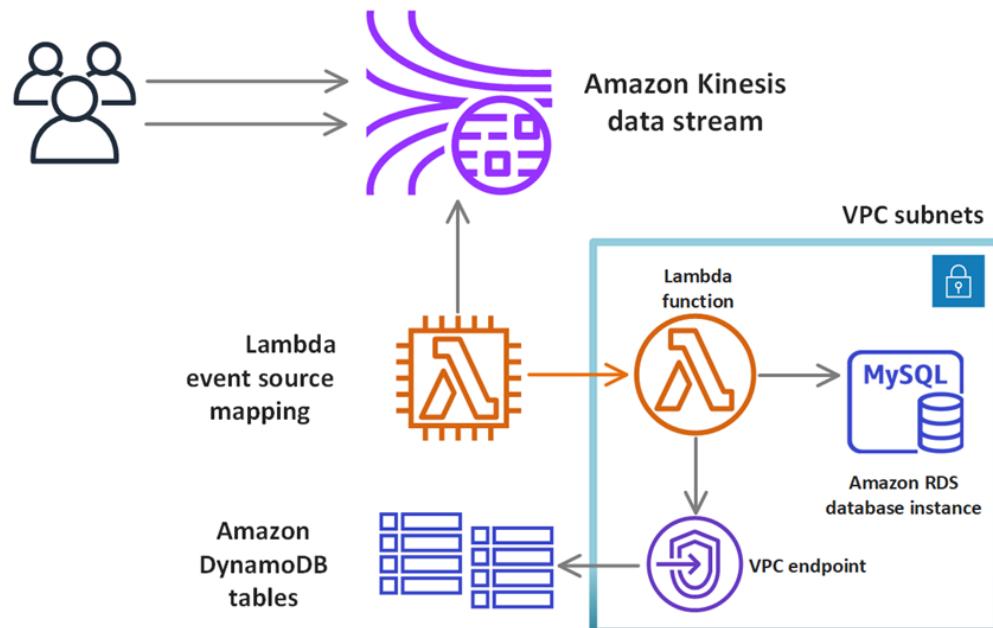
- [RDS MySQL](#) – The AWS CloudFormation template `template-vpcrds.yml` creates a MySQL 5.7 database in a private VPC. In the sample application, a Lambda function proxies queries to the database. The function and database templates both use Secrets Manager to access database credentials.

RDS MySQL Application



- [List Manager](#) – A processor function reads events from a Kinesis stream. It uses the data from the events to update DynamoDB tables, and stores a copy of the event in a MySQL database.

List manager application



To use the sample applications, follow the instructions in the GitHub repository: [RDS MySQL](#), [List Manager](#).

Configuring file system access for Lambda functions

You can configure a function to mount an Amazon Elastic File System (Amazon EFS) file system to a local directory. With Amazon EFS, your function code can access and modify shared resources safely and at high concurrency.

Sections

- [Execution role and user permissions \(p. 236\)](#)
- [Configuring a file system and access point \(p. 236\)](#)
- [Connecting to a file system \(console\) \(p. 237\)](#)
- [Configuring file system access with the Lambda API \(p. 238\)](#)
- [AWS CloudFormation and AWS SAM \(p. 238\)](#)
- [Sample applications \(p. 240\)](#)

Execution role and user permissions

If the file system doesn't have a user-configured AWS Identity and Access Management (IAM) policy, EFS uses a default policy that grants full access to any client that can connect to the file system using a file system mount target. If the file system has a user-configured IAM policy, your function's execution role must have the correct `elasticfilesystem` permissions.

Execution role permissions

- `elasticfilesystem:ClientMount`
- `elasticfilesystem:ClientWrite` (not required for read-only connections)

These permissions are included in the `AmazonElasticFileSystemClientReadWriteAccess` managed policy. Additionally, your execution role must have the [permissions required to connect to the file system's VPC \(p. 223\)](#).

When you configure a file system, Lambda uses your permissions to verify mount targets. To configure a function to connect to a file system, your user needs the following permissions:

User permissions

- `elasticfilesystem:DescribeMountTargets`

Configuring a file system and access point

Create a file system in Amazon EFS with a mount target in every Availability Zone that your function connects to. For performance and resilience, use at least two Availability Zones. For example, in a simple configuration you could have a VPC with two private subnets in separate Availability Zones. The function connects to both subnets and a mount target is available in each. Ensure that NFS traffic (port 2049) is allowed by the security groups used by the function and mount targets.

Note

When you create a file system, you choose a performance mode that can't be changed later. **General purpose** mode has lower latency, and **Max I/O** mode supports a higher maximum throughput and IOPS. For help choosing, see [Amazon EFS performance](#) in the *Amazon Elastic File System User Guide*.

An access point connects each instance of the function to the right mount target for the Availability Zone it connects to. For best performance, create an access point with a non-root path, and limit the number of files that you create in each directory. The following example creates a directory named `my-function` on the file system and sets the owner ID to 1001 with standard directory permissions (755).

Example access point configuration

- **Name** – `files`
- **User ID** – 1001
- **Group ID** – 1001
- **Path** – `/my-function`
- **Permissions** – 755
- **Owner user ID** – 1001
- **Group user ID** – 1001

When a function uses the access point, it is given user ID 1001 and has full access to the directory.

For more information, see the following topics in the *Amazon Elastic File System User Guide*:

- [Creating resources for Amazon EFS](#)
- [Working with users, groups, and permissions](#)

Connecting to a file system (console)

A function connects to a file system over the local network in a VPC. The subnets that your function connects to can be the same subnets that contain mount points for your file system, or subnets in the same Availability Zone that can route NFS traffic (port 2049) to the file system.

Note

If your function is not already connected to a VPC, see [Configuring a Lambda function to access resources in a VPC \(p. 222\)](#).

To configure file system access

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **File systems**.
4. Under **File system**, choose **Add file system**.
5. Configure the following properties:
 - **EFS file system** – The access point for a file system in the same VPC.
 - **Local mount path** – The location where the file system is mounted on the Lambda function, starting with `/mnt/`.

Pricing

Amazon EFS charges for storage and throughput, with rates that vary by storage class. For details, see [Amazon EFS pricing](#).

Lambda charges for data transfer between VPCs. This only applies if your function's VPC is peered to another VPC with a file system. The rates are the same as for Amazon EC2 data transfer between VPCs in the same Region. For details, see [Lambda pricing](#).

For more information about Lambda's integration with Amazon EFS, see [Using Amazon EFS with Lambda \(p. 666\)](#).

Configuring file system access with the Lambda API

Use the following API operations to connect your Lambda function to a file system:

- [CreateFunction \(p. 1165\)](#)
- [UpdateFunctionConfiguration \(p. 1377\)](#)

To connect a function to a file system, use the update-function-configuration command. The following example connects a function named my-function to a file system with ARN of an access point.

```
ARN=arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd
aws lambda update-function-configuration --function-name my-function \
--file-system-configs Arn=$ARN,LocalMountPath=/mnt/efs0
```

You can get the ARN of a file system's access point with the describe-access-points command.

```
aws efs describe-access-points
```

You should see the following output:

```
{
    "AccessPoints": [
        {
            "ClientToken": "console-aa50c1fd-xmpl-48b5-91ce-57b27a3b1017",
            "Name": "lambda-ap",
            "Tags": [
                {
                    "Key": "Name",
                    "Value": "lambda-ap"
                }
            ],
            "AccessPointId": "fsap-015cxmplb72b405fd",
            "AccessPointArn": "arn:aws:elasticfilesystem:us-east-2:123456789012:access-
point/fsap-015cxmplb72b405fd",
            "FileSystemId": "fs-aea3xmpl",
            "RootDirectory": {
                "Path": "/"
            },
            "OwnerId": "123456789012",
            "LifeCycleState": "available"
        }
    ]
}
```

AWS CloudFormation and AWS SAM

You can use AWS CloudFormation and the AWS Serverless Application Model (AWS SAM) to automate the creation of Lambda applications. To enable a file system connection on an AWS SAM `AWS::Serverless::Function` resource, use the `FileSystemConfigs` property.

Example template.yml – File system configuration

```
Transform: AWS::Serverless-2016-10-31
Resources:
```

```

VPC:
  Type: AWS::EC2::VPC
  Properties:
    CidrBlock: 10.0.0.0/16
Subnet1:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId:
      Ref: VPC
    CidrBlock: 10.0.1.0/24
    AvailabilityZone: "us-west-2a"
EfsSecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
    VpcId:
      Ref: VPC
    GroupDescription: "mnt target sg"
    SecurityGroupIngress:
      - IpProtocol: -1
        CidrIp: "0.0.0.0/0"
FileSystem:
  Type: AWS::EFS::FileSystem
  Properties:
    PerformanceMode: generalPurpose
AccessPoint:
  Type: AWS::EFS::AccessPoint
  Properties:
    FileSystemId:
      Ref: FileSystem
    PosixUser:
      Uid: "1001"
      Gid: "1001"
    RootDirectory:
      CreationInfo:
        OwnerGid: "1001"
        OwnerUid: "1001"
        Permissions: "755"
MountTarget1:
  Type: AWS::EFS::MountTarget
  Properties:
    FileSystemId:
      Ref: FileSystem
    SubnetId:
      Ref: Subnet1
    SecurityGroups:
      - Ref: EfsSecurityGroup
MyFunctionWithEfs:
  Type: AWS::Serverless::Function
  Properties:
    Handler: index.handler
    Runtime: python3.10
    VpcConfig:
      SecurityGroupIds:
        - Ref: EfsSecurityGroup
      SubnetIds:
        - Ref: Subnet1
    FileSystemConfigs:
      - Arn: !GetAtt AccessPoint.Arn
        LocalMountPath: "/mnt/efs"
    Description: Use a file system.
    DependsOn: "MountTarget1"

```

You must add the DependsOn to ensure that the mount targets are fully created before the Lambda runs for the first time.

For the AWS CloudFormation `AWS::Lambda::Function` type, the property name and fields are the same. For more information, see [Using AWS Lambda with AWS CloudFormation \(p. 598\)](#).

Sample applications

The GitHub repository for this guide includes a sample application that demonstrates the use of Amazon EFS with a Lambda function.

- [`efs-nodejs`](#) – A function that uses an Amazon EFS file system in a Amazon VPC. This sample includes a VPC, file system, mount targets, and access point configured for use with Lambda.

Configuring code signing for AWS Lambda

Code signing for AWS Lambda helps to ensure that only trusted code runs in your Lambda functions. When you enable code signing for a function, Lambda checks every code deployment and verifies that the code package is signed by a trusted source.

Note

Functions defined as container images do not support code signing.

To verify code integrity, use [AWS Signer](#) to create digitally signed code packages for functions and layers. When a user attempts to deploy a code package, Lambda performs validation checks on the code package before accepting the deployment. Because code signing validation checks run at deployment time, there is no performance impact on function execution.

You also use AWS Signer to create *signing profiles*. You use a signing profile to create the signed code package. Use AWS Identity and Access Management (IAM) to control who can sign code packages and create signing profiles. For more information, see [Authentication and Access Control](#) in the *AWS Signer Developer Guide*.

To enable code signing for a function, you create a *code signing configuration* and attach it to the function. A code signing configuration defines a list of allowed signing profiles and the policy action to take if any of the validation checks fail.

Lambda layers follow the same signed code package format as function code packages. When you add a layer to a function that has code signing enabled, Lambda checks that the layer is signed by an allowed signing profile. When you enable code signing for a function, all layers that are added to the function must also be signed by one of the allowed signing profiles.

Use IAM to control who can create code signing configurations. Typically, you allow only specific administrative users to have this ability. Additionally, you can set up IAM policies to enforce that developers only create functions that have code signing enabled.

You can configure code signing to log changes to AWS CloudTrail. Successful and blocked deployments to functions are logged to CloudTrail with information about the signature and validation checks.

You can configure code signing for your functions using the Lambda console, the AWS Command Line Interface (AWS CLI), AWS CloudFormation, and the AWS Serverless Application Model (AWS SAM).

There is no additional charge for using AWS Signer or code signing for AWS Lambda.

Sections

- [Signature validation \(p. 241\)](#)
- [Configuration prerequisites \(p. 242\)](#)
- [Creating code signing configurations \(p. 242\)](#)
- [Updating a code signing configuration \(p. 242\)](#)
- [Deleting a code signing configuration \(p. 243\)](#)
- [Enabling code signing for a function \(p. 243\)](#)
- [Configuring IAM policies \(p. 243\)](#)
- [Configuring code signing with the Lambda API \(p. 244\)](#)

Signature validation

Lambda performs the following validation checks when you deploy a signed code package to your function:

1. Integrity – Validates that the code package has not been modified since it was signed. Lambda compares the hash of the package with the hash from the signature.
2. Expiry – Validates that the signature of the code package has not expired.
3. Mismatch – Validates that the code package is signed with one of the allowed signing profiles for the Lambda function. A mismatch also occurs if a signature is not present.
4. Revocation – Validates that the signature of the code package has not been revoked.

The signature validation policy defined in the code signing configuration determines which of the following actions Lambda takes if any of the validation checks fail:

- Warn – Lambda allows the deployment of the code package, but issues a warning. Lambda issues a new Amazon CloudWatch metric and also stores the warning in the CloudTrail log.
- Enforce – Lambda issues a warning (the same as for the Warn action) and blocks the deployment of the code package.

You can configure the policy for the expiry, mismatch, and revocation validation checks. Note that you cannot configure a policy for the integrity check. If the integrity check fails, Lambda blocks deployment.

Configuration prerequisites

Before you can configure code signing for a Lambda function, use AWS Signer to do the following:

- Create one or more signing profiles.
- Use a signing profile to create a signed code package for your function.

For more information, see [Creating Signing Profiles \(Console\)](#) in the *AWS Signer Developer Guide*.

Creating code signing configurations

A code signing configuration defines a list of allowed signing profiles and the signature validation policy.

To create a code signing configuration (console)

1. Open the [Code signing configurations page](#) of the Lambda console.
2. Choose **Create configuration**.
3. For **Description**, enter a descriptive name for the configuration.
4. Under **Signing profiles**, add up to 20 signing profiles to the configuration.
 - a. For **Signing profile version ARN**, choose a profile version's Amazon Resource Name (ARN), or enter the ARN.
 - b. To add an additional signing profile, choose **Add signing profiles**.
5. Under **Signature validation policy**, choose **Warn** or **Enforce**.
6. Choose **Create configuration**.

Updating a code signing configuration

When you update a code signing configuration, the changes impact the future deployments of functions that have the code signing configuration attached.

To update a code signing configuration (console)

1. Open the [Code signing configurations page](#) of the Lambda console.
2. Select a code signing configuration to update, and then choose **Edit**.
3. For **Description**, enter a descriptive name for the configuration.
4. Under **Signing profiles**, add up to 20 signing profiles to the configuration.
 - a. For **Signing profile version ARN**, choose a profile version's Amazon Resource Name (ARN), or enter the ARN.
 - b. To add an additional signing profile, choose **Add signing profiles**.
5. Under **Signature validation policy**, choose **Warn** or **Enforce**.
6. Choose **Save changes**.

Deleting a code signing configuration

You can delete a code signing configuration only if no functions are using it.

To delete a code signing configuration (console)

1. Open the [Code signing configurations page](#) of the Lambda console.
2. Select a code signing configuration to delete, and then choose **Delete**.
3. To confirm, choose **Delete** again.

Enabling code signing for a function

To enable code signing for a function, you associate a code signing configuration with the function.

To associate a code signing configuration with a function (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function for which you want to enable code signing.
3. Under **Code signing configuration**, choose **Edit**.
4. In **Edit code signing**, choose a code signing configuration for this function.
5. Choose **Save**.

Configuring IAM policies

To grant permission for a user to access the [code signing API operations \(p. 244\)](#), attach one or more policy statements to the user policy. For more information about user policies, see [Identity-based IAM policies for Lambda \(p. 823\)](#).

The following example policy statement grants permission to create, update, and retrieve code signing configurations.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "lambda:CreateCodeSigningConfig",  
                "lambda:UpdateCodeSigningConfig",  
                "lambda:GetCodeSigningConfig"  
            ]  
        }  
    ]  
}
```

```
        "lambda:UpdateCodeSigningConfig",
        "lambda:GetCodeSigningConfig"
    ],
    "Resource": "*"
}
]
```

Administrators can use the `CodeSigningConfigArn` condition key to specify the code signing configurations that developers must use to create or update your functions.

The following example policy statement grants permission to create a function. The policy statement includes a `lambda:CodeSigningConfigArn` condition to specify the allowed code signing configuration. Lambda blocks any `CreateFunction` API request if its `CodeSigningConfigArn` parameter is missing or does not match the value in the condition.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowReferencingCodeSigningConfig",
            "Effect": "Allow",
            "Action": [
                "lambda>CreateFunction",
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "lambda:CodeSigningConfigArn":
                        "arn:aws:lambda:us-west-2:123456789012:code-signing-
config:csc-0d4518bd353a0a7c6"
                }
            }
        }
    ]
}
```

Configuring code signing with the Lambda API

To manage code signing configurations with the AWS CLI or AWS SDK, use the following API operations:

- [ListCodeSigningConfigs](#)
- [CreateCodeSigningConfig](#)
- [GetCodeSigningConfig](#)
- [UpdateCodeSigningConfig](#)
- [DeleteCodeSigningConfig](#)

To manage the code signing configuration for a function, use the following API operations:

- [CreateFunction \(p. 1165\)](#)
- [GetFunctionCodeSigningConfig](#)
- [PutFunctionCodeSigningConfig](#)
- [DeleteFunctionCodeSigningConfig](#)
- [ListFunctionsByCodeSigningConfig](#)

Using layers with your Lambda function

A Lambda layer is a .zip file archive that can contain additional code or other content. A layer can contain libraries, a custom runtime, data, or configuration files. Use layers to reduce deployment package size and to promote code sharing and separation of responsibilities so that you can iterate faster on writing business logic.

You can use layers only with Lambda functions [deployed as a .zip file archive \(p. 18\)](#). For a function [defined as a container image \(p. 881\)](#), you can package your preferred runtime and all code dependencies when you create the container image. For more information, see [Working with Lambda layers and extensions in container images](#) on the AWS Compute Blog.

Sections

- [Prerequisites \(p. 245\)](#)
- [Configuring functions to use layers \(p. 245\)](#)
- [Accessing layer content from your function \(p. 248\)](#)
- [Finding layer information \(p. 248\)](#)
- [Using AWS SAM to add a layer to a function \(p. 97\)](#)
- [Sample applications \(p. 249\)](#)

Prerequisites

Before you can configure a Lambda function to use a layer, you must:

- Create a layer. For more information, see [Creating and sharing Lambda layers \(p. 93\)](#).
- Make sure that you have permission to call the [GetLayerVersion \(p. 1243\)](#) API operation on the layer version. For functions in your AWS account, you can add this permission from your [user policy \(p. 823\)](#). To use a layer in another account, the owner of that account must grant your account permission in a [resource-based policy \(p. 832\)](#). For examples, see [Granting layer access to other accounts \(p. 836\)](#).

Configuring functions to use layers

You can add up to five layers to a Lambda function. The total unzipped size of the function and all layers cannot exceed the unzipped deployment package size quota of 250 MB. For more information, see [Lambda quotas \(p. 1131\)](#).

If your functions consume a layer that a different AWS account publishes, your functions can continue to use the layer version after it has been deleted, or after your permission to access the layer is revoked. However, you cannot create a new function that uses a deleted layer version.

Note

Make sure that the layers that you add to a function are compatible with the runtime and instruction set architecture of the function.

Configuring layers with the console

Adding a layer to a function

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to configure.
3. Under **Layers**, choose **Add a layer**

4. Under **Choose a layer**, choose a layer source.
5. For the **AWS layers or Custom layers** layer source:
 - a. Choose a layer from the pull-down menu.
 - b. Under **Version**, choose a layer version from the pull-down menu. Each layer version entry lists its compatible runtimes and architectures.
 - c. Choose **Add**.
6. For the **Specify an ARN** layer source:
 - a. Enter an ARN in the text box and choose **Verify**.
 - b. Choose **Add**.

The order in which you add the layers is the order in which Lambda later merges the layer content into the execution environment. You can change the layer merge order using the console.

Update layer order for your function

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to configure.
3. Under **Layers**, choose **Edit**
4. Choose one of the layers.
5. Choose **Merge earlier** or **Merge later** to adjust the order of the layers.
6. Choose **Save**.

Layers are versioned, and the content of each layer version is immutable. The layer owner can release a new layer version to provide updated content. You can use the console to update your functions' layer versions.

Update layer versions for your function

1. Open the [Functions page](#) of the Lambda console.
2. Under **Additional resources**, choose **Layers**.
3. Choose the layer to modify.
4. Under **Functions using this version**, select the functions you want to modify, then choose **Edit**.
5. From **Layer version**, select the layer version to change to.
6. Choose **Update functions**.

You cannot update functions' layer versions across AWS accounts.

Configuring layers with the API

To add layers to your function, use the **update-function-configuration** command. The following example adds two layers: one from the same AWS account as the function, and one from a different account.

```
aws lambda update-function-configuration --function-name my-function \
--layers arn:aws:lambda:us-east-2:123456789012:layer:my-layer:3
 \
arn:aws:lambda:us-east-2:11122223333:layer:their-layer:2
```

You should see output similar to the following:

```
{
  "FunctionName": "test-layers",
  "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
  "Runtime": "nodejs18.x",
  "Role": "arn:aws:iam::123456789012:role/service-role/lambda-role",
  "Layers": [
    {
      "Arn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer:3",
      "CodeSize": 169
    },
    {
      "Arn": "arn:aws:lambda:us-east-2:111122223333:layer:their-layer:2",
      "CodeSize": 169
    }
  ],
  "RevisionId": "81cc64f5-5772-449a-b63e-12330476bcc4",
  ...
}
```

To specify the layer versions to use, you must provide the full Amazon Resource Name (ARN) of each layer version. When you add layers to a function that already has layers, you overwrite the previous list of layers. Be sure to include all layers every time that you update the layer configuration. The order in which you add the layers is the order in which Lambda later extracts the layer content into the execution environment.

To remove all layers, specify an empty list.

```
aws lambda update-function-configuration --function-name my-function --layers []
```

The creator of a layer can delete a version of the layer. If you're using that layer version in a function, your function continues to run as though the layer version still exists. However, when you update the layer configuration, you must remove the reference to the deleted version.

Layers are versioned, and the content of each layer version is immutable. The layer owner can release a new layer version to provide updated content. You can use the API to update the layer versions that your function uses.

Update layer versions for your function

To update one or more layer versions for your function, use the **update-function-configuration** command. Use the **--layers** option with this command to include all of the layer versions for the function, even if you are updating one of the layer versions. If the function already has layers, the new list overwrites the previous list.

The following procedure steps assume that you have packaged the updated layer code into a local file named `layer.zip`.

1. (Optional) If the new layer version is not published yet, publish the new version.

```
aws lambda publish-layer-version --layer-name my-layer --description "My Layer" --
license-info "MIT" \
--zip-file "fileb://layer.zip" --compatible-runtimes python3.6 python3.7
```

2. (Optional) If the function has more than one layer, get the current layer versions associated with the function.

```
aws lambda get-function-configuration --function-name my-function --query 'Layers[*].Arn' --output yaml
```

3. Add the new layer version to the function. In the following example command, the function also has a layer version named other-layer:5:

```
aws lambda update-function-configuration --function-name my-function \
--layers arn:aws:lambda:us-east-2:123456789012:layer:my-layer:2 \
arn:aws:lambda:us-east-2:123456789012:layer:other-layer:5
```

Accessing layer content from your function

If your Lambda function includes layers, Lambda extracts the layer contents into the /opt directory in the function execution environment. Lambda extracts the layers in the order (low to high) listed by the function. Lambda merges folders with the same name, so if the same file appears in multiple layers, the function uses the version in the last extracted layer.

Each [Lambda runtime \(p. 37\)](#) adds specific /opt directory folders to the PATH variable. Your function code can access the layer content without the need to specify the path. For more information about path settings in the Lambda execution environment, see [Defined runtime environment variables \(p. 79\)](#).

Finding layer information

To find layers in your AWS account that are compatible with your Lambda function's runtime, use the **list-layers** command.

```
aws lambda list-layers --compatible-runtime python3.8
```

You should see output similar to the following:

```
{  
    "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": "2018-11-15T00:37:46.592+0000",  
                "CompatibleRuntimes": [  
                    "python3.6",  
                    "python3.7",  
                    "python3.8",  
                ]  
            }  
        }  
    ]  
}
```

To list all layers in your account, you can omit the --compatible-runtime option. The details in the response reflect the latest version of the layer.

You can also get the latest version of a layer using the **list-layer-versions** command.

```
aws lambda list-layer-versions --layer-name my-layer --query  
'LayerVersions[0].LayerVersionArn'
```

Using AWS SAM to add a layer to a function

To automate the creation and mapping of layers in your application, use the AWS Serverless Application Model (AWS SAM). The `AWS::Serverless::LayerVersion` resource type creates a layer version that you can reference from your Lambda function configuration.

Example [blank-nodejs/template.yml](#) – Serverless resources

```
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:  
      Handler: index.handler  
      Runtime: nodejs12.x  
      CodeUri: function/.  
      Description: Call the AWS Lambda API  
      Timeout: 10  
      # Function's execution role  
      Policies:  
        - AWSLambdaBasicExecutionRole  
        - AWSLambda_ReadOnlyAccess  
        - AWSSXrayWriteOnlyAccess  
      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:  
        - nodejs12.x
```

When you update your dependencies and deploy, AWS SAM creates a new version of the layer and updates the mapping.

Sample applications

The GitHub repository for this guide provides blank sample applications that demonstrate the use of layers for dependency management.

- **Node.js** – [blank-nodejs](#)
- **Python** – [blank-python](#)
- **Ruby** – [blank-ruby](#)
- **Java** – [blank-java](#)

For more information about the blank sample app, see [Blank function sample application for AWS Lambda \(p. 1010\)](#). For other samples, see [Sample applications \(p. 1008\)](#).

Using tags on Lambda functions

You can tag AWS Lambda functions to activate [attribute-based access control \(ABAC\) \(p. 828\)](#) and to organize them by owner, project, or department. Tags are free-form key-value pairs that are supported across AWS services for use in ABAC, filtering resources, and [adding detail to billing reports](#).

Tags apply at the function level, not to versions or aliases. Tags are not part of the version-specific configuration that Lambda creates a snapshot of when you publish a version.

Sections

- [Permissions required for working with tags \(p. 250\)](#)
- [Using tags with the Lambda console \(p. 250\)](#)
- [Using tags with the AWS CLI \(p. 252\)](#)
- [Requirements for tags \(p. 253\)](#)

Permissions required for working with tags

Grant appropriate permissions to the AWS Identity and Access Management (IAM) identity (user, group, or role) for the person working with the function:

- **lambda>ListTags** – When a function has tags, grant this permission to anyone who needs to call GetFunction or ListTags on it.
- **lambda:TagResource** – Grant this permission to anyone who needs to call CreateFunction or TagResource.

For more information, see [Identity-based IAM policies for Lambda \(p. 823\)](#).

Using tags with the Lambda console

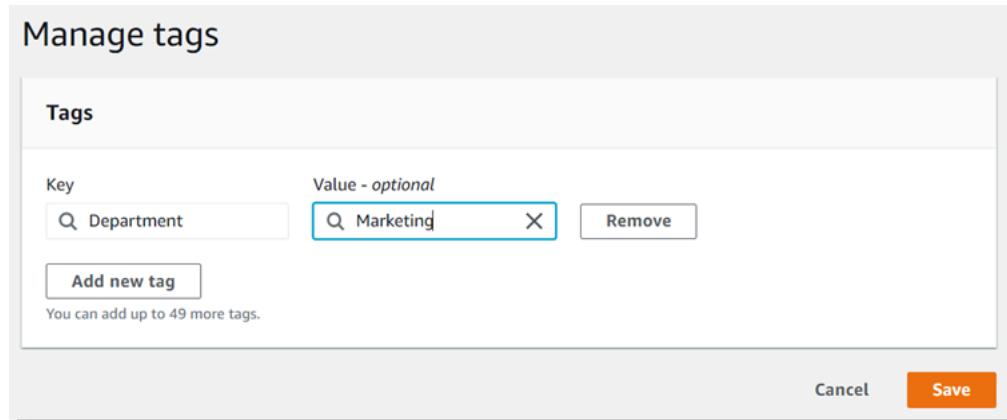
You can use the Lambda console to create functions that have tags, add tags to existing functions, and filter functions by tags that you add.

To add tags when you create a function

1. Open the [Functions page](#) of the Lambda console.
2. Choose **Create function**.
3. Choose **Author from scratch** or **Container image**.
4. Under **Basic information**, do the following:
 - a. For **Function name**, enter the function name. Function names are limited to 64 characters in length.
 - b. For **Runtime**, choose the language version to use for your function.
 - c. (Optional) For **Architecture**, choose the [instruction set architecture \(p. 29\)](#) to use for your function. The default architecture is x86_64. When you build the deployment package for your function, make sure that it is compatible with the instruction set architecture that you choose.
5. Expand **Advanced settings**, and then select **Enable tags**.
6. Choose **Add new tag**, and then enter a **Key** and an optional **Value**. To add more tags, repeat this step.
7. Choose **Create function**.

To add tags to an existing function

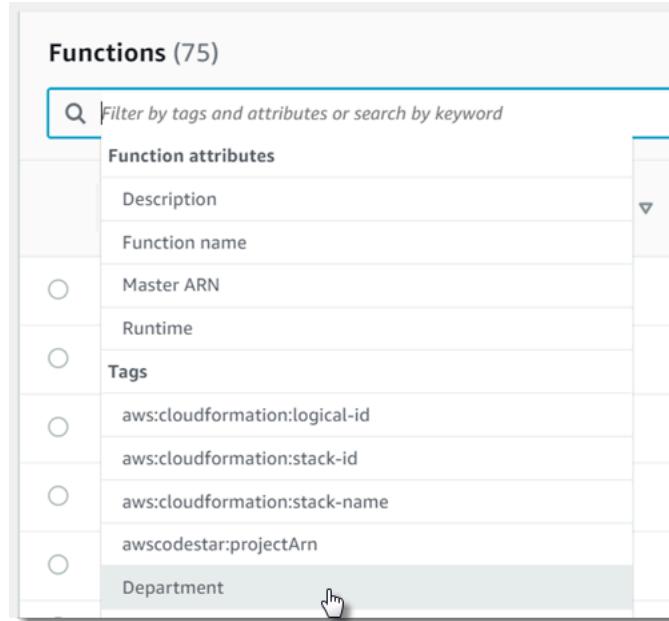
1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of a function.
3. Choose **Configuration**, and then choose **Tags**.
4. Under **Tags**, choose **Manage tags**.
5. Choose **Add new tag**, and then enter a **Key** and an optional **Value**. To add more tags, repeat this step.



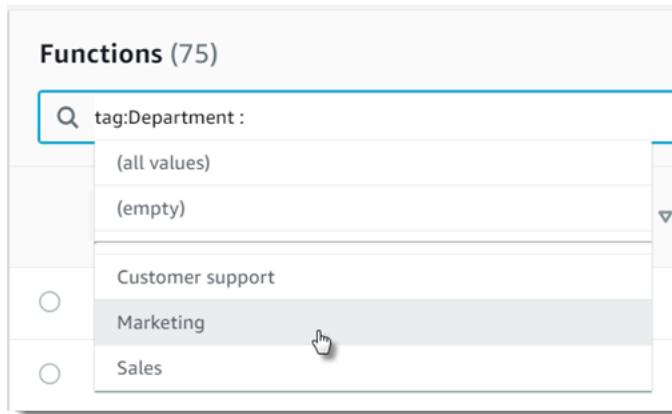
6. Choose **Save**.

To filter functions with tags

1. Open the [Functions page](#) of the Lambda console.
2. Choose the search bar to see a list of function attributes and tag keys.



3. Choose a tag key to see a list of values that are in use in the current AWS Region.
4. Choose a value to see functions with that value, or choose **(all values)** to see all functions that have a tag with that key.



The search bar also supports searching for tag keys. Enter tag to see only a list of tag keys, or enter the name of a key to find it in the list.

Using tags with the AWS CLI

Adding and removing tags

To create a new Lambda function with tags, use the **create-function** command with the --tags option.

```
aws lambda create-function --function-name my-function
--handler index.js --runtime nodejs18.x \
--role arn:aws:iam::123456789012:role/Lambda-role \
--tags Department=Marketing,CostCenter=1234ABCD
```

To add tags to an existing function, use the **tag-resource** command.

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:function:my-function \
--tags Department=Marketing,CostCenter=1234ABCD
```

To remove tags, use the **untag-resource** command.

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:function:my-function \
--tag-keys Department
```

Viewing tags on a function

If you want to view the tags that are applied to a specific Lambda function, you can use either of the following AWS CLI commands:

- [ListTags \(p. 1304\)](#) – To view a list of the tags associated with this function, include your Lambda function ARN (Amazon Resource Name):

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:function:my-function
```

- [GetFunction \(p. 1220\)](#) – To view a list of the tags associated with this function, include your Lambda function name:

```
aws lambda get-function --function-name my-function
```

Filtering functions by tag

You can use the AWS Resource Groups Tagging API [GetResources](#) API operation to filter your resources by tags. The GetResources operation receives up to 10 filters, with each filter containing a tag key and up to 10 tag values. You provide GetResources with a ResourceType to filter by specific resource types.

For more information about AWS Resource Groups, see [What are resource groups?](#) in the *AWS Resource Groups and Tags User Guide*.

Requirements for tags

The following requirements apply to tags:

- Maximum number of tags per resource: 50
- Maximum key length: 128 Unicode characters in UTF-8
- Maximum value length: 256 Unicode characters in UTF-8
- Tag keys and values are case sensitive.
- Do not use the aws : prefix in your tag names or values because it is reserved for AWS use. You can't edit or delete tag names or values with this prefix. Tags with this prefix do not count against your tags per resource limit.
- If you plan to use your tagging schema across multiple services and resources, remember that other services may have restrictions on allowed characters. Generally allowed characters are: letters, spaces, and numbers representable in UTF-8, plus the following special characters: + - = . _ : / @.

Building Lambda functions with Node.js

You can run JavaScript code with Node.js in AWS Lambda. Lambda provides [runtimes \(p. 37\)](#) for Node.js that run your code to process events. Your code runs in an environment that includes the AWS SDK for JavaScript, with credentials from an AWS Identity and Access Management (IAM) role that you manage.

Lambda supports the following Node.js runtimes.

Node.js

Name	Identifier	SDK	Operating system	Architectures	Deprecation (Phase 1)
Node.js 18	nodejs18.x	3.188.0	Amazon Linux 2	x86_64, arm64	
Node.js 16	nodejs16.x	2.1083.0	Amazon Linux 2	x86_64, arm64	
Node.js 14	nodejs14.x	2.1055.0	Amazon Linux 2	x86_64, arm64	
Node.js 12	nodejs12.x	2.1055.0	Amazon Linux 2	x86_64, arm64	Mar 31, 2023

Note

The Node 18 runtime uses AWS SDK for JavaScript v3. To migrate a function to Node 18 from an earlier runtime, follow the [migration workshop](#) on GitHub. For more information about AWS SDK for JavaScript version 3, see the [Modular AWS SDK for JavaScript is now generally available](#) blog post.

Lambda functions use an [execution role \(p. 816\)](#) to get permission to write logs to Amazon CloudWatch Logs, and to access other services and resources. If you don't already have an execution role for function development, create one.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – Lambda.**
 - **Permissions – AWSLambdaBasicExecutionRole.**
 - **Role name – lambda-role.**

The **AWSLambdaBasicExecutionRole** policy has the permissions that the function needs to write logs to CloudWatch Logs.

You can add permissions to the role later, or swap it out for a different role that's specific to a single function.

To create a Node.js function

1. Open the [Lambda console](#).
2. Choose **Create function**.
3. Configure the following settings:
 - **Name** – **my-function**.
 - **Runtime** – **Node.js 18.x**.
 - **Role** – **Choose an existing role**.
 - **Existing role** – **lambda-role**.
4. Choose **Create function**.
5. To configure a test event, choose **Test**.
6. For **Event name**, enter **test**.
7. Choose **Save changes**.
8. To invoke the function, choose **Test**.

The console creates a Lambda function with a single source file named `index.js` or `index.mjs`. You can edit this file and add more files in the built-in [code editor \(p. 21\)](#). To save your changes, choose **Save**. Then, to run your code, choose **Test**.

Note

The Lambda console uses AWS Cloud9 to provide an integrated development environment in the browser. You can also use AWS Cloud9 to develop Lambda functions in your own environment. For more information, see [Working with Lambda Functions](#) in the AWS Cloud9 user guide.

The `index.js` or `index.mjs` file exports a function named `handler` that takes an event object and a context object. This is the [handler function \(p. 258\)](#) that Lambda calls when the function is invoked. The Node.js function runtime gets invocation events from Lambda and passes them to the handler. In the function configuration, the handler value is `index.handler`.

When you save your function code, the Lambda console creates a .zip file archive deployment package. When you develop your function code outside of the console (using an IDE) you need to [create a deployment package \(p. 263\)](#) to upload your code to the Lambda function.

Note

To get started with application development in your local environment, deploy one of the sample applications available in this guide's GitHub repository.

Sample Lambda applications in Node.js

- [blank-nodejs](#) – A Node.js function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.
- [nodejs-apig](#) – A function with a public API endpoint that processes an event from API Gateway and returns an HTTP response.
- [rds-mysql](#) – A function that relays queries to a MySQL for RDS Database. This sample includes a private VPC and database instance configured with a password in AWS Secrets Manager.
- [efs-nodejs](#) – A function that uses an Amazon EFS file system in a Amazon VPC. This sample includes a VPC, file system, mount targets, and access point configured for use with Lambda.
- [list-manager](#) – A function processes events from an Amazon Kinesis data stream and update aggregate lists in Amazon DynamoDB. The function stores a record of each event in a MySQL for RDS Database in a private VPC. This sample includes a private VPC with a VPC endpoint for DynamoDB and a database instance.
- [error-processor](#) – A Node.js function generates errors for a specified percentage of requests. A CloudWatch Logs subscription invokes a second function when an error is recorded. The

processor function uses the AWS SDK to gather details about the request and stores them in an Amazon S3 bucket.

The function runtime passes a context object to the handler, in addition to the invocation event. The [context object \(p. 271\)](#) contains additional information about the invocation, the function, and the execution environment. More information is available from environment variables.

Your Lambda function comes with a CloudWatch Logs log group. The function runtime sends details about each invocation to CloudWatch Logs. It relays any [logs that your function outputs \(p. 273\)](#) during invocation. If your function [returns an error \(p. 278\)](#), Lambda formats the error and returns it to the invoker.

Node.js initialization

Node.js has a unique event loop model that causes its initialization behavior to be different from other runtimes. Specifically, Node.js uses a non-blocking I/O model that supports asynchronous operations. This model allows Node.js to perform efficiently for most workloads. For example, if a Node.js function makes a network call, that request may be designated as an asynchronous operation and placed into a callback queue. The function may continue to process other operations within the main call stack without getting blocked by waiting for the network call to return. Once the network call is completed, its callback is executed and then removed from the callback queue.

Some initialization tasks may run asynchronously. These asynchronous tasks are not guaranteed to complete execution prior to an invocation. For example, code that makes a network call to fetch a parameter from AWS Parameter Store may not be complete by the time Lambda executes the handler function. As a result, the variable may be null during an invocation. To avoid this, ensure that variables and other asynchronous code are fully initialized before continuing with the rest of the function's core business logic.

Alternatively, you can designate your function code as an ES module, allowing you to use `await` at the top level of the file, outside the scope of your function handler. When you await every Promise, the asynchronous initialization code completes before handler invocations, maximizing the effectiveness of [provisioned concurrency \(p. 213\)](#) in reducing cold start latency. For more information and an example, see [Using Node.js ES modules and top-level await in AWS Lambda](#).

Designating a function handler as an ES module

By default, Lambda treats files with the `.js` suffix as CommonJS modules. Optionally, you can designate your code as an ES module. You can do this in two ways: specifying the type as `module` in the function's package `.json` file, or by using the `.mjs` file name extension. In the first approach, your function code treats all `.js` files as ES modules, while in the second scenario, only the file you specify with `.mjs` is an ES module. You can mix ES modules and CommonJS modules by naming them `.mjs` and `.cjs` respectively, as `.mjs` files are always ES modules and `.cjs` files are always CommonJS modules.

In Node.js 14 and Node.js 16, the Lambda runtime loads ES modules from the same folder as your function handler, or a subfolder. Starting with Node.js 18, Lambda searches folders in the `NODE_PATH` environment variable when loading ES modules. With Node.js 18, you can load the AWS SDK that's included in the runtime using ES module `import` statements. You can also load ES modules from [layers \(p. 11\)](#).

Topics

- [AWS Lambda function handler in Node.js \(p. 258\)](#)
- [Deploy Node.js Lambda functions with .zip file archives \(p. 263\)](#)
- [Deploy Node.js Lambda functions with container images \(p. 266\)](#)

- [AWS Lambda context object in Node.js \(p. 271\)](#)
- [AWS Lambda function logging in Node.js \(p. 273\)](#)
- [AWS Lambda function errors in Node.js \(p. 278\)](#)
- [Instrumenting Node.js code in AWS Lambda \(p. 282\)](#)

AWS Lambda function handler in Node.js

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. When the handler exits or returns a response, it becomes available to handle another event.

The following example function logs the contents of the event object and returns the location of the logs.

Note

This page shows examples of both CommonJS and ES module handlers. To learn about the difference between these two handler types, see [Designating a function handler as an ES module \(p. 256\)](#).

CommonJS module handler

Example

```
exports.handler = async function (event, context) {
  console.log("EVENT: \n" + JSON.stringify(event, null, 2));
  return context.logStreamName;
};
```

ES module handler

Example

```
export const handler = async (event, context) => {
  console.log("EVENT: \n" + JSON.stringify(event, null, 2));
  return context.logStreamName;
};
```

When you configure a function, the value of the handler setting is the file name and the name of the exported handler method, separated by a dot. The default in the console and for examples in this guide is `index.handler`. This indicates the `handler` method that's exported from the `index.js` file.

The runtime passes arguments to the handler method. The first argument is the event object, which contains information from the invoker. The invoker passes this information as a JSON-formatted string when it calls [Invoke \(p. 1260\)](#), and the runtime converts it to an object. When an AWS service invokes your function, the event structure [varies by service \(p. 556\)](#).

The second argument is the [context object \(p. 271\)](#), which contains information about the invocation, function, and execution environment. In the preceding example, the function gets the name of the [log stream \(p. 273\)](#) from the context object and returns it to the invoker.

You can also use a callback argument, which is a function that you can call in non-async handlers to send a response. We recommend that you use `async/await` instead of callbacks. `Async/await` provides improved readability, error handling, and efficiency. For more information about the differences between `async/await` and callbacks, see [Using callbacks \(p. 260\)](#).

Using `async/await`

If your code performs an asynchronous task, use the `async/await` pattern to make sure that the handler finishes running. `Async/await` is a concise and readable way to write asynchronous code in Node.js,

without the need for nested callbacks or chaining promises. With `async/await`, you can write code that reads like synchronous code, while still being asynchronous and non-blocking.

The `async` keyword marks a function as asynchronous, and the `await` keyword pauses the execution of the function until a Promise is resolved.

Note

Make sure to wait for asynchronous events to complete. If the function returns before `async` events are complete, the function might fail or cause unexpected behavior in your application. This can happen when a `forEach` loop contains an `async` event. `forEach` loops expect a synchronous call. For more information, see [Array.prototype.forEach\(\)](#) in the Mozilla documentation.

CommonJS module handler

Example – HTTP request with `async/await`

```
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;
};
```

ES module handler

Example – HTTP request with `async/await`

This example uses `fetch`, which is available in the nodejs18.x runtime.

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

export const handler = async(event) => {
    try {
        // fetch is available with Node.js 18
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

The next example uses `async/await` to list your Amazon Simple Storage Service buckets.

Note

Before using this example, make sure that your function's execution role has Amazon S3 read permissions.

CommonJS module handler

Example – AWS SDK v2 with `async/await`

This example uses the [AWS SDK for JavaScript v2](#), which is available in the Node.js 12, 14, and 16 runtimes.

```
const AWS = require('aws-sdk')
const s3 = new AWS.S3()

exports.handler = async function(event) {
  const buckets = await s3.listBuckets().promise()
  return buckets
}
```

ES module handler

Example – AWS SDK v3 with `async/await`

This example uses the [AWS SDK for JavaScript v3](#), which is available in the nodejs18.x runtime.

```
import {S3Client, ListBucketsCommand} from '@aws-sdk/client-s3';
const s3 = new S3Client({region: 'us-east-1'});

export const handler = async(event) => {
  const data = await s3.send(new ListBucketsCommand({}));
  return data.Buckets;
};
```

Using callbacks

We recommend that you use [async/await \(p. 258\)](#) to declare the function handler instead of using callbacks. Async/await is a better choice for several reasons:

- **Readability:** Async/await code is easier to read and understand than callback code, which can quickly become difficult to follow and result in callback hell.
- **Debugging and error handling:** Debugging callback-based code can be difficult. The call stack can become hard to follow and errors can easily be swallowed. With async/await, you can use try/catch blocks to handle errors.
- **Efficiency:** Callbacks often require switching between different parts of the code. Async/await can reduce the number of context switches, resulting in more efficient code.

When you use callbacks in your handler, the function continues to execute until the [event loop](#) is empty or the function times out. The response isn't sent to the invoker until all event loop tasks are finished. If the function times out, an error is returned instead. You can configure the runtime to send the response immediately by setting [context.callbackWaitsForEmptyEventLoop \(p. 271\)](#) to false.

The callback function takes two arguments: an `Error` and a response. The response object must be compatible with `JSON.stringify`.

The following example function checks a URL and returns the status code to the invoker.

CommonJS module handler

Example – HTTP request with callback

```
const https = require("https");
```

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

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

ES module handler

Example – HTTP request with callback

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

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

In the next example, the response from Amazon S3 is returned to the invoker as soon as it's available. The timeout running on the event loop is frozen, and it continues running the next time the function is invoked.

Note

Before using this example, make sure that your function's execution role has Amazon S3 read permissions.

CommonJS module handler

Example – AWS SDK v2 with callbackWaitsForEmptyEventLoop

This example uses the [AWS SDK for JavaScript v2](#), which is available in the Node.js 12, 14, and 16 runtimes.

```
const AWS = require("aws-sdk");
const s3 = new AWS.S3();

exports.handler = function (event, context, callback) {
    context.callbackWaitsForEmptyEventLoop = false;
    s3.listBuckets(null, callback);
    setTimeout(function () {
        console.log("Timeout complete.");
    }, 5000);
};
```

ES module handler

Example – AWS SDK v3 with callbackWaitsForEmptyEventLoop

This example uses the [AWS SDK for JavaScript v3](#), which is available in the nodejs18.x runtime.

```
import AWS from "@aws-sdk/client-s3";
const s3 = new AWS.S3({});
```

```
export const handler = function (event, context, callback) {
  context.callbackWaitsForEmptyEventLoop = false;
  s3.listBuckets({}, callback);
  setTimeout(function () {
    console.log("Timeout complete.");
  }, 5000);
};
```

Deploy Node.js Lambda functions with .zip file archives

Your AWS Lambda function's code consists of scripts or compiled programs and their dependencies. You use a *deployment package* to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives.

To create the deployment package for a .zip file archive, you can use a built-in .zip file archive utility or any other .zip file utility (such as [7zip](#)) for your command line tool. Note the following requirements for using a .zip file as your deployment package:

- The .zip file contains your function's code and any dependencies used to run your function's code (if applicable) on Lambda. If your function depends only on standard libraries, or AWS SDK libraries, you don't need to include these libraries in your .zip file. These libraries are included with the supported [Lambda runtime \(p. 37\)](#) environments.
- If the .zip file is larger than 50 MB, we recommend uploading it to your function from an Amazon Simple Storage Service (Amazon S3) bucket.
- If your deployment package contains native libraries, you can build the deployment package with AWS Serverless Application Model (AWS SAM). You can use the AWS SAM CLI `sam build` command with the `--use-container` to create your deployment package. This option builds a deployment package inside a Docker image that is compatible with the Lambda execution environment.

For more information, see [sam build](#) in the *AWS Serverless Application Model Developer Guide*.

- You need to build the deployment package to be compatible with this [instruction set architecture \(p. 29\)](#) of the function.
- Lambda uses POSIX file permissions, so you may need to [set permissions for the deployment package folder](#) before you create the .zip file archive.

Sections

- [Prerequisites \(p. 263\)](#)
- [Updating a function with no dependencies \(p. 263\)](#)
- [Updating a function with additional dependencies \(p. 264\)](#)

Prerequisites

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

Updating a function with no dependencies

To update a function by using the Lambda API, use the [UpdateFunctionCode \(p. 1367\)](#) operation. Create an archive that contains your function code, and upload it using the AWS Command Line Interface (AWS CLI).

To update a Node.js function with no dependencies

1. Create a .zip file archive.

```
zip function.zip index.js
```

2. To upload the package, use the update-function-code command.

```
aws lambda update-function-code --function-name my-function --zip-file fileb://function.zip
```

You should see the following output:

```
{  
  "FunctionName": "my-function",  
  "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",  
  "Runtime": "nodejs12.x",  
  "Role": "arn:aws:iam::123456789012:role/lambda-role",  
  "Handler": "index.handler",  
  "CodeSha256": "0f0hMc1I2di6YFMi9aXm3JtGTmcDbjniEuiYonYptAk=",  
  "Version": "$LATEST",  
  "TracingConfig": {  
    "Mode": "Active"  
  },  
  "RevisionId": "983ed1e3-ca8e-434b-8dc1-7d72ebadd83d",  
  ...  
}
```

Updating a function with additional dependencies

If your function depends on libraries other than the AWS SDK for JavaScript, use [npm](#) to include them in your deployment package. Ensure that the Node.js version in your local environment matches the Node.js version of your function. If any of the libraries use native code, [use an Amazon Linux environment](#) to create the deployment package.

You can add the SDK for JavaScript to the deployment package if you need a newer version than the one [included on the runtime \(p. 254\)](#), or to ensure that the version doesn't change in the future.

If your deployment package contains native libraries, you can build the deployment package with AWS Serverless Application Model (AWS SAM). You can use the AWS SAM CLI `sam build` command with the `--use-container` to create your deployment package. This option builds a deployment package inside a Docker image that is compatible with the Lambda execution environment.

For more information, see [sam build](#) in the *AWS Serverless Application Model Developer Guide*.

As an alternative, you can create the deployment package using an Amazon EC2 instance that provides an Amazon Linux environment. For instructions, see [Using Packages and Native nodejs Modules in AWS](#) in the AWS compute blog.

To update a Node.js function with dependencies

1. Open a command line terminal or shell. Ensure that the Node.js version in your local environment matches the Node.js version of your function.
2. Create a folder for the deployment package. The following steps assume that the folder is named `my-function`.
3. Install libraries in the `node_modules` directory using the `npm install` command.

```
npm install aws-xray-sdk
```

This creates a folder structure that's similar to the following:

```
~/my-function
### index.js
### node_modules
###  async
###  async-listener
###  atomic-batcher
###  aws-sdk
###  aws-xray-sdk
###  aws-xray-sdk-core
```

4. Create a .zip file that contains the contents of your project folder. Use the `r` (recursive) option to ensure that zip compresses the subfolders.

```
zip -r function.zip .
```

5. Upload the package using the update-function-code command.

```
aws lambda update-function-code --function-name my-function --zip-file fileb://
function.zip
```

You should see the following output:

```
{
  "FunctionName": "my-function",
  "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
  "Runtime": "nodejs12.x",
  "Role": "arn:aws:iam::123456789012:role/lambda-role",
  "Handler": "index.handler",
  "CodeSha256": "Qf0hMc1I2di6YFMi9aXm3JtGTmcDbjniEuiYonYptAk=",
  "Version": "$LATEST",
  "TracingConfig": {
    "Mode": "Active"
  },
  "RevisionId": "983ed1e3-ca8e-434b-8dc1-7d72ebadd83d",
  ...
}
```

In addition to code and libraries, your deployment package can also contain executable files and other resources. For more information, see [Running Arbitrary Executables in AWS Lambda](#) in the AWS Compute Blog.

Deploy Node.js Lambda functions with container images

You can deploy your Lambda function code as a [container image \(p. 881\)](#). To help you build a container image for your Node.js function, AWS provides the following resources:

- AWS base images for Lambda

These base images are preloaded with a language runtime and other components that are required to run the image on Lambda. AWS provides a Dockerfile for each of the base images to help with building your container image.

- Open-source runtime interface clients

If you use a community or private enterprise base image, you must add a [runtime interface client \(p. 883\)](#) to the base image to make it compatible with Lambda.

- Open-source runtime interface emulator

Lambda provides a runtime interface emulator (RIE) for you to test your function locally. The base images for Lambda and base images for custom .runtimes include the RIE. For other base images, you can download the RIE for [testing your image \(p. 884\)](#) locally.

The workflow for a function defined as a container image includes these steps:

1. Build your container image using the resources listed in this topic.
2. Upload the image to your [Amazon Elastic Container Registry \(Amazon ECR\) container registry \(p. 891\)](#).
3. [Create the function \(p. 113\)](#) or [update the function code \(p. 115\)](#) to deploy the image to an existing function.

Topics

- [AWS base images for Node.js \(p. 266\)](#)
- [Using a Node.js base image \(p. 267\)](#)
- [Node.js runtime interface clients \(p. 270\)](#)

AWS base images for Node.js

AWS provides the following base images for Node.js:

Tags	Runtime	Operating system	Dockerfile	Deprecation
18	Node.js 18	Amazon Linux 2	Dockerfile for Node.js 18 on GitHub	
16	Node.js 16	Amazon Linux 2	Dockerfile for Node.js 16 on GitHub	
14	Node.js 14	Amazon Linux 2	Dockerfile for Node.js 14 on GitHub	

Tags	Runtime	Operating system	Dockerfile	Deprecation
12	Node.js 12	Amazon Linux 2	Dockerfile for Node.js 12 on GitHub	Mar 31, 2023

Amazon ECR repository: [gallery.ecr.aws/lambda/nodejs](#)

Using a Node.js base image

Prerequisites

To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [Docker](#)
- Node.js

Creating an image from a base image

To create a container image from an AWS base image for Node.js

1. Create a directory for the project, and then switch to that directory.

```
mkdir example
cd example
```

2. Create a new Node.js project with npm. To accept the default options provided in the interactive experience, press Enter.

```
npm init
```

3. Create a new file called `index.js`. You can add the following sample function code to the file for testing, or use your own.

Example CommonJS handler

```
exports.handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

4. If your function depends on libraries other than the AWS SDK for JavaScript, use [npm](#) to add them to your package.
5. Create a new Dockerfile with the following configuration:
 - Set the FROM property to the [URI of the base image](#).
 - Use the COPY command to copy the function code and runtime dependencies to `{LAMBDA_TASK_ROOT}`.
 - Set the CMD argument to the Lambda function handler.

Example Dockerfile

```
FROM public.ecr.aws/Lambda/nodejs:16

# 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. Build the Docker image with the [docker build](#) command. The following example names the image `docker-image` and gives it the `test` tag.

```
docker build -t docker-image:test .
```

(Optional) Test the image locally

1. Start the Docker image with the [docker run](#) command. In this example, `docker-image` is the image name and `test` is the tag.

```
docker run -p 9000:8080 docker-image:test
```

This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

2. Test your application locally using the [RIE \(p. 884\)](#). From a new terminal window, post an event to the following endpoint using a [curl](#) command:

```
curl -XPOST "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function running in the container image and returns a response.

3. Get the container ID.

```
docker ps
```

4. Use the [docker kill](#) command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the [get-login-password](#) command to authenticate the Docker CLI to your Amazon ECR registry.
 - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
 - Replace `111122223333` with your AWS account 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. Create a repository in Amazon ECR using the [create-repository](#) command.

```
aws ecr create-repository --repository-name hello-world --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

If successful, you see a response like this:

```
{  
    "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. Copy the `repositoryUri` from the output in the previous step.
4. Run the [docker tag](#) command to tag your local image into your Amazon ECR repository as the latest version. In this command:
 - Replace `docker-image:test` with the name and [tag](#) of your Docker image.
 - Replace the Amazon ECR repository URI with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the [docker push](#) command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.
6. [Create an execution role \(p. 191\)](#) for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.
7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
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
```

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{  
    "ExecutedVersion": "$LATEST",  
    "StatusCode": 200  
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must create a new image version and store the image in the Amazon ECR repository. For more information, see [Updating function code \(p. 115\)](#).

Node.js runtime interface clients

Install the runtime interface client for Node.js using the npm package manager:

```
npm install aws-lambda-ric
```

For package details, see [Lambda RIC](#) on the npm website.

You can also download the [Node.js runtime interface client](#) from GitHub. The NodeJS Runtime Interface Client package currently supports the following NodeJS versions:

- 10.x
- 12.x
- 14.x

AWS Lambda context object in Node.js

When Lambda runs your function, it passes a context object to the [handler \(p. 258\)](#). This object provides methods and properties that provide information about the invocation, function, and execution environment.

Context methods

- `getRemainingTimeInMillis()` – Returns the number of milliseconds left before the execution times out.

Context properties

- `functionName` – The name of the Lambda function.
- `functionVersion` – The [version \(p. 83\)](#) of the function.
- `invokedFunctionArn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memoryLimitInMB` – The amount of memory that's allocated for the function.
- `awsRequestId` – The identifier of the invocation request.
- `logGroupName` – The log group for the function.
- `logStreamName` – The log stream for the function instance.
- `identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
 - `cognitoIdentityId` – The authenticated Amazon Cognito identity.
 - `cognitoIdentityPoolId` – The Amazon Cognito identity pool that authorized the invocation.
- `clientContext` – (mobile apps) Client context that's provided to Lambda by the client application.
 - `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` – Custom values that are set by the client application.
- `callbackWaitsForEmptyEventLoop` – Set to `false` to send the response right away when the [callback \(p. 260\)](#) runs, instead of waiting for the Node.js event loop to be empty. If this is `false`, any outstanding events continue to run during the next invocation.

The following example function logs context information and returns the location of the logs.

Example index.js file

```
exports.handler = async function(event, context) {
  console.log('Remaining time: ', context.getRemainingTimeInMillis())
  console.log('Function name: ', context.functionName)
  return context.logStreamName
```

}

AWS Lambda function logging in Node.js

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

This page describes how to produce log output from your Lambda function's code, or access logs using the AWS Command Line Interface, the Lambda console, or the CloudWatch console.

Sections

- [Creating a function that returns logs \(p. 273\)](#)
- [Using the Lambda console \(p. 274\)](#)
- [Using the CloudWatch console \(p. 274\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 274\)](#)
- [Deleting logs \(p. 277\)](#)

Creating a function that returns logs

To output logs from your function code, you can use methods on the [console object](#), or any logging library that writes to `stdout` or `stderr`. The following example logs the values of environment variables and the event object.

Example index.js file – Logging

```
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 log format

```
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: 200 ms Memory Size: 128 MB Max Memory Used: 74 MB Init Duration: 166.62 ms XRAY TraceId: 1-5d9d007f-0a8c7fd02xmp1480aed55ef0 SegmentId: 3d752xmpl1bbe37e Sampled: true
```

The Node.js runtime logs the START, END, and REPORT lines for each invocation. It adds a timestamp, request ID, and log level to each entry logged by the function. The report line provides the following details.

Report Log

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TraceId** – For traced requests, the [AWS X-Ray trace ID \(p. 807\)](#).
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

You can view logs in the Lambda console, in the CloudWatch Logs console, or from the command line.

Using the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

Using the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the [Log groups page](#) on the CloudWatch console.
2. Choose the log group for your function (`/aws/lambda/your-function-name`).
3. Choose a log stream.

Each log stream corresponds to an [instance of your function \(p. 14\)](#). A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see [Error processor sample application for AWS Lambda \(p. 1015\)](#).

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)

- [AWS CLI – Quick configuration with aws configure](#)

You can use the [AWS CLI](#) to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{  
    "StatusCode": 200,  
    "LogResult":  
        "U1RBULQgUmVxdWVzdElk0iA4N2QwNDRi0C1mMTU0LTExZTgt0GNkYS0y0Tc0YzVlNGZiMjEgVmVyc21vb...",  
    "ExecutedVersion": "$LATEST"  
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \  
--query 'LogResult' --output text | base64 -d
```

You should see the following output:

```
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
```

The `base64` utility is available on Linux, macOS, and [Ubuntu on Windows](#). macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

```
#!/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 and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod +R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
  "StatusCode": 200,
  "ExecutedVersion": "$LATEST"
}
{
  "events": [
    {
      "timestamp": 1559763003171,
      "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST\n",
      "ingestionTime": 1559763003309
    },
    {
      "timestamp": 1559763003173,
      "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf\n\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\", \r ...",
      "ingestionTime": 1559763018353
    },
    {
      "timestamp": 1559763003173,
      "message": "\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",
      "ingestionTime": 1559763018353
    }
  ],
  "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
  "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.

AWS Lambda function errors in Node.js

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

This page describes how to view Lambda function invocation errors for the Node.js runtime using the Lambda console and the AWS CLI.

Sections

- [Syntax \(p. 278\)](#)
- [How it works \(p. 278\)](#)
- [Using the Lambda console \(p. 279\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 279\)](#)
- [Error handling in other AWS services \(p. 280\)](#)
- [What's next? \(p. 281\)](#)

Syntax

Example index.js file – Reference error

```
exports.handler = async function() {
    return x + 10
}
```

This code results in a reference error. Lambda catches the error and generates a JSON document with fields for the error message, the type, and the stack trace.

```
{
  "errorType": "ReferenceError",
  "errorMessage": "x is not defined",
  "trace": [
    "ReferenceError: x is not defined",
    "  at Runtime.exports.handler (/var/task/index.js:2:3)",
    "  at Runtime.handleOnce (/var/runtime/Runtime.js:63:25)",
    "  at process._tickCallback (internal/process/next_tick.js:68:7)"
  ]
}
```

How it works

When you invoke a Lambda function, Lambda receives the invocation request and validates the permissions in your execution role, verifies that the event document is a valid JSON document, and checks parameter values.

If the request passes validation, Lambda sends the request to a function instance. The [Lambda runtime \(p. 37\)](#) environment converts the event document into an object, and passes it to your function handler.

If Lambda encounters an error, it returns an exception type, message, and HTTP status code that indicates the cause of the error. The client or service that invoked the Lambda function can handle the error programmatically, or pass it along to an end user. The correct error handling behavior depends on the type of application, the audience, and the source of the error.

The following list describes the range of status codes you can receive from Lambda.

2xx

A 2xx series error with a X-Amz-Function-Error header in the response indicates a Lambda runtime or function error. A 2xx series status code indicates that Lambda accepted the request, but instead of an error code, Lambda indicates the error by including the X-Amz-Function-Error header in the response.

4xx

A 4xx series error indicates an error that the invoking client or service can fix by modifying the request, requesting permission, or by retrying the request. 4xx series errors other than 429 generally indicate an error with the request.

5xx

A 5xx series error indicates an issue with Lambda, or an issue with the function's configuration or resources. 5xx series errors can indicate a temporary condition that can be resolved without any action by the user. These issues can't be addressed by the invoking client or service, but a Lambda function's owner may be able to fix the issue.

For a complete list of invocation errors, see [InvokeFunction errors \(p. 1262\)](#).

Using the Lambda console

You can invoke your function on the Lambda console by configuring a test event and viewing the output. The output is captured in the function's execution logs and, when [active tracing \(p. 807\)](#) is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to test, and choose **Test**.
3. Under **Test event**, select **New event**.
4. Select a **Template**.
5. For **Name**, enter a name for the test. In the text entry box, enter the JSON test event.
6. Choose **Save changes**.
7. Choose **Test**.

The Lambda console invokes your function [synchronously \(p. 120\)](#) and displays the result. To see the response, logs, and other information, expand the **Details** section.

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

When you invoke a Lambda function in the AWS CLI, the AWS CLI splits the response into two documents. The AWS CLI response is displayed in your command prompt. If an error has occurred, the response contains a `FunctionError` field. The invocation response or error returned by the function is written to an output file. For example, `output.json` or `output.txt`.

The following [invoke](#) command example demonstrates how to invoke a function and write the invocation response to an `output.txt` file.

```
aws lambda invoke \
--function-name my-function \
--cli-binary-format raw-in-base64-out \
--payload '{"key1": "value1", "key2": "value2", "key3": "value3"}' output.txt
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

You should see the AWS CLI response in your command prompt:

```
{  
  "StatusCode": 200,  
  "FunctionError": "Unhandled",  
  "ExecutedVersion": "$LATEST"  
}
```

You should see the function invocation response in the `output.txt` file. In the same command prompt, you can also view the output in your command prompt using:

```
cat output.txt
```

You should see the invocation response in your command prompt.

```
{"errorType": "ReferenceError", "errorMessage": "x is not defined", "trace": ["ReferenceError: x is not defined", " at Runtime.exports.handler (/var/task/index.js:2:3)", " at Runtime.handleOnce (/var/runtime/Runtime.js:63:25)", " at process._tickCallback (internal/process/next_tick.js:68:7)"]}
```

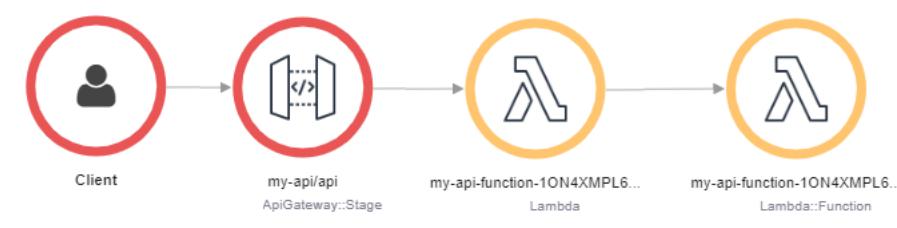
Lambda also records up to 256 KB of the error object in the function's logs. For more information, see [AWS Lambda function logging in Node.js \(p. 273\)](#).

Error handling in other AWS services

When another AWS service invokes your function, the service chooses the invocation type and retry behavior. AWS services can invoke your function on a schedule, in response to a lifecycle event on a resource, or to serve a request from a user. Some services invoke functions asynchronously and let Lambda handle errors, while others retry or pass errors back to the user.

For example, API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502 error code. To customize the error response, you must catch errors in your code and format a response in the required format.

We recommend using AWS X-Ray to determine the source of an error and its cause. X-Ray allows you to find out which component encountered an error, and see details about the errors. The following example shows a function error that resulted in a 502 response from API Gateway.



For more information, see [Instrumenting Node.js code in AWS Lambda \(p. 282\)](#).

What's next?

- Learn how to show logging events for your Lambda function on the [the section called "Logging" \(p. 273\)](#) page.

Instrumenting Node.js code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of two SDK libraries:

- [AWS Distro for OpenTelemetry \(ADOT\)](#) – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- [AWS X-Ray SDK for Node.js](#) – An SDK for generating and sending trace data to X-Ray.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

Important

The X-Ray and AWS Lambda Powertools SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see [Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs](#).

Sections

- [Using ADOT to instrument your Node.js functions \(p. 282\)](#)
- [Using the X-Ray SDK to instrument your Node.js functions \(p. 282\)](#)
- [Activating tracing with the Lambda console \(p. 283\)](#)
- [Activating tracing with the Lambda API \(p. 284\)](#)
- [Activating tracing with AWS CloudFormation \(p. 284\)](#)
- [Interpreting an X-Ray trace \(p. 284\)](#)
- [Storing runtime dependencies in a layer \(X-Ray SDK\) \(p. 286\)](#)

Using ADOT to instrument your Node.js functions

ADOT provides fully managed Lambda [layers \(p. 11\)](#) that package everything you need to collect telemetry data using the OTel SDK. By consuming this layer, you can instrument your Lambda functions without having to modify any function code. You can also configure your layer to do custom initialization of OTel. For more information, see [Custom configuration for the ADOT Collector on Lambda](#) in the ADOT documentation.

For Node.js runtimes, you can add the **AWS managed Lambda layer for ADOT Javascript** to automatically instrument your functions. For detailed instructions on how to add this layer, see [AWS Distro for OpenTelemetry Lambda Support for JavaScript](#) in the ADOT documentation.

Using the X-Ray SDK to instrument your Node.js functions

To record details about calls that your Lambda function makes to other resources in your application, you can also use the AWS X-Ray SDK for Node.js. To get the SDK, add the `aws-xray-sdk-core` package to your application's dependencies.

Example [blank-nodejs/package.json](#)

```
{  
  "name": "blank-nodejs",  
  "version": "1.0.0",  
  "private": true,  
  "devDependencies": {  
    "aws-sdk": "2.631.0",  
    "jest": "25.4.0"  
  },  
  "dependencies": {  
    "aws-xray-sdk-core": "1.1.2"  
  },  
  "scripts": {  
    "test": "jest"  
  }  
}
```

To instrument AWS SDK clients, wrap the aws-sdk library with the captureAWS method.

Example [blank-nodejs/function/index.js](#) – Tracing an AWS SDK client

```
const AWSXRay = require('aws-xray-sdk-core')  
const AWS = AWSXRay.captureAWS(require('aws-sdk'))  
  
// Create client outside of handler to reuse  
const lambda = new AWS.Lambda()  
  
// Handler  
exports.handler = async function(event, context) {  
  event.Records.forEach(record => {  
    ...  
  }  
}
```

The Lambda runtime sets some environment variables to configure the X-Ray SDK. For example, Lambda sets AWS_XRAY_CONTEXT_MISSING to LOG_ERROR to avoid throwing runtime errors from the X-Ray SDK. To set a custom context missing strategy, override the environment variable in your function configuration to have no value, and then you can set the context missing strategy programmatically.

Example Example initialization code

```
const AWSXRay = require('aws-xray-sdk-core');  
  
// Configure the context missing strategy to do nothing  
AWSXRay.setContextMissingStrategy(() => {});
```

For more information, see [the section called “Environment variables” \(p. 76\)](#).

After you add the correct dependencies and make the necessary code changes, activate tracing in your function's configuration via the Lambda console or the API.

Activating tracing with the Lambda console

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring and operations tools**.

4. Choose **Edit**.
5. Under **X-Ray**, toggle on **Active tracing**.
6. Choose **Save**.

Activating tracing with the Lambda API

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- [UpdateFunctionConfiguration \(p. 1377\)](#)
- [GetFunctionConfiguration \(p. 1229\)](#)
- [CreateFunction \(p. 1165\)](#)

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Activating tracing with AWS CloudFormation

To activate tracing on an `AWS::Lambda::Function` resource in an AWS CloudFormation template, use the `TracingConfig` property.

Example [function-inline.yml](#) – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
    ...
```

For an AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` resource, use the `Tracing` property.

Example [template.yml](#) – Tracing configuration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
    ...
```

Interpreting an X-Ray trace

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's [execution role \(p. 816\)](#). Otherwise, add the [AWSXRayDaemonWriteAccess](#) policy to the execution role.

After you've configured active tracing, you can observe specific requests through your application. The [X-Ray service graph](#) shows information about your application and all its components. The following example from the [error processor \(p. 1015\)](#) sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.

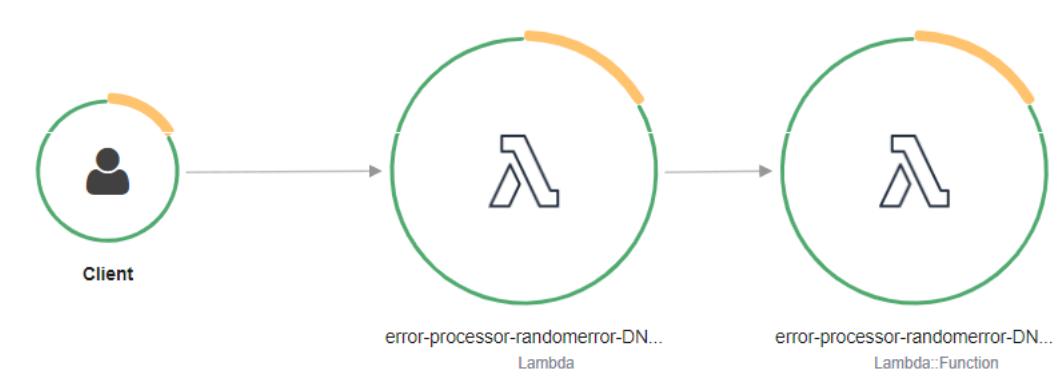


X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

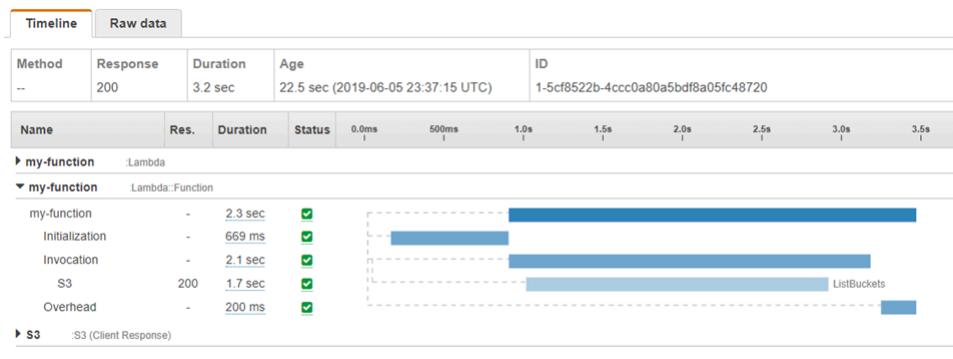
Note

You cannot configure the X-Ray sampling rate for your functions.

When using active tracing, Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes for the primary function from the [error processor sample application \(p. 1015\)](#).



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of AWS::Lambda and the other has origin AWS::Function.



This example expands the function segment to show its three subsegments:

- **Initialization** – Represents time spent loading your function and running [initialization code \(p. 13\)](#). This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see the [AWS X-Ray SDK for Node.js](#) in the [AWS X-Ray Developer Guide](#).

Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see [AWS X-Ray pricing](#).

Storing runtime dependencies in a layer (X-Ray SDK)

If you use the X-Ray SDK to instrument AWS SDK clients your function code, your deployment package can become quite large. To avoid uploading runtime dependencies every time you update your function code, package the X-Ray SDK in a [Lambda layer \(p. 93\)](#).

The following example shows an `AWS::Serverless::LayerVersion` resource that stores the AWS X-Ray SDK for Node.js.

Example [template.yml](#) – Dependencies layer

```
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:
        - nodejs12.x
```

With this configuration, you update the library layer only if you change your runtime dependencies. Since the function deployment package contains only your code, this can help reduce upload times.

Creating a layer for dependencies requires build changes to generate the layer archive prior to deployment. For a working example, see the [blank-nodejs](#) sample application.

Building Lambda functions with TypeScript

You can use the Node.js runtime to run TypeScript code in AWS Lambda. Because Node.js doesn't run TypeScript code natively, you must first transpile your TypeScript code into JavaScript. Then, use the JavaScript files to deploy your function code to Lambda. Your code runs in an environment that includes the AWS SDK for JavaScript, with credentials from an AWS Identity and Access Management (IAM) role that you manage.

Lambda supports the following Node.js runtimes.

Node.js

Name	Identifier	SDK	Operating system	Architectures	Deprecation (Phase 1)
Node.js 18	nodejs18.x	3.188.0	Amazon Linux 2	x86_64, arm64	
Node.js 16	nodejs16.x	2.1083.0	Amazon Linux 2	x86_64, arm64	
Node.js 14	nodejs14.x	2.1055.0	Amazon Linux 2	x86_64, arm64	
Node.js 12	nodejs12.x	2.1055.0	Amazon Linux 2	x86_64, arm64	Mar 31, 2023

Setting up a TypeScript development environment

Use a local integrated development environment (IDE), text editor, or [AWS Cloud9](#) to write your TypeScript function code. You can't create TypeScript code on the Lambda console.

To transpile your TypeScript code, set up a compiler such as [esbuild](#) or Microsoft's TypeScript compiler (`tsc`), which is bundled with the [TypeScript distribution](#). You can use the [AWS Serverless Application Model \(AWS SAM\)](#) or the [AWS Cloud Development Kit \(AWS CDK\)](#) to simplify building and deploying TypeScript code. Both tools use esbuild to transpile TypeScript code into JavaScript.

When using esbuild, consider the following:

- There are several [TypeScript caveats](#).
- You must configure your TypeScript transpilation settings to match the Node.js runtime that you plan to use. For more information, see [Target](#) in the esbuild documentation. For an example of a `tsconfig.json` file that demonstrates how to target a specific Node.js version supported by Lambda, refer to the [TypeScript GitHub repository](#).
- esbuild doesn't perform type checks. To check types, use the `tsc` compiler. Run `tsc -noEmit` or add a "noEmit" parameter to your `tsconfig.json` file, as shown in the following example. This configures `tsc` to not emit JavaScript files. After checking types, use esbuild to convert the TypeScript files into 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"]  
}
```

Topics

- [AWS Lambda function handler in TypeScript \(p. 290\)](#)
- [Deploy transpiled TypeScript code in Lambda with .zip file archives \(p. 293\)](#)
- [Deploy transpiled TypeScript code in Lambda with container images \(p. 298\)](#)
- [AWS Lambda context object in TypeScript \(p. 302\)](#)
- [AWS Lambda function logging in TypeScript \(p. 304\)](#)
- [AWS Lambda function errors in TypeScript \(p. 310\)](#)
- [Tracing TypeScript code in AWS Lambda \(p. 312\)](#)

AWS Lambda function handler in TypeScript

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. When the handler exits or returns a response, it becomes available to handle another event.

The following example function logs the contents of the event object and returns the location of the logs.

Example TypeScript handler

```
import { Handler } from 'aws-lambda';

export const handler: Handler = async (event, context) => {
    console.log('EVENT: \n' + JSON.stringify(event, null, 2));
    return context.logStreamName;
};
```

The runtime passes arguments to the handler method. The first argument is the event object, which contains information from the invoker. The invoker passes this information as a JSON-formatted string when it calls [Invoke \(p. 1260\)](#), and the runtime converts it to an object. When an AWS service invokes your function, the event structure [varies by service \(p. 556\)](#). With TypeScript, we recommend using type annotations for the event object. For more information, see [Using types for the event object \(p. 291\)](#).

The second argument is the [context object \(p. 302\)](#), which contains information about the invocation, function, and execution environment. In the preceding example, the function gets the name of the [log stream \(p. 304\)](#) from the context object and returns it to the invoker.

You can also use a callback argument, which is a function that you can call in non-async handlers to send a response. We recommend that you use `async/await` instead of callbacks. `Async/await` provides improved readability, error handling, and efficiency. For more information about the differences between `async/await` and callbacks, see [Using callbacks \(p. 291\)](#).

Using `async/await`

If your code performs an asynchronous task, use the `async/await` pattern to make sure that the handler finishes running. `Async/await` is a concise and readable way to write asynchronous code in Node.js, without the need for nested callbacks or chaining promises. With `async/await`, you can write code that reads like synchronous code, while still being asynchronous and non-blocking.

The `async` keyword marks a function as asynchronous, and the `await` keyword pauses the execution of the function until a Promise is resolved.

Example TypeScript function – asynchronous

This example uses `fetch`, which is available in the `nodejs18.x` runtime.

```
import { APIGatewayProxyEvent, APIGatewayProxyResult } from 'aws-lambda';
const url = 'https://aws.amazon.com/';
export const lambdaHandler = async (event: APIGatewayProxyEvent): Promise<APIGatewayProxyResult> => {
    try {
        // fetch is available with Node.js 18
        const res = await fetch(url);
        return {
            statusCode: res.status,
            body: JSON.stringify({
                message: await res.text(),
            })
        };
    }
};
```

```
        }),
    );
} catch (err) {
    console.log(err);
    return {
        statusCode: 500,
        body: JSON.stringify({
            message: 'some error happened',
        }),
    };
}
```

Using callbacks

We recommend that you use [async/await \(p. 290\)](#) to declare the function handler instead of using callbacks. Async/await is a better choice for several reasons:

- **Readability:** Async/await code is easier to read and understand than callback code, which can quickly become difficult to follow and result in callback hell.
- **Debugging and error handling:** Debugging callback-based code can be difficult. The call stack can become hard to follow and errors can easily be swallowed. With async/await, you can use try/catch blocks to handle errors.
- **Efficiency:** Callbacks often require switching between different parts of the code. Async/await can reduce the number of context switches, resulting in more efficient code.

When you use callbacks in your handler, the function continues to execute until the [event loop](#) is empty or the function times out. The response isn't sent to the invoker until all event loop tasks are finished. If the function times out, an error is returned instead. You can configure the runtime to send the response immediately by setting [context.callbackWaitsForEmptyEventLoop \(p. 302\)](#) to false.

The callback function takes two arguments: an `Error` and a response. The response object must be compatible with `JSON.stringify`.

Example TypeScript function with callback

This sample function receives an event from Amazon API Gateway, logs the event and context objects, and then returns a response to API Gateway.

```
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',
        }),
    });
}
```

Using types for the event object

We recommend that you don't use the `any` type for the handler arguments and return type because you lose the ability to check types. Instead, generate an event using the [sam local generate-event](#) AWS

Serverless Application Model CLI command, or use an open-source definition from the [@types/aws-lambda package](#).

Generating an event using the sam local generate-event command

1. Generate an Amazon Simple Storage Service (Amazon S3) proxy event.

```
sam local generate-event s3 put >> S3PutEvent.json
```

2. Use the [quicktype utility](#) to generate type definitions from the **S3PutEvent.json** file.

```
npm install -g quicktype
quicktype S3PutEvent.json -o S3PutEvent.ts
```

3. Use the generated types in your code.

```
import { S3PutEvent } from './S3PutEvent';

export const lambdaHandler = async (event: S3PutEvent): Promise<void> => {
  event.Records.map((record) => console.log(record.s3.object.key));
};
```

Generating an event using an open-source definition from the @types/aws-lambda package

1. Add the [@types/aws-lambda](#) package as a development dependency.

```
npm install -D @types/aws-lambda
```

2. Use the types in your code.

```
import { S3Event } from "aws-lambda";

export const lambdaHandler = async (event: S3Event): Promise<void> => {
  event.Records.map((record) => console.log(record.s3.object.key));
};
```

Deploy transpiled TypeScript code in Lambda with .zip file archives

Before you can deploy TypeScript code to AWS Lambda, you need to transpile it into JavaScript. This page explains three ways to build and deploy TypeScript code to Lambda with .zip file archives:

- [Using AWS Serverless Application Model \(AWS SAM\) \(p. 293\)](#)
- [Using the AWS Cloud Development Kit \(AWS CDK\) \(p. 294\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) and esbuild \(p. 296\)](#)

AWS SAM and AWS CDK simplify building and deploying TypeScript functions. The [AWS SAM template specification](#) provides a simple and clean syntax to describe the Lambda functions, APIs, permissions, configurations, and events that make up your serverless application. The [AWS CDK](#) lets you build reliable, scalable, cost-effective applications in the cloud with the considerable expressive power of a programming language. The AWS CDK is intended for moderately to highly experienced AWS users. Both the AWS CDK and the AWS SAM use esbuild to transpile TypeScript code into JavaScript.

Using AWS SAM to deploy TypeScript code to Lambda

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application using the AWS SAM. This application implements a basic API backend. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function is invoked. The function returns a `hello world` message.

Note

AWS SAM uses esbuild to create Node.js Lambda functions from TypeScript code. esbuild support is currently in public preview. During public preview, esbuild support may be subject to backwards incompatible changes.

Prerequisites

To complete the steps in this section, you must have the following:

- [AWS CLI version 2](#)
- [AWS SAM CLI version 1.39 or later](#)
- Node.js 14.x or later

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.

```
sam init --app-template hello-world-typescript --name sam-app --package-type Zip --  
runtime nodejs16.x
```

2. (Optional) The sample application includes configurations for commonly used tools, such as [ESLint](#) for code linting and [Jest](#) for unit testing. To run lint and test commands:

```
cd sam-app/hello-world  
npm install  
npm run lint
```

```
npm run test
```

3. Build the app.

```
cd sam-app  
sam build
```

4. Deploy the app.

```
sam deploy --guided
```

5. Follow the on-screen prompts. To accept the default options provided in the interactive experience, respond with Enter.
6. The output shows the endpoint for the REST API. Open the endpoint in a browser to test the function. You should see this response:

```
{"message": "hello world"}
```

7. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

Using the AWS CDK to deploy TypeScript code to Lambda

Follow the steps below to build and deploy a sample TypeScript application using the AWS CDK. This application implements a basic API backend. It consists of an API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function is invoked. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- [AWS CLI version 2](#)
- [AWS CDK version 2](#)
- Node.js 14.x or later

Deploy a sample AWS CDK application

1. Create a project directory for your new application.

```
mkdir hello-world  
cd hello-world
```

2. Initialize the app.

```
cdk init app --language typescript
```

3. Add the [@types/aws-lambda](#) package as a development dependency.

```
npm install -D @types/aws-lambda
```

4. Open the **lib** directory. You should see a file called **hello-world-stack.ts**. Create two new files in this directory: **hello-world.function.ts** and **hello-world.ts**.
5. Open **hello-world.function.ts** and add the following code to the file. This is the code for the Lambda function.

```
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. Open **hello-world.ts** and add the following code to the file. This contains the [NodejsFunction construct](#), which creates the Lambda function, and the [LambdaRestApi construct](#), which creates the REST API.

```
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,
        });
    }
}
```

The [NodejsFunction construct](#) assumes the following by default:

- Your function handler is called `handler`.
- The `.ts` file that contains the function code (**hello-world.function.ts**) is in the same directory as the `.ts` file that contains the construct (**hello-world.ts**). The construct uses the construct's ID ("hello-world") and the name of the Lambda handler file ("function") to find the function code. For example, if your function code is in a file called **hello-world.my-function.ts**, the **hello-world.ts** file must reference the function code like this:

```
const helloFunction = new NodejsFunction(this, 'my-function');
```

You can change this behavior and configure other esbuild parameters. For more information, see [Configuring esbuild](#) in the AWS CDK API reference.

7. Open **hello-world-stack.ts**. This is the code that defines your [AWS CDK stack](#). Replace the code with the following:

```
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. Deploy your application.

```
cdk deploy
```

9. The AWS CDK builds and packages the Lambda function using esbuild, and then deploys the function to the Lambda runtime. The output shows the endpoint for the REST API. Open the endpoint in a browser to test the function. You should see this response:

```
{"message": "hello world"}
```

This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

Using the AWS CLI and esbuild to deploy TypeScript code to Lambda

The following example demonstrates how to transpile and deploy TypeScript code to Lambda using esbuild and the AWS CLI. esbuild produces one JavaScript file with all dependencies. This is the only file that you need to add to the .zip archive.

Prerequisites

To complete the steps in this section, you must have the following:

- [AWS CLI version 2](#)
- Node.js 14.x or later
- An [execution role \(p. 816\)](#) for the Lambda function

Deploy a sample function

1. On your local machine, create a project directory for your new function.
2. Create a new Node.js project with npm or a package manager of your choice.

```
npm init
```

3. Add the [@types/aws-lambda](#) and [esbuild](#) packages as development dependencies.

```
npm install -D @types/aws-lambda esbuild
```

4. Create a new file called **index.ts**. Add the following code to the new file. This is the code for the Lambda function. The function returns a hello world message. The function doesn't create any API Gateway resources.

```
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. Add a build script to the **package.json** file. This configures esbuild to automatically create the .zip deployment package. For more information, see [Build scripts](#) in the esbuild documentation.

```
"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"
},
```

6. Build the package.

```
npm run build
```

7. Create a Lambda function using the .zip deployment package. Replace the highlighted text with the Amazon Resource Name (ARN) of your [execution role \(p. 816\)](#).

```
aws lambda create-function --function-name hello-world --runtime "nodejs16.x" --
role arn:aws:iam::123456789012:role/lambda-ex --zip-file "fileb://dist/index.zip" --
handler index.handler
```

8. [Run a test event \(p. 163\)](#) to confirm that the function returns the following response. If you want to invoke this function using API Gateway, [create and configure a REST API](#).

```
{
  "statusCode": 200,
  "body": "{\"message\":\"hello world\"}"
}
```

Deploy transpiled TypeScript code in Lambda with container images

You can deploy your TypeScript code to an AWS Lambda function as a Node.js [container image \(p. 881\)](#). AWS provides [base images \(p. 266\)](#) for Node.js to help you build the container image. These base images are preloaded with a language runtime and other components that are required to run the image on Lambda. AWS provides a Dockerfile for each of the base images to help with building your container image.

If you use a community or private enterprise base image, you must [add the Node.js runtime interface client \(RIC\) \(p. 270\)](#) to the base image to make it compatible with Lambda. For more information, see [Creating images from alternative base images \(p. 890\)](#).

Lambda provides a runtime interface emulator (RIE) for you to test your function locally. The base images for Lambda and base images for custom runtimes include the RIE. For other base images, you can download the RIE for [testing your image locally \(p. 884\)](#).

Using a Node.js base image to build and package TypeScript function code

Prerequisites

To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [Docker](#)
- Node.js 14.x or later

Creating an image from a base image

To create an image from an AWS base image for Lambda

1. On your local machine, create a project directory for your new function.
2. Create a new Node.js project with npm or a package manager of your choice.

```
npm init
```

3. Add the [@types/aws-lambda](#) and [esbuild](#) packages as development dependencies.

```
npm install -D @types/aws-lambda esbuild
```

4. Add a [build script](#) to the package.json file.

```
"scripts": {  
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --  
  target=es2020 --outfile=dist/index.js"  
}
```

5. Create a new file called `index.ts`. Add the following sample code to the new file. This is the code for the Lambda function. The function returns a `Hello World` message.

```
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. Create a new Dockerfile with the following configuration:

- Set the FROM property to the URI of the base image.
- Set the CMD argument to specify the Lambda function handler.

Example Dockerfile

The following Dockerfile uses a multi-stage build. The first step transpiles the TypeScript code into JavaScript. The second step produces a container image that contains only JavaScript files and production dependencies.

```
FROM public.ecr.aws/lambda/nodejs:16 as builder
WORKDIR /usr/app
COPY package.json index.ts .
RUN npm install
RUN npm run build

FROM public.ecr.aws/lambda/nodejs:16
WORKDIR ${LAMBDA_TASK_ROOT}
COPY --from=builder /usr/app/dist/* ./
CMD ["index.handler"]
```

7. Build your image.

```
docker build -t docker-image .
```

(Optional) Test the image locally

1. Start the Docker image with the **docker run** command. In this example, docker-image is the image name and test is the tag.

```
docker run -p 9000:8080 docker-image:test
```

This command runs the image as a container and creates a local endpoint at localhost:9000/2015-03-31/functions/function/invocations.

2. Test your application locally using the [RIE \(p. 884\)](#). From a new terminal window, post an event to the following endpoint using a **curl** command:

```
curl -XPOST "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function running in the container image and returns a response.

3. Get the container ID.

```
docker ps
```

4. Use the [docker kill](#) command to stop the container. In this command, replace 3766c4ab331c with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the [get-login-password](#) command to authenticate the Docker CLI to your Amazon ECR registry.
 - Set the --region value to the AWS Region where you want to create the Amazon ECR repository.
 - Replace 111122223333 with your AWS account 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. Create a repository in Amazon ECR using the [create-repository](#) command.

```
aws ecr create-repository --repository-name hello-world --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

If successful, you see a response like this:

```
{  
  "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. Copy the repositoryUri from the output in the previous step.
4. Run the [docker tag](#) command to tag your local image into your Amazon ECR repository as the latest version. In this command:
 - Replace docker-image:test with the name and [tag](#) of your Docker image.
 - Replace the Amazon ECR repository URI with the repositoryUri that you copied. Make sure to include :latest at the end of the URI.

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the [docker push](#) command to deploy your local image to the Amazon ECR repository. Make sure to include :latest at the end of the repository URI.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. [Create an execution role \(p. 191\)](#) for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.
7. Create the Lambda function. For ImageUri, specify the repository URI from earlier. Make sure to include :latest at the end of the URI.

```
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
```

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{  
    "ExecutedVersion": "$LATEST",  
    "StatusCode": 200  
}
```

9. To see the output of the function, check the response.json file.

To update the function code, you must create a new image version and store the image in the Amazon ECR repository. For more information, see [Updating function code \(p. 115\)](#).

AWS Lambda context object in TypeScript

When Lambda runs your function, it passes a context object to the [handler \(p. 290\)](#). This object provides methods and properties that provide information about the invocation, function, and execution environment.

Context methods

- `getRemainingTimeInMillis()` – Returns the number of milliseconds left before the execution times out.

Context properties

- `functionName` – The name of the Lambda function.
- `functionVersion` – The [version \(p. 83\)](#) of the function.
- `invokedFunctionArn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memoryLimitInMB` – The amount of memory that's allocated for the function.
- `awsRequestId` – The identifier of the invocation request.
- `logGroupName` – The log group for the function.
- `logStreamName` – The log stream for the function instance.
- `identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
 - `cognitoIdentityId` – The authenticated Amazon Cognito identity.
 - `cognitoIdentityPoolId` – The Amazon Cognito identity pool that authorized the invocation.
- `clientContext` – (mobile apps) Client context that's provided to Lambda by the client application.
 - `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` – Custom values that are set by the client application.
- `callbackWaitsForEmptyEventLoop` – Set to `false` to send the response right away when the [callback \(p. 291\)](#) runs, instead of waiting for the Node.js event loop to be empty. If this is `false`, any outstanding events continue to run during the next invocation.

You can use the [`@types/aws-lambda`](#) npm package to work with the context object.

Example index.ts file

The following example function logs context information and returns the location of the logs.

```
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;
};
```

AWS Lambda function logging in TypeScript

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

To output logs from your function code, you can use methods on the [console object](#), or any logging library that writes to `stdout` or `stderr`. For simple use cases, this approach might be sufficient.

This page describes how to produce log output from your Lambda function's code, or access logs using the AWS Command Line Interface, the Lambda console, the CloudWatch console, or Infrastructure as code tools such as the AWS Serverless Application Model(AWS SAM).

Sections

- [Tools and libraries \(p. 304\)](#)
- [Using AWS Lambda Powertools for TypeScript and AWS SAM for structured logging \(p. 304\)](#)
- [Using AWS Lambda Powertools for TypeScript and the AWS CDK for structured logging \(p. 306\)](#)
- [Using the Lambda console \(p. 309\)](#)
- [Using the CloudWatch console \(p. 309\)](#)

Tools and libraries

[AWS Lambda Powertools for TypeScript](#) is a developer toolkit to implement Serverless best practices and increase developer velocity. The [Logger utility](#) provides a Lambda optimized logger which includes additional information about function context across all your functions with output structured as JSON. Use this utility to do the following:

- Capture key fields from the Lambda context, cold start and structures logging output as JSON
- Log Lambda invocation events when instructed (disabled by default)
- Print all the logs only for a percentage of invocations via log sampling (disabled by default)
- Append additional keys to structured log at any point in time
- Use a custom log formatter (Bring Your Own Formatter) to output logs in a structure compatible with your organization's Logging RFC

Using AWS Lambda Powertools for TypeScript and AWS SAM for structured logging

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application with integrated [AWS Lambda Powertools for TypeScript](#) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a `hello world` message.

Prerequisites

To complete the steps in this section, you must have the following:

- Node.js 18.x or later

- [AWS CLI version 2](#)
- [AWS SAM CLI version 1.75 or later](#). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs18.x
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.

Note

For **HelloWorldFunction** may not have authorization defined, Is this okay?, make sure to enter y.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[? OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message": "hello world"}
```

7. To get the logs for the function, run [sam logs](#). For more information, see [Working with logs](#) in the [AWS Serverless Application Model Developer Guide](#).

```
sam logs --stack-name sam-app
```

The log output looks like this:

```
2023/01/31/[LATEST]4d53e8d279824834a1cccd35511a4949c 2022-08-31T09:33:10.552000 START
RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Version: $LATEST
2023/01/31/[LATEST]4d53e8d279824834a1cccd35511a4949c 2022-08-31T09:33:10.594000
2022-08-31T09:33:10.557Z 70693159-7e94-4102-a2af-98a6343fb8fb
INFO {"_aws": {"Timestamp": "1661938390556", "CloudWatchMetrics": [{"Namespace": "sam-app", "Dimensions": [{"service": "helloWorld"}], "Metrics": [{"Name": "ColdStart", "Unit": "Count"}]}]}, "service": "helloWorld", "ColdStart": 1}
2023/01/31/[LATEST]4d53e8d279824834a1cccd35511a4949c 2022-08-31T09:33:10.595000
2022-08-31T09:33:10.595Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
{"level": "INFO", "message": "This is an INFO log - sending HTTP 200 - hello world response", "service": "helloWorld", "timestamp": "2022-08-31T09:33:10.594Z"}
```

```
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.655000
2022-08-31T09:33:10.655Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO {"_aws": {
  "Timestamp": 1661938390655,
  "CloudWatchMetrics": [{"Namespace": "sam-app", "Dimensions": [{"service": "helloWorld"}], "Metrics": []}], "service": "helloWorld"
}
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.754000 END
RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.754000 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. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

Managing log retention

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which CloudWatch automatically deletes the logs. To set up log retention, add the following to your AWS SAM template:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Using AWS Lambda Powertools for TypeScript and the AWS CDK for structured logging

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application with integrated [AWS Lambda Powertools for TypeScript](#) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Node.js 18.x or later
- [AWS CLI version 2](#)
- [AWS CDK version 2](#)
- [AWS SAM CLI version 1.75 or later](#). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS CDK application

1. Create a project directory for your new application.

```
mkdir hello-world
cd hello-world
```

2. Initialize the app.

```
cdk init app --language typescript
```

3. Add the [@types/aws-lambda](#) package as a development dependency.

```
npm install -D @types/aws-lambda
```

4. Install the Powertools [Logger utility](#).

```
npm install @aws-lambda-powertools/logger
```

5. Open the **lib** directory. You should see a file called **hello-world-stack.ts**. Create new two new files in this directory: **hello-world.function.ts** and **hello-world.ts**.
6. Open **hello-world.function.ts** and add the following code to the file. This is the code for the Lambda function.

```
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. Open **hello-world.ts** and add the following code to the file. This contains the [NodejsFunction construct](#), which creates the Lambda function, configures environment variables for Powertools, and sets log retention to one week. It also includes the [LambdaRestApi construct](#), which creates the REST API.

```
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. Open **hello-world-stack.ts**. This is the code that defines your [AWS CDK stack](#). Replace the code with the following:

```
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. Go back to the project directory.

```
cd hello-world
```

10. Deploy your application.

```
cdk deploy
```

11. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

12. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message":"hello world"}
```

13. To get the logs for the function, run [sam logs](#). For more information, see [Working with logs](#) in the [AWS Serverless Application Model Developer Guide](#).

```
sam logs --stack-name HelloWorldStack
```

The log output looks like this:

```
2023/01/31[$LATEST]2ca67f180dcfd3e88b5d68576740c8e 2022-08-31T14:48:37.047000 START
RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Version: $LATEST
2023/01/31[$LATEST]2ca67f180dcfd3e88b5d68576740c8e 2022-08-31T14:48:37.050000 {
"level": "INFO",
"message": "This is an INFO log - sending HTTP 200 - hello world response",
"service": "helloWorld",
"timestamp": "2022-08-31T14:48:37.048Z",
```

```
"xray_trace_id": "1-630f74c4-2b080cf77680a04f2362bcf2"
}
2023/01/31/[$LATEST]2ca67f180cd4d3e88b5d68576740c8e 2022-08-31T14:48:37.082000 END
RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c
2023/01/31/[$LATEST]2ca67f180cd4d3e88b5d68576740c8e 2022-08-31T14:48:37.082000 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. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
cdk destroy
```

Using the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

Using the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the [Log groups page](#) on the CloudWatch console.
2. Choose the log group for your function (`/aws/lambda/your-function-name`).
3. Choose a log stream.

Each log stream corresponds to an [instance of your function \(p. 14\)](#). A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see [Error processor sample application for AWS Lambda \(p. 1015\)](#).

AWS Lambda function errors in TypeScript

If an exception occurs in TypeScript code that's transpiled into JavaScript, use source map files to determine where the error occurred. Source map files allow debuggers to map compiled JavaScript files to the TypeScript source code.

For example, the following code results in an error:

```
export const handler = async (event: unknown): Promise<unknown> => {
    throw new Error('Some exception');
};
```

AWS Lambda catches the error and generates a JSON document. However, this JSON document refers to the compiled JavaScript file (**app.js**), not the TypeScript source file.

```
{
  "errorType": "Error",
  "errorMessage": "Some exception",
  "stack": [
    "Error: Some exception",
    "    at Runtime.p [as handler] (/var/task/app.js:1:491)",
    "    at Runtime.handleOnce (/var/runtime/Runtime.js:66:25)"
  ]
}
```

To get an error response that maps to your TypeScript source file

Note

The following steps aren't valid for Lambda@Edge functions because Lambda@Edge doesn't support environment variables.

1. Generate a source map file with esbuild or another TypeScript compiler. Example:

```
esbuild app.ts --sourcemap --outfile=output.js
```

2. Add the source map to your deployment.
3. Turn on source maps for the Node.js runtime by adding `--enable-source-maps` to your `NODE_OPTIONS`.

Example for the AWS Serverless Application Model (AWS SAM)

```
Globals:
  Function:
    Environment:
      Variables:
        NODE_OPTIONS: '--enable-source-maps'
```

Make sure that the esbuild properties in your **template.yaml** file include `Sourcemap: true`. Example:

```
Metadata: # Manage esbuild properties
BuildMethod: esbuild
BuildProperties:
  Minify: true
  Target: "es2020"
  Sourcemap: true
EntryPoints:
```

- app.ts

Example Example for the AWS Cloud Development Kit (AWS CDK)

To use a source map with an AWS CDK application, add the following code to the file that contains the [NodejsFunction construct](#).

```
const helloFunction = new NodejsFunction(this, 'function', {
  bundling: {
    minify: true,
    sourceMap: true
  },
  environment: {
    NODE_OPTIONS: '--enable-source-maps',
  }
});
```

When you use a source map in your code, you get an error response similar to the following. This response shows that the error happened at line 2, column 11 in the `app.ts` file.

```
{
  "errorType": "Error",
  "errorMessage": "Some exception",
  "stack": [
    "Error: Some exception",
    "    at Runtime.p (/private/var/folders/3c/0d4wz7dn2y75bw_hxdwc0h6w0000gr/T/
tmpfmxb4ziy/app.ts:2:11)",
    "        at Runtime.handleOnce (/var/runtime/Runtime.js:66:25)"
  ]
}
```

Tracing TypeScript code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of three SDK libraries:

- [AWS Distro for OpenTelemetry \(ADOT\)](#) – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- [AWS X-Ray SDK for .NET](#) – An SDK for generating and sending trace data to X-Ray.
- [AWS Lambda Powertools for .NET](#) – A developer toolkit to implement Serverless best practices and increase developer velocity.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

Important

The X-Ray and AWS Lambda Powertools SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see [Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs](#).

Sections

- [Using AWS Lambda Powertools for TypeScript and AWS SAM for tracing \(p. 312\)](#)
- [Using AWS Lambda Powertools for TypeScript and the AWS CDK for tracing \(p. 314\)](#)
- [Interpreting an X-Ray trace \(p. 317\)](#)

Using AWS Lambda Powertools for TypeScript and AWS SAM for tracing

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application with integrated [AWS Lambda Powertools for TypeScript](#) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Node.js 18.x or later
- [AWS CLI version 2](#)
- [AWS SAM CLI version 1.75 or later](#). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs18.x --no-tracing
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.

Note

For **HelloWorldFunction** may not have authorization defined, Is this okay?, make sure to enter y.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[? OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message": "hello world"}
```

7. To get the traces for the function, run [sam traces](#).

```
sam traces
```

The trace output looks like this:

```
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. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

Note

You cannot configure the X-Ray sampling rate for your functions.

Using AWS Lambda Powertools for TypeScript and the AWS CDK for tracing

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application with integrated [AWS Lambda Powertools for TypeScript](#) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a `hello world` message.

Prerequisites

To complete the steps in this section, you must have the following:

- Node.js 18.x or later
- [AWS CLI version 2](#)
- [AWS CDK version 2](#)
- [AWS SAM CLI version 1.75 or later](#). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS Cloud Development Kit (AWS CDK) application

1. Create a project directory for your new application.

```
mkdir hello-world
cd hello-world
```

2. Initialize the app.

```
cdk init app --language typescript
```

3. Add the [@types/aws-lambda](#) package as a development dependency.

```
npm install -D @types/aws-lambda
```

4. Install the Powertools [Tracer utility](#).

```
npm install @aws-lambda-powertools/tracer
```

5. Open the `lib` directory. You should see a file called `hello-world-stack.ts`. Create new two new files in this directory: `hello-world.function.ts` and `hello-world.ts`.
6. Open `hello-world.function.ts` and add the following code to the file. This is the code for the Lambda function.

```
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. Open **hello-world.ts** and add the following code to the file. This contains the [NodejsFunction construct](#), which creates the Lambda function, configures environment variables for Powertools, and sets log retention to one week. It also includes the [LambdaRestApi construct](#), which creates the REST API.

```
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. Open **hello-world-stack.ts**. This is the code that defines your [AWS CDK stack](#). Replace the code with the following:

```
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. Deploy your application.

```
cd ..
cdk deploy
```

10. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

11. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message": "hello world"}
```

12. To get the traces for the function, run [sam traces](#).

```
sam traces
```

The trace output looks like this:

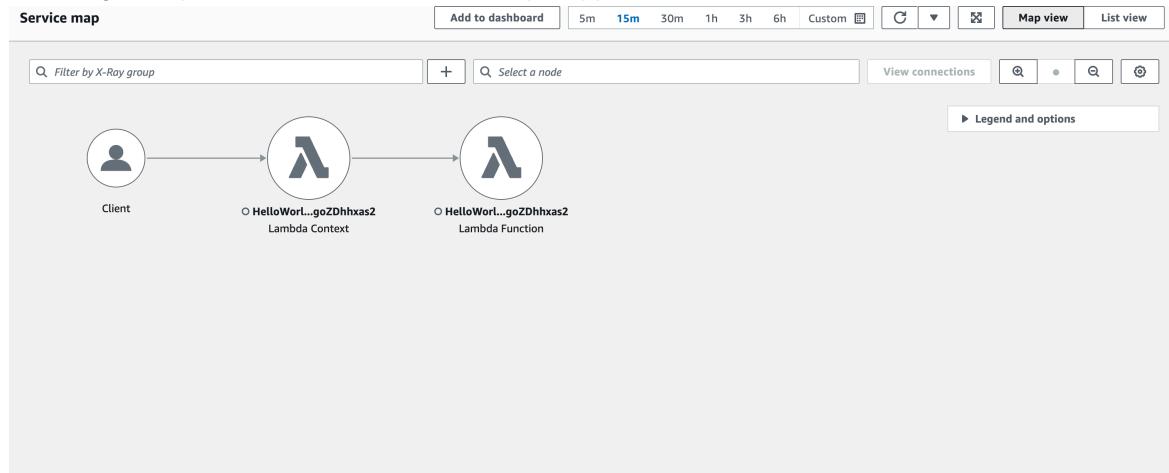
```
XRay Event [revision 1] at (2023-01-31T11:50:06.997000) with id
(1-11a2222-111a22222cb33de3b95daf9) and duration (0.449s)
- 0.350s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde [HTTP: 200]
- 0.157s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde
  - 0.169s - Initialization
  - 0.058s - Invocation
    - 0.055s - ## index.handler
      - 0.000s - ### MySubSegment
  - 0.099s - Overhead
```

13. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
cdk destroy
```

Interpreting an X-Ray trace

After you've configured active tracing, you can observe specific requests through your application. [CloudWatch ServiceLens](#) provides information about your application and all its components. The following example shows a trace from the sample application:



Building Lambda functions with Python

You can run Python code in AWS Lambda. Lambda provides [runtimes \(p. 37\)](#) for Python that run your code to process events. Your code runs in an environment that includes the SDK for Python (Boto3), with credentials from an AWS Identity and Access Management (IAM) role that you manage.

Lambda supports the following Python runtimes.

Python

Name	Identifier	SDK	Operating system	Architectures	Deprecation (Phase 1)
Python 3.10	python3.10	boto3-1.26.90 botocore-1.29.902	Amazon Linux	x86_64, arm64	
Python 3.9	python3.9	boto3-1.26.90 botocore-1.29.902	Amazon Linux	x86_64, arm64	
Python 3.8	python3.8	boto3-1.26.90 botocore-1.29.902	Amazon Linux	x86_64, arm64	
Python 3.7	python3.7	boto3-1.26.90 botocore-1.29.90	Amazon Linux	x86_64	

The runtime information in this table undergoes continuous updates. For more information on using AWS SDKs in Lambda, see [Managing AWS SDKs in Lambda functions](#).

Lambda functions use an [execution role \(p. 816\)](#) to get permission to write logs to Amazon CloudWatch Logs, and to access other services and resources. If you don't already have an execution role for function development, create one.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – Lambda.**
 - **Permissions – AWSLambdaBasicExecutionRole.**
 - **Role name – lambda-role.**

The **AWSLambdaBasicExecutionRole** policy has the permissions that the function needs to write logs to CloudWatch Logs.

You can add permissions to the role later, or swap it out for a different role that's specific to a single function.

To create a Python function

1. Open the [Lambda console](#).
2. Choose **Create function**.
3. Configure the following settings:
 - **Name** – **my-function**.
 - **Runtime** – **Python 3.9**.
 - **Role** – **Choose an existing role**.
 - **Existing role** – **lambda-role**.
4. Choose **Create function**.
5. To configure a test event, choose **Test**.
6. For **Event name**, enter **test**.
7. Choose **Save changes**.
8. To invoke the function, choose **Test**.

The console creates a Lambda function with a single source file named `lambda_function`. You can edit this file and add more files in the built-in [code editor \(p. 21\)](#). To save your changes, choose **Save**. Then, to run your code, choose **Test**.

Note

The Lambda console uses AWS Cloud9 to provide an integrated development environment in the browser. You can also use AWS Cloud9 to develop Lambda functions in your own environment. For more information, see [Working with Lambda Functions](#) in the AWS Cloud9 user guide.

Note

To get started with application development in your local environment, deploy one of the sample applications available in this guide's GitHub repository.

Sample Lambda applications in Python

- [blank-python](#) – A Python function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.

Your Lambda function comes with a CloudWatch Logs log group. The function runtime sends details about each invocation to CloudWatch Logs. It relays any [logs that your function outputs \(p. 334\)](#) during invocation. If your function [returns an error \(p. 346\)](#), Lambda formats the error and returns it to the invoker.

Topics

- [Lambda function handler in Python \(p. 320\)](#)
- [Deploy Python Lambda functions with .zip file archives \(p. 324\)](#)
- [Deploy Python Lambda functions with container images \(p. 329\)](#)
- [AWS Lambda context object in Python \(p. 332\)](#)
- [AWS Lambda function logging in Python \(p. 334\)](#)
- [AWS Lambda function errors in Python \(p. 346\)](#)
- [Instrumenting Python code in AWS Lambda \(p. 350\)](#)

Lambda function handler in Python

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. When the handler exits or returns a response, it becomes available to handle another event.

You can use the following general syntax when creating a function handler in Python:

```
def handler_name(event, context):
    ...
    return some_value
```

Naming

The Lambda function handler name specified at the time that you create a Lambda function is derived from:

- The name of the file in which the Lambda handler function is located.
- The name of the Python handler function.

A function handler can be any name; however, the default name in the Lambda console is `lambda_function.lambda_handler`. This function handler name reflects the function name (`lambda_handler`) and the file where the handler code is stored (`lambda_function.py`).

If you create a function in the console using a different file name or function handler name, you must edit the default handler name.

To change the function handler name (console)

1. Open the [Functions](#) page of the Lambda console and choose your function.
2. Choose the **Code** tab.
3. Scroll down to the **Runtime settings** pane and choose **Edit**.
4. In **Handler**, enter the new name for your function handler.
5. Choose **Save**.

How it works

When Lambda invokes your function handler, the [Lambda runtime \(p. 37\)](#) passes two arguments to the function handler:

- The first argument is the [event object](#). An event is a JSON-formatted document that contains data for a Lambda function to process. The [Lambda runtime \(p. 37\)](#) converts the event to an object and passes it to your function code. It is usually of the Python `dict` type. It can also be `list`, `str`, `int`, `float`, or the `NoneType` type.

The event object contains information from the invoking service. When you invoke a function, you determine the structure and contents of the event. When an AWS service invokes your function, the service defines the event structure. For more information about events from AWS services, see [Using AWS Lambda with other services \(p. 556\)](#).

- The second argument is the [context object \(p. 332\)](#). A context object is passed to your function by Lambda at runtime. This object provides methods and properties that provide information about the invocation, function, and runtime environment.

Returning a value

Optionally, a handler can return a value. What happens to the returned value depends on the [invocation type \(p. 118\)](#) and the [service \(p. 556\)](#) that invoked the function. For example:

- If you use the RequestResponse invocation type, such as [Synchronous invocation \(p. 120\)](#), AWS Lambda returns the result of the Python function call to the client invoking the Lambda function (in the HTTP response to the invocation request, serialized into JSON). For example, AWS Lambda console uses the RequestResponse invocation type, so when you invoke the function on the console, the console will display the returned value.
- If the handler returns objects that can't be serialized by `json.dumps`, the runtime returns an error.
- If the handler returns `None`, as Python functions without a `return` statement implicitly do, the runtime returns `null`.
- If you use the Event invocation type (an [asynchronous invocation \(p. 123\)](#)), the value is discarded.

Note

In Python 3.9 and later releases, Lambda includes the `requestId` of the invocation in the error response.

Examples

The following section shows examples of Python functions you can use with Lambda. If you use the Lambda console to author your function, you do not need to attach a [zip archive file \(p. 324\)](#) to run the functions in this section. These functions use standard Python libraries which are included with the Lambda runtime you selected. For more information, see [Lambda deployment packages \(p. 18\)](#).

Returning a message

The following example shows a function called `lambda_handler`. The function accepts user input of a first and last name, and returns a message that contains data from the event it received as input.

```
def lambda_handler(event, context):
    message = 'Hello {} {}!'.format(event['first_name'], event['last_name'])
    return {
        'message' : message
    }
```

You can use the following event data to invoke the function:

```
{
    "first_name": "John",
    "last_name": "Smith"
}
```

The response shows the event data passed as input:

```
{
    "message": "Hello John Smith!"
}
```

Parsing a response

The following example shows a function called `lambda_handler`. The function uses event data passed by Lambda at runtime. It parses the [environment variable \(p. 76\)](#) in `AWS_REGION` returned in the JSON response.

```
import os
import json

def lambda_handler(event, context):
    json_region = os.environ['AWS_REGION']
    return {
        "statusCode": 200,
        "headers": {
            "Content-Type": "application/json"
        },
        "body": json.dumps({
            "Region": json_region
        })
    }
```

You can use any event data to invoke the function:

```
{
    "key1": "value1",
    "key2": "value2",
    "key3": "value3"
}
```

Lambda runtimes set several environment variables during initialization. For more information on the environment variables returned in the response at runtime, see [Using AWS Lambda environment variables \(p. 76\)](#).

The function in this example depends on a successful response (in 200) from the Invoke API. For more information on the Invoke API status, see the [Invoke \(p. 1260\)](#) Response Syntax.

Returning a calculation

The following example shows a function called `lambda_handler`. The function accepts user input and returns a calculation to the user. For more information about this example, see the [aws-doc-sdk-examples GitHub repository](#).

```
import logging

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

def lambda_handler(event, context):
    ...
    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
```

You can use the following event data to invoke the function:

```
{
    "action": "increment",
    "number": 3
}
```

| }

Deploy Python Lambda functions with .zip file archives

Your AWS Lambda function's code consists of scripts or compiled programs and their dependencies. You use a *deployment package* to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives.

To create the deployment package for a .zip file archive, you can use a built-in .zip file archive utility or any other .zip file utility (such as [7zip](#)) for your command line tool. Note the following requirements for using a .zip file as your deployment package:

- The .zip file contains your function's code and any dependencies used to run your function's code (if applicable) on Lambda. If your function depends only on standard libraries, or AWS SDK libraries, you don't need to include these libraries in your .zip file. These libraries are included with the supported [Lambda runtime \(p. 37\)](#) environments.
- If the .zip file is larger than 50 MB, we recommend uploading it to your function from an Amazon Simple Storage Service (Amazon S3) bucket.
- If your deployment package contains native libraries, you can build the deployment package with AWS Serverless Application Model (AWS SAM). You can use the AWS SAM CLI `sam build` command with the `--use-container` to create your deployment package. This option builds a deployment package inside a Docker image that is compatible with the Lambda execution environment.

For more information, see [sam build](#) in the *AWS Serverless Application Model Developer Guide*.

- You need to build the deployment package to be compatible with this [instruction set architecture \(p. 29\)](#) of the function.
- Lambda uses POSIX file permissions, so you may need to [set permissions for the deployment package folder](#) before you create the .zip file archive.

Note

A python package may contain initialization code in the `__init__.py` file. Prior to Python 3.9, Lambda did not run the `__init__.py` code for packages in the function handler's directory or parent directories. In Python 3.9 and later releases, Lambda runs the init code for packages in these directories during initialization.

Note that Lambda runs the init code only when the execution environment is first initialized, not for each function invocation in that initialized environment.

Topics

- [Prerequisites \(p. 324\)](#)
- [What is a runtime dependency? \(p. 325\)](#)
- [Deployment package with no dependencies \(p. 325\)](#)
- [Deployment package with dependencies \(p. 325\)](#)
- [Using a virtual environment \(p. 326\)](#)
- [Deploy your .zip file to the function \(p. 327\)](#)

Prerequisites

Install the [AWS Command Line Interface \(AWS CLI\) version 2](#).

What is a runtime dependency?

A [deployment package \(p. 18\)](#) is required to create or update a Lambda function with or without runtime dependencies. The deployment package acts as the source bundle to run your function's code and dependencies (if applicable) on Lambda.

A dependency can be any package, module or other assembly dependency that is not included with the [Lambda runtime \(p. 37\)](#) environment for your function's code.

The following describes a Lambda function without runtime dependencies:

- If your function's code is in Python 3.8 or later, and it depends only on standard Python math and logging libraries, you don't need to include the libraries in your .zip file. These libraries are included with the Python runtime.
- If your function's code depends on the [AWS SDK for Python \(Boto3\)](#), you don't need to include the boto3 library in your .zip file. These libraries are included with Python3.8 and later runtimes.

Note: Lambda periodically updates the Boto3 libraries to enable the latest set of features and security updates. To have full control of the dependencies your function uses, package all of your dependencies with your deployment package.

Deployment package with no dependencies

Create the .zip file for your deployment package.

To create the deployment package

1. Open a command prompt and create a my-math-function project directory. For example, on macOS:

```
mkdir my-math-function
```

2. Navigate to the my-math-function project directory.

```
cd my-math-function
```

3. Copy the contents of the [sample Python code from GitHub](#) and save it in a new file named lambda_function.py. Your directory structure should look like this:

```
my-math-function$  
| lambda_function.py
```

4. Add the lambda_function.py file to the root of the .zip file.

```
zip my-deployment-package.zip lambda_function.py
```

This generates a my-deployment-package.zip file in your project directory. The command produces the following output:

```
adding: lambda_function.py (deflated 50%)
```

Deployment package with dependencies

Create the .zip file for your deployment package.

To create the deployment package

1. Open a command prompt and create a my-sourcecode-function project directory. For example, on macOS:

```
mkdir my-sourcecode-function
```

2. Navigate to the my-sourcecode-function project directory.

```
cd my-sourcecode-function
```

3. Copy the contents of the following sample Python code and save it in a new file named lambda_function.py:

```
import requests
def lambda_handler(event, context):
    response = requests.get("https://www.example.com/")
    print(response.text)
    return response.text
```

Your directory structure should look like this:

```
my-sourcecode-function$  
| lambda_function.py
```

4. Install the requests library to a new package directory.

```
pip install --target ./package requests
```

5. Create a deployment package with the installed library at the root.

```
cd package  
zip -r ../my-deployment-package.zip .
```

This generates a my-deployment-package.zip file in your project directory. The command produces the following output:

```
adding: chardet/ (stored 0%)
adding: chardet/enums.py (deflated 58%)
...

```

6. Add the lambda_function.py file to the root of the zip file.

```
cd ..  
zip my-deployment-package.zip lambda_function.py
```

Using a virtual environment

To update a Python function using a virtual environment

1. Activate the virtual environment. For example:

```
~/my-function$ source myvenv/bin/activate
```

2. Install libraries with pip.

```
(myvenv) ~/my-function$ pip install requests
```

3. Deactivate the virtual environment.

```
(myvenv) ~/my-function$ deactivate
```

4. Create a deployment package with the installed libraries at the root.

```
~/my-function$ cd myenv/lib/python3.10/site-packages  
zip -r ../../../../../my-deployment-package.zip .
```

The last command saves the deployment package to the root of the my-function directory.

Tip

A library may appear in site-packages or dist-packages and the first folder lib or lib64. You can use the pip show command to locate a specific package.

5. Add function code files to the root of your deployment package.

```
~/my-function/myenv/lib/python3.10/site-packages$ cd ../../../../../  
~/my-function$ zip -g my-deployment-package.zip lambda_function.py
```

After you complete this step, you should have the following directory structure:

```
my-deployment-package.zip$  
# lambda_function.py  
# __pycache__  
# certifi/  
# certifi-2020.6.20.dist-info/  
# chardet/  
# chardet-3.0.4.dist-info/  
...
```

Deploy your .zip file to the function

To deploy the new code to your function, you upload the new .zip file deployment package. You can use the [Lambda console \(p. 108\)](#) to upload a .zip file to the function, or you can use the [UpdateFunctionCode \(p. 1367\)](#) CLI command.

The following example uploads a file named **my-deployment-package.zip**. Use the [file://](#) file prefix to upload the binary .zip file to Lambda.

```
~/my-function$ aws lambda update-function-code --function-name MyLambdaFunction --zip-file  
fileb://my-deployment-package.zip  
{  
  "FunctionName": "mylambdafunction",  
  "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:mylambdafunction",  
  "Runtime": "python3.10",  
  "Role": "arn:aws:iam::123456789012:role/lambda-role",  
  "Handler": "lambda_function.lambda_handler",  
  "CodeSize": 5912988,  
  "CodeSha256": "A2P0NUWq1J+LtSbkuP8tm9uNYqs1TAa3M76ptmZCw5g=",  
  "Version": "$LATEST",  
  "RevisionId": "5afdc7dc-2fcf-4ca8-8f24-947939ca707f",  
  ...  
}
```


Deploy Python Lambda functions with container images

You can deploy your Lambda function code as a [container image \(p. 881\)](#). AWS provides the following resources to help you build a container image for your Python function:

- AWS base images for Lambda

These base images are preloaded with a language runtime and other components that are required to run the image on Lambda. AWS provides a Dockerfile for each of the base images to help with building your container image.

- Open-source runtime interface clients

If you use a community or private enterprise base image, you must add a [runtime interface client \(p. 883\)](#) to the base image to make it compatible with Lambda.

- Open-source runtime interface emulator

Lambda provides a runtime interface emulator (RIE) for you to test your function locally. The base images for Lambda and base images for custom .runtimes include the RIE. For other base images, you can download the RIE for [testing your image \(p. 884\)](#) locally.

The workflow for a function defined as a container image includes these steps:

1. Build your container image using the resources listed in this topic.
2. Upload the image to your [Amazon Elastic Container Registry \(Amazon ECR\) container registry \(p. 891\)](#).
3. [Create the function \(p. 113\)](#) or [update the function code \(p. 115\)](#) to deploy the image to an existing function.

Topics

- [AWS base images for Python \(p. 329\)](#)
- [Create a Python image from an AWS base image \(p. 330\)](#)
- [Create a Python image from an alternative base image \(p. 331\)](#)
- [Python runtime interface clients \(p. 331\)](#)
- [Deploy the container image \(p. 331\)](#)

AWS base images for Python

AWS provides the following base images for Python:

Tags	Runtime	Operating system	Dockerfile	Deprecation
3.10	Python 3.10	Amazon Linux 2	Dockerfile for Python 3.10 on GitHub	
3.9	Python 3.9	Amazon Linux 2	Dockerfile for Python 3.9 on GitHub	

Tags	Runtime	Operating system	Dockerfile	Deprecation
3.8	Python 3.8	Amazon Linux 2	Dockerfile for Python 3.8 on GitHub	
3.7	Python 3.7	Amazon Linux	Dockerfile for Python 3.7 on GitHub	

Amazon ECR repository: [gallery.ecr.aws/lambda/python](#)

Create a Python image from an AWS base image

When you build a container image for Python using an AWS base image, you only need to copy the python app to the container and install any dependencies.

If your function has dependencies, your local Python environment must match the version in the base image that you specify in the Dockerfile.

To build and deploy a Python function with the `python:3.8` base image.

1. On your local machine, create a project directory for your new function.
2. In your project directory, add a file named `app.py` containing your function code. The following example shows a simple Python handler.

```
import sys
def handler(event, context):
    return 'Hello from AWS Lambda using Python' + sys.version + '!'
```

3. In your project directory, add a file named `requirements.txt`. List each required library as a separate line in this file. Leave the file empty if there are no dependencies.
4. Use a text editor to create a Dockerfile in your project directory. The following example shows the Dockerfile for the handler that you created in the previous step. Install any dependencies under the `${LAMBDA_TASK_ROOT}` directory alongside the function handler to ensure that the Lambda runtime can locate them when the function is invoked.

```
FROM public.ecr.aws/lambda/python:3.8

# Install the function's dependencies using file requirements.txt
# from your project folder.

COPY requirements.txt .
RUN pip3 install -r requirements.txt --target "${LAMBDA_TASK_ROOT}"

# Copy function code
COPY app.py ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside of
# the Dockerfile)
CMD [ "app.handler" ]
```

5. To create the container image, follow steps 4 through 7 in [Create an image from an AWS base image for Lambda \(p. 888\)](#).

Create a Python image from an alternative base image

When you use an alternative base image, you need to install the [Python runtime interface client \(p. 331\)](#)

For an example of how to create a Python image from an Alpine base image, see [Container image support for Lambda](#) on the AWS Blog.

Python runtime interface clients

Install the [runtime interface client \(p. 883\)](#) for Python using the pip package manager:

```
pip install awslambdaric
```

For package details, see [Lambda RIC](#) on the Python Package Index (PyPI) website.

You can also download the [Python runtime interface client](#) from GitHub.

Deploy the container image

For a new function, you deploy the Python image when you [create the function \(p. 113\)](#). For an existing function, if you rebuild the container image, you need to redeploy the image by [updating the function code \(p. 115\)](#).

AWS Lambda context object in Python

When Lambda runs your function, it passes a context object to the [handler \(p. 320\)](#). This object provides methods and properties that provide information about the invocation, function, and execution environment. For more information on how the context object is passed to the function handler, see [Lambda function handler in Python \(p. 320\)](#).

Context methods

- `get_remaining_time_in_millis` – Returns the number of milliseconds left before the execution times out.

Context properties

- `function_name` – The name of the Lambda function.
- `function_version` – The [version \(p. 83\)](#) of the function.
- `invoked_function_arn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memory_limit_in_mb` – The amount of memory that's allocated for the function.
- `aws_request_id` – The identifier of the invocation request.
- `log_group_name` – The log group for the function.
- `log_stream_name` – The log stream for the function instance.
- `identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
 - `cognito_identity_id` – The authenticated Amazon Cognito identity.
 - `cognito_identity_pool_id` – The Amazon Cognito identity pool that authorized the invocation.
- `client_context` – (mobile apps) Client context that's provided to Lambda by the client application.
 - `client.installation_id`
 - `client.app_title`
 - `client.app_version_name`
 - `client.app_version_code`
 - `client.app_package_name`
 - `custom` – A dict of custom values set by the mobile client application.
 - `env` – A dict of environment information provided by the AWS SDK.

The following example shows a handler function that logs context information.

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())
```

In addition to the options listed above, you can also use the AWS X-Ray SDK for [Instrumenting Python code in AWS Lambda \(p. 350\)](#) to identify critical code paths, trace their performance and capture the data for analysis.

AWS Lambda function logging in Python

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

To output logs from your function code, you can use [logging library](#) that writes to stdout or stderr. For simple use cases, this approach might be sufficient.

This page describes how to produce log output from your Lambda function's code, or access logs using the AWS Command Line Interface, the Lambda console, the CloudWatch console, or Infrastructure as code tools such as the AWS Serverless Application Model(AWS SAM).

Sections

- [Tools and libraries \(p. 334\)](#)
- [Creating a function that returns logs \(p. 334\)](#)
- [Using AWS Lambda Powertools for Python and AWS SAM for structured logging \(p. 335\)](#)
- [Using AWS Lambda Powertools for Python and the AWS CDK for structured logging \(p. 338\)](#)
- [Using the Lambda console \(p. 342\)](#)
- [Using the CloudWatch console \(p. 342\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 343\)](#)
- [Deleting logs \(p. 345\)](#)
- [Logging library \(p. 345\)](#)

Tools and libraries

[AWS Lambda Powertools for Python](#) is a developer toolkit to implement Serverless best practices and increase developer velocity. The [Logger utility](#) provides a Lambda optimized logger which includes additional information about function context across all your functions with output structured as JSON. Use this utility to do the following:

- Capture key fields from the Lambda context, cold start and structures logging output as JSON
- Log Lambda invocation events when instructed (disabled by default)
- Print all the logs only for a percentage of invocations via log sampling (disabled by default)
- Append additional keys to structured log at any point in time
- Use a custom log formatter (Bring Your Own Formatter) to output logs in a structure compatible with your organization's Logging RFC

Creating a function that returns logs

To output logs from your function code, you can use the [print method](#), or any logging library that writes to stdout or stderr. The following example logs the values of the Amazon CloudWatch Logs group and stream for the function and the event object.

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 log format

```
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'})
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Duration: 15.74 ms Billed
Duration: 16 ms Memory Size: 128 MB Max Memory Used: 56 MB Init Duration: 130.49 ms
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1 SegmentId: 07f5xmpl2d1f6f85 Sampled:
true
```

The Python runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details.

Report Log

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TracId** – For traced requests, the [AWS X-Ray trace ID \(p. 807\)](#).
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

Using AWS Lambda Powertools for Python and AWS SAM for structured logging

Follow the steps below to download, build, and deploy a sample Hello World Python application with integrated [Powertools for Python](#) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Python 3.9
- [AWS CLI version 2](#)
- [AWS SAM CLI version 1.75 or later](#). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World Python template.

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type Zip  
--runtime python3.9 --no-tracing
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.

Note

For **HelloWorldFunction** may not have authorization defined, Is this okay?, make sure to enter y.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message": "hello world"}
```

7. To get the logs for the function, run [sam logs](#). For more information, see [Working with logs](#) in the *AWS Serverless Application Model Developer Guide*.

```
sam logs --stack-name sam-app
```

The log output looks like this:

```
2023/02/03/[LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:50.371000  
INIT_START Runtime Version: python:3.9.v16 Runtim Version ARN: arn:aws:lambda:us-east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525  
2023/02/03/[LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.112000 START  
RequestId: d455cf4-7704-46df-901b-2a5cce9405be Version: $LATEST  
2023/02/03/[LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.114000 {  
    "level": "INFO",  
    "location": "hello:23",  
    "message": "Hello world API - HTTP 200",  
    "timestamp": "2023-02-03 14:59:51,113+0000",  
    "service": "PowertoolsHelloWorld",  
    "cold_start": true,  
    "function_name": "sam-app-HelloWorldFunction-YBg8yfYt0c9j",  
    "function_memory_size": "128",  
    "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-HelloWorldFunction-YBg8yfYt0c9j",  
    "function_request_id": "d455cf4-7704-46df-901b-2a5cce9405be",
```

```
"correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
"xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2023/02/03/[${LATEST}]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "function_name",
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      ]
    ],
    "function_name": "sam-app-HelloWorldFunction-YBg8yfYt0c9j",
    "service": "PowertoolsHelloWorld",
    "ColdStart": [
      1.0
    ]
  }
}
2023/02/03/[${LATEST}]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
      ]
    ],
    "service": "PowertoolsHelloWorld",
    "HelloWorldInvocations": [
      1.0
    ]
  }
}
2023/02/03/[${LATEST}]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000 END
RequestId: d455cf4-7704-46df-901b-2a5cce9405be
2023/02/03/[${LATEST}]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000 REPORT
RequestId: d455cf4-7704-46df-901b-2a5cce9405be Duration: 16.33 ms Billed
Duration: 17 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. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

Managing log retention

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which CloudWatch automatically deletes the logs. To set up log retention, add the following to your AWS SAM template:

```
Resources:  
  HelloWorldFunction:  
    Type: AWS::Serverless::Function  
    Properties:  
      # Omitting other properties  
  
  LogGroup:  
    Type: AWS::Logs::LogGroup  
    Properties:  
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"  
      RetentionInDays: 7
```

Using AWS Lambda Powertools for Python and the AWS CDK for structured logging

Follow the steps below to download, build, and deploy a sample Hello World Python application with integrated [AWS Lambda Powertools for Python](#) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Python 3.9
- [AWS CLI version 2](#)
- [AWS CDK version 2](#)
- [AWS SAM CLI version 1.75 or later](#). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS CDK application

1. Create a project directory for your new application.

```
mkdir hello-world  
cd hello-world
```

2. Initialize the app.

```
cdk init app --language python
```

3. Install the Python dependencies.

```
pip install -r requirements.txt
```

4. Create a directory **lambda_function** under the root folder.

```
mkdir lambda_function
cd lambda_function
```

5. Create a file **app.py** and add the following code to the file. This is the code for the Lambda function.

```
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://awslabs.github.io/aws-lambda-powertools-python/latest/core/metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count, value=1)

    # structured log
    # See: https://awslabs.github.io/aws-lambda-powertools-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://awslabs.github.io/aws-lambda-powertools-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. Open the **hello_world** directory. You should see a file called **hello_world_stack.py**.

```
cd ..
cd hello_world
```

7. Open **hello_world_stack.py** and add the following code to the file. This contains the [Lambda Constructor](#), which creates the Lambda function, configures environment variables for Powertools and sets log retention to one week, and the [ApiGatewayv1 Constructor](#), which creates the REST API.

```
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, so 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://aws-labs.github.io/aws-  
            # lambda-powertools-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_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. Deploy your application.

```
cd ..  
cdk deploy
```

9. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query  
'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message": "hello world"}
```

- To get the logs for the function, run `sam logs`. For more information, see [Working with logs](#) in the *AWS Serverless Application Model Developer Guide*.

```
sam logs --stack-name HelloWorldStack
```

The log output looks like this:

```
2023/02/03/[LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:50.371000
  INIT_START Runtime: python:3.9.v16   Runtime Version ARN: arn:aws:lambda:us-
  east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2023/02/03/[LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.112000 START
  RequestId: d455cf4-7704-46df-901b-2a5cce9405be Version: $LATEST
2023/02/03/[LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.114000 {
    "level": "INFO",
    "location": "hello:23",
    "message": "Hello world API - HTTP 200",
    "timestamp": "2023-02-03 14:59:51,113+0000",
    "service": "PowertoolsHelloWorld",
    "cold_start": true,
    "function_name": "sam-app-HelloWorldFunction-YBg8yfYt0c9j",
    "function_memory_size": "128",
    "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYt0c9j",
    "function_request_id": "d455cf4-7704-46df-901b-2a5cce9405be",
    "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
    "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2023/02/03/[LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
    "_aws": {
        "Timestamp": 1675436391126,
        "CloudWatchMetrics": [
            {
                "Namespace": "Powertools",
                "Dimensions": [
                    [
                        "function_name",
                        "service"
                    ]
                ],
                "Metrics": [
                    {
                        "Name": "ColdStart",
                        "Unit": "Count"
                    }
                ]
            }
        ],
        "function_name": "sam-app-HelloWorldFunction-YBg8yfYt0c9j",
        "service": "PowertoolsHelloWorld",
        "ColdStart": [
            1.0
        ]
    }
}
2023/02/03/[LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
    "_aws": {
        "Timestamp": 1675436391126,
        "CloudWatchMetrics": [
            {
                "Namespace": "Powertools",
```

```
"Dimensions": [
    [
        "service"
    ]
],
"Metrics": [
    {
        "Name": "HelloWorldInvocations",
        "Unit": "Count"
    }
]
},
"service": "PowertoolsHelloWorld",
"HelloWorldInvocations": [
    1.0
]
}
2023/02/03[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000 END
RequestId: d455cf4-7704-46df-901b-2a5cce9405be
2023/02/03[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000 REPORT
RequestId: d455cf4-7704-46df-901b-2a5cce9405be Duration: 16.33 ms Billed
Duration: 17 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. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
cdk destroy
```

Using the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

Using the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the [Log groups page](#) on the CloudWatch console.
2. Choose the log group for your function (`/aws/lambda/your-function-name`).
3. Choose a log stream.

Each log stream corresponds to an [instance of your function \(p. 14\)](#). A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see [Error processor sample application for AWS Lambda \(p. 1015\)](#).

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

You can use the [AWS CLI](#) to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{  
    "StatusCode": 200,  
    "LogResult":  
        "U1RBULQgUmVxdWVzdElk0iA4N2QwNDRi0C1mMTU0LTExZTgt0GNkYS0y0Tc0YzVlNGZiMjEgVmVyc21vb...",  
    "ExecutedVersion": "$LATEST"  
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \  
--query 'LogResult' --output text | base64 -d
```

You should see the following output:

```
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
```

The `base64` utility is available on Linux, macOS, and [Ubuntu on Windows](#). macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

```
#!/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 and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf \tINFO\tENVIRONMENT VARIABLES\r{\r\t\t\"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\", \r\t\t...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf \tINFO\tEVENT\r{\r\t\t\"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"
}
```

Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.

Logging library

For more detailed logs, use the [logging library](#).

```
import os
import logging
logger = logging.getLogger()
logger.setLevel(logging.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)
```

The output from logger includes the log level, timestamp, and request 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'})}

[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: 3 ms Memory Size: 128 MB Max Memory Used: 56 MB Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370 SegmentId: 073cxmpl3e442861 Sampled:
true
```

AWS Lambda function errors in Python

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

This page describes how to view Lambda function invocation errors for the Python runtime using the Lambda console and the AWS CLI.

Sections

- [How it works \(p. 346\)](#)
- [Using the Lambda console \(p. 347\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 347\)](#)
- [Error handling in other AWS services \(p. 348\)](#)
- [Error examples \(p. 348\)](#)
- [Sample applications \(p. 349\)](#)
- [What's next? \(p. 281\)](#)

How it works

When you invoke a Lambda function, Lambda receives the invocation request and validates the permissions in your execution role, verifies that the event document is a valid JSON document, and checks parameter values.

If the request passes validation, Lambda sends the request to a function instance. The [Lambda runtime \(p. 37\)](#) environment converts the event document into an object, and passes it to your function handler.

If Lambda encounters an error, it returns an exception type, message, and HTTP status code that indicates the cause of the error. The client or service that invoked the Lambda function can handle the error programmatically, or pass it along to an end user. The correct error handling behavior depends on the type of application, the audience, and the source of the error.

The following list describes the range of status codes you can receive from Lambda.

2xx

A 2xx series error with a X-Amz-Function-Error header in the response indicates a Lambda runtime or function error. A 2xx series status code indicates that Lambda accepted the request, but instead of an error code, Lambda indicates the error by including the X-Amz-Function-Error header in the response.

4xx

A 4xx series error indicates an error that the invoking client or service can fix by modifying the request, requesting permission, or by retrying the request. 4xx series errors other than 429 generally indicate an error with the request.

5xx

A 5xx series error indicates an issue with Lambda, or an issue with the function's configuration or resources. 5xx series errors can indicate a temporary condition that can be resolved without any action by the user. These issues can't be addressed by the invoking client or service, but a Lambda function's owner may be able to fix the issue.

For a complete list of invocation errors, see [InvokeFunction errors \(p. 1262\)](#).

Using the Lambda console

You can invoke your function on the Lambda console by configuring a test event and viewing the output. The output is captured in the function's execution logs and, when [active tracing \(p. 807\)](#) is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to test, and choose **Test**.
3. Under **Test event**, select **New event**.
4. Select a **Template**.
5. For **Name**, enter a name for the test. In the text entry box, enter the JSON test event.
6. Choose **Save changes**.
7. Choose **Test**.

The Lambda console invokes your function [synchronously \(p. 120\)](#) and displays the result. To see the response, logs, and other information, expand the **Details** section.

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

When you invoke a Lambda function in the AWS CLI, the AWS CLI splits the response into two documents. The AWS CLI response is displayed in your command prompt. If an error has occurred, the response contains a `FunctionError` field. The invocation response or error returned by the function is written to an output file. For example, `output.json` or `output.txt`.

The following [invoke](#) command example demonstrates how to invoke a function and write the invocation response to an `output.txt` file.

```
aws lambda invoke \
--function-name my-function \
--cli-binary-format raw-in-base64-out \
--payload '{"key1": "value1", "key2": "value2", "key3": "value3"}' output.txt
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

You should see the AWS CLI response in your command prompt:

```
{  
    "StatusCode": 200,  
    "FunctionError": "Unhandled",  
    "ExecutedVersion": "$LATEST"  
}
```

You should see the function invocation response in the `output.txt` file. In the same command prompt, you can also view the output in your command prompt using:

```
cat output.txt
```

You should see the invocation response in your command prompt.

```
{"errorMessage": "'action'", "errorType": "KeyError", "stackTrace": ["  File \"/var/task/lambda_function.py\"", "  line 36, in lambda_handler\n      result = ACTIONS[event['action']]  
(event['number'])\\n"]}
```

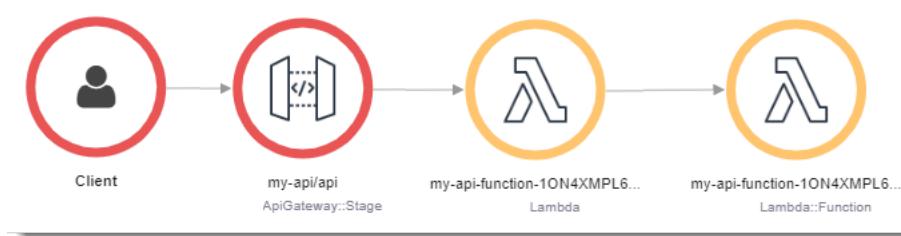
Lambda also records up to 256 KB of the error object in the function's logs. For more information, see [AWS Lambda function logging in Python \(p. 334\)](#).

Error handling in other AWS services

When another AWS service invokes your function, the service chooses the invocation type and retry behavior. AWS services can invoke your function on a schedule, in response to a lifecycle event on a resource, or to serve a request from a user. Some services invoke functions asynchronously and let Lambda handle errors, while others retry or pass errors back to the user.

For example, API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502 error code. To customize the error response, you must catch errors in your code and format a response in the required format.

We recommend using AWS X-Ray to determine the source of an error and its cause. X-Ray allows you to find out which component encountered an error, and see details about the errors. The following example shows a function error that resulted in a 502 response from API Gateway.



For more information, see [Instrumenting Python code in AWS Lambda \(p. 350\)](#).

Error examples

The following section shows common errors you may receive when creating, updating, or invoking your function using the Python [Lambda runtimes \(p. 37\)](#).

Example Runtime exception – ImportError

```
{  
  "errorMessage": "Unable to import module 'lambda_function': Cannot import name '_imaging'  
  from 'PIL' (/var/task/PIL/_init__.py)",  
  "errorType": "Runtime.ImportModuleError"  
}
```

This error is a result of using the AWS Command Line Interface (AWS CLI) to upload a deployment package that contains a C or C++ library. For example, the [Pillow \(PIL\)](#), [numpy](#), or [pandas](#) library.

We recommend using the AWS SAM CLI [sam build](#) command with the --use-container option to create your deployment package. Using the AWS SAM CLI with this option creates a Docker container with a Lambda-like environment that is compatible with Lambda.

Example JSON serialization error – Runtime.MarshalError

```
{  
    "errorMessage": "Unable to marshal response: Object of type AttributeError is not JSON  
    serializable",  
    "errorType": "Runtime.MarshalError"  
}
```

This error can be the result of the base64-encoding mechanism you are using in your function code. For example:

```
import base64  
encrypted_data = base64.b64encode(payload_enc).decode("utf-8")
```

This error can also be the result of not specifying your .zip file as a binary file when you created or updated your function. We recommend using the [fileb://](#) command option to upload your deployment package (.zip file).

```
aws lambda create-function --function-name my-function --zip-file fileb://my-deployment-  
package.zip --handler lambda_function.lambda_handler --runtime python3.8 --role  
arn:aws:iam::your-account-id:role/lambda-ex
```

Sample applications

The GitHub repository for this guide includes sample applications that demonstrate the use of the errors. Each sample application includes scripts for easy deployment and cleanup, an AWS Serverless Application Model (AWS SAM) template, and supporting resources.

Sample Lambda applications in Python

- [blank-python](#) – A Python function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.

What's next?

- Learn how to show logging events for your Lambda function on the [the section called "Logging" \(p. 334\)](#) page.

Instrumenting Python code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of three SDK libraries:

- [AWS Distro for OpenTelemetry \(ADOT\)](#) – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- [AWS X-Ray SDK for Python](#) – An SDK for generating and sending trace data to X-Ray.
- [AWS Lambda Powertools for Python](#) – A developer toolkit to implement Serverless best practices and increase developer velocity.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

Important

The X-Ray and AWS Lambda Powertools SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see [Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs](#).

Sections

- [Using AWS Lambda Powertools for Python and AWS SAM for tracing \(p. 350\)](#)
- [Using AWS Lambda Powertools for Python and the AWS CDK for tracing \(p. 338\)](#)
- [Using ADOT to instrument your Python functions \(p. 356\)](#)
- [Using the X-Ray SDK to instrument your Python functions \(p. 356\)](#)
- [Activating tracing with the Lambda console \(p. 356\)](#)
- [Activating tracing with the Lambda API \(p. 357\)](#)
- [Activating tracing with AWS CloudFormation \(p. 357\)](#)
- [Interpreting an X-Ray trace \(p. 357\)](#)
- [Storing runtime dependencies in a layer \(X-Ray SDK\) \(p. 359\)](#)

Using AWS Lambda Powertools for Python and AWS SAM for tracing

Follow the steps below to download, build, and deploy a sample Hello World Python application with integrated [AWS Lambda Powertools for Python](#) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Python 3.9
- [AWS CLI version 2](#)
- [AWS SAM CLI version 1.75 or later](#). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World Python template.

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type Zip  
--runtime python3.9 --no-tracing
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.

Note

For **HelloWorldFunction** may not have authorization defined, Is this okay?, make sure to enter y.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message": "hello world"}
```

7. To get the traces for the function, run [sam traces](#).

```
sam traces
```

The trace output looks like this:

```
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-YBg8yfYt0c9j -  
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-YBg8yfYt0c9j -  
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-YBg8yfYt0c9j - 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-YBg8yfYt0c9j [HTTP: 200]  
- 0.016s - sam-app-HelloWorldFunction-YBg8yfYt0c9j  
- 0.739s - Initialization  
- 0.016s - Invocation  
- 0.013s - ## lambda_handler  
- 0.000s - ## app.hello  
- 0.000s - Overhead
```

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

Note

You cannot configure the X-Ray sampling rate for your functions.

Using AWS Lambda Powertools for Python and the AWS CDK for tracing

Follow the steps below to download, build, and deploy a sample Hello World Python application with integrated [AWS Lambda Powertools for Python](#) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Python 3.9
- [AWS CLI version 2](#)
- [AWS CDK version 2](#)
- [AWS SAM CLI version 1.75 or later](#). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS CDK application

1. Create a project directory for your new application.

```
mkdir hello-world
cd hello-world
```

2. Initialize the app.

```
cdk init app --language python
```

3. Install the Python dependencies.

```
pip install -r requirements.txt
```

4. Create a directory **lambda_function** under the root folder.

```
mkdir lambda_function
cd lambda_function
```

5. Create a file **app.py** and add the following code to the file. This is the code for the Lambda function.

```
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://awslabs.github.io/aws-lambda-powertools-python/latest/core/metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count, value=1)

    # structured log
    # See: https://awslabs.github.io/aws-lambda-powertools-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://awslabs.github.io/aws-lambda-powertools-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. Open the **hello_world** directory. You should see a file called **hello_world_stack.py**.

```
cd ..
```

```
cd hello_world
```

7. Open **hello_world_stack.py** and add the following code to the file. This contains the [Lambda Constructor](#), which creates the Lambda function, configures environment variables for Powertools and sets log retention to one week, and the [ApiGatewayv1 Constructor](#), which creates the REST API.

```
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, so 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://aws-labs.github.io/aws-
            # lambda-powertools-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_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. Deploy your application.

```
cd ..
cdk deploy
```

9. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message": "hello world"}
```

11. To get the traces for the function, run [sam traces](#).

```
sam traces
```

The traces output looks like this:

```
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-YBg8yfYt0c9j -
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-YBg8yfYt0c9j -
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-YBg8yfYt0c9j - 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-YBg8yfYt0c9j [HTTP: 200]
- 0.016s - sam-app-HelloWorldFunction-YBg8yfYt0c9j
- 0.739s - Initialization
- 0.016s - Invocation
- 0.013s - ## lambda_handler
- 0.000s - ## app.hello
- 0.000s - Overhead
```

12. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
cdk destroy
```

Using ADOT to instrument your Python functions

ADOT provides fully managed Lambda [layers \(p. 11\)](#) that package everything you need to collect telemetry data using the OTel SDK. By consuming this layer, you can instrument your Lambda functions without having to modify any function code. You can also configure your layer to do custom initialization of OTel. For more information, see [Custom configuration for the ADOT Collector on Lambda](#) in the ADOT documentation.

For Python runtimes, you can add the **AWS managed Lambda layer for ADOT Python** to automatically instrument your functions. This layer works for both arm64 and x86_64 architectures. For detailed instructions on how to add this layer, see [AWS Distro for OpenTelemetry Lambda Support for Python](#) in the ADOT documentation.

Using the X-Ray SDK to instrument your Python functions

To record details about calls that your Lambda function makes to other resources in your application, you can also use the AWS X-Ray SDK for Python. To get the SDK, add the `aws-xray-sdk` package to your application's dependencies.

Example [requirements.txt](#)

```
jsonpickle==1.3
aws-xray-sdk==2.4.3
```

In your function code, you can instrument AWS SDK clients by patching the `boto3` library with the `aws_xray_sdk.core` module.

Example [function – Tracing an 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)))
    ...
```

After you add the correct dependencies and make the necessary code changes, activate tracing in your function's configuration via the Lambda console or the API.

Activating tracing with the Lambda console

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.

3. Choose **Configuration** and then choose **Monitoring and operations tools**.
4. Choose **Edit**.
5. Under **X-Ray**, toggle on **Active tracing**.
6. Choose **Save**.

Activating tracing with the Lambda API

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- [UpdateFunctionConfiguration \(p. 1377\)](#)
- [GetFunctionConfiguration \(p. 1229\)](#)
- [CreateFunction \(p. 1165\)](#)

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Activating tracing with AWS CloudFormation

To activate tracing on an AWS::Lambda::Function resource in an AWS CloudFormation template, use the **TracingConfig** property.

Example [function-inline.yml](#) – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
    ...
```

For an AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function resource, use the **Tracing** property.

Example [template.yml](#) – Tracing configuration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
    ...
```

Interpreting an X-Ray trace

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's [execution role \(p. 816\)](#). Otherwise, add the [AWSXRayDaemonWriteAccess](#) policy to the execution role.

After you've configured active tracing, you can observe specific requests through your application. The [X-Ray service graph](#) shows information about your application and all its components. The following example from the [error processor \(p. 1015\)](#) sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.

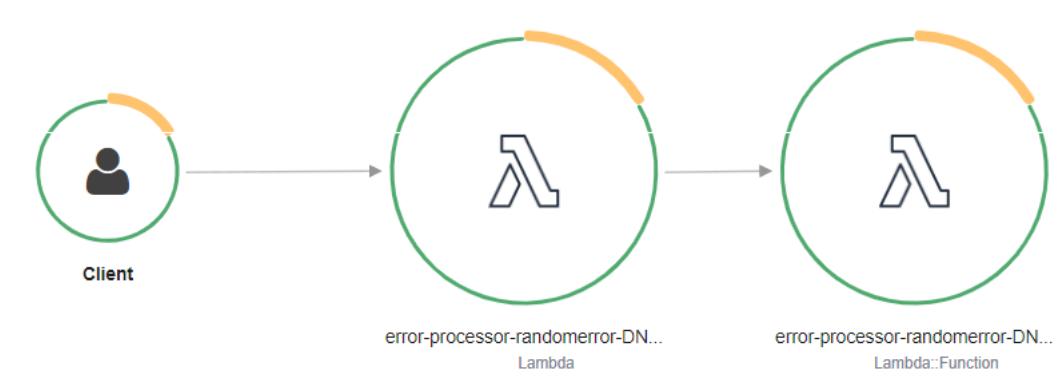


X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

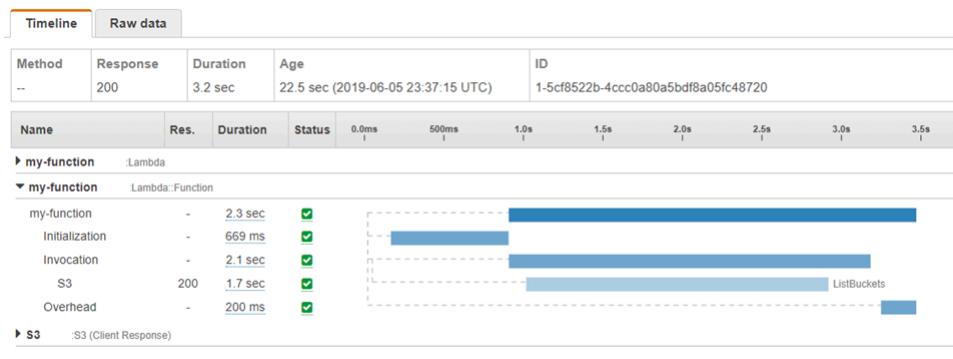
Note

You cannot configure the X-Ray sampling rate for your functions.

When using active tracing, Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes for the primary function from the [error processor sample application \(p. 1015\)](#).



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of AWS::Lambda and the other has origin AWS::Lambda::Function.



This example expands the function segment to show its three subsegments:

- **Initialization** – Represents time spent loading your function and running [initialization code \(p. 13\)](#). This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see the [AWS X-Ray SDK for Python](#) in the [AWS X-Ray Developer Guide](#).

Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see [AWS X-Ray pricing](#).

Storing runtime dependencies in a layer (X-Ray SDK)

If you use the X-Ray SDK to instrument AWS SDK clients your function code, your deployment package can become quite large. To avoid uploading runtime dependencies every time you update your function code, package the X-Ray SDK in a [Lambda layer \(p. 93\)](#).

The following example shows an `AWS::Serverless::LayerVersion` resource that stores the AWS X-Ray SDK for Python.

Example [template.yml](#) – Dependencies layer

```
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:
        - python3.8
```

With this configuration, you update the library layer only if you change your runtime dependencies. Since the function deployment package contains only your code, this can help reduce upload times.

Creating a layer for dependencies requires build changes to generate the layer archive prior to deployment. For a working example, see the [blank-python](#) sample application.

Building Lambda functions with Ruby

You can run Ruby code in AWS Lambda. Lambda provides [runtimes \(p. 37\)](#) for Ruby that run your code to process events. Your code runs in an environment that includes the AWS SDK for Ruby, with credentials from an AWS Identity and Access Management (IAM) role that you manage.

Lambda supports the following Ruby runtimes.

Ruby

Name	Identifier	SDK	Operating system	Architectures	Deprecation (Phase 1)
Ruby 2.7*	ruby2.7	3.1.0	Amazon Linux 2	x86_64, arm64	Nov 15, 2023

*The deprecation date for Ruby 2.7 above is an estimate. Lambda will support the Ruby 2.7 runtime for at least 6 months after the general availability (GA) release of the Ruby 3.2 runtime.

Lambda functions use an [execution role \(p. 816\)](#) to get permission to write logs to Amazon CloudWatch Logs, and to access other services and resources. If you don't already have an execution role for function development, create one.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – Lambda.**
 - **Permissions – AWSLambdaBasicExecutionRole.**
 - **Role name – lambda-role.**

The **AWSLambdaBasicExecutionRole** policy has the permissions that the function needs to write logs to CloudWatch Logs.

You can add permissions to the role later, or swap it out for a different role that's specific to a single function.

To create a Ruby function

1. Open the [Lambda console](#).
2. Choose **Create function**.
3. Configure the following settings:
 - **Name – my-function.**
 - **Runtime – Ruby 2.7.**

- **Role – Choose an existing role.**
 - **Existing role – lambda-role.**
4. Choose **Create function**.
 5. To configure a test event, choose **Test**.
 6. For **Event name**, enter **test**.
 7. Choose **Save changes**.
 8. To invoke the function, choose **Test**.

The console creates a Lambda function with a single source file named `lambda_function.rb`. You can edit this file and add more files in the built-in [code editor \(p. 21\)](#). To save your changes, choose **Save**. Then, to run your code, choose **Test**.

Note

The Lambda console uses AWS Cloud9 to provide an integrated development environment in the browser. You can also use AWS Cloud9 to develop Lambda functions in your own environment. For more information, see [Working with Lambda Functions](#) in the AWS Cloud9 user guide.

The `lambda_function.rb` file exports a function named `lambda_handler` that takes an event object and a context object. This is the [handler function \(p. 364\)](#) that Lambda calls when the function is invoked. The Ruby function runtime gets invocation events from Lambda and passes them to the handler. In the function configuration, the handler value is `lambda_function.lambda_handler`.

When you save your function code, the Lambda console creates a .zip file archive deployment package. When you develop your function code outside of the console (using an IDE) you need to [create a deployment package \(p. 365\)](#) to upload your code to the Lambda function.

Note

To get started with application development in your local environment, deploy one of the sample applications available in this guide's GitHub repository.

Sample Lambda applications in Ruby

- [blank-ruby](#) – A Ruby function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.
- [Ruby Code Samples for AWS Lambda](#) – Code samples written in Ruby that demonstrate how to interact with AWS Lambda.

The function runtime passes a context object to the handler, in addition to the invocation event. The [context object \(p. 371\)](#) contains additional information about the invocation, the function, and the execution environment. More information is available from environment variables.

Your Lambda function comes with a CloudWatch Logs log group. The function runtime sends details about each invocation to CloudWatch Logs. It relays any [logs that your function outputs \(p. 372\)](#) during invocation. If your function [returns an error \(p. 377\)](#), Lambda formats the error and returns it to the invoker.

Topics

- [AWS Lambda function handler in Ruby \(p. 364\)](#)
- [Deploy Ruby Lambda functions with .zip file archives \(p. 365\)](#)
- [Deploy Ruby Lambda functions with container images \(p. 368\)](#)
- [AWS Lambda context object in Ruby \(p. 371\)](#)
- [AWS Lambda function logging in Ruby \(p. 372\)](#)
- [AWS Lambda function errors in Ruby \(p. 377\)](#)

- [Instrumenting Ruby code in AWS Lambda \(p. 381\)](#)

AWS Lambda function handler in Ruby

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. When the handler exits or returns a response, it becomes available to handle another event.

In the following example, the file `function.rb` defines a handler method named `handler`. The handler function takes two objects as input and returns a JSON document.

Example `function.rb`

```
require 'json'

def handler(event:, context:)
    { event: JSON.generate(event), context: JSON.generate(context.inspect) }
end
```

In your function configuration, the `handler` setting tells Lambda where to find the handler. For the preceding example, the correct value for this setting is `function.handler`. It includes two names separated by a dot: the name of the file and the name of the handler method.

You can also define your handler method in a class. The following example defines a handler method named `process` on a class named `Handler` in a module named `LambdaFunctions`.

Example `source.rb`

```
module LambdaFunctions
  class Handler
    def self.process(event:, context:)
      "Hello!"
    end
  end
end
```

In this case, the handler setting is `source.LambdaFunctions::Handler.process`.

The two objects that the handler accepts are the invocation event and context. The event is a Ruby object that contains the payload that's provided by the invoker. If the payload is a JSON document, the event object is a Ruby hash. Otherwise, it's a string. The [context object \(p. 371\)](#) has methods and properties that provide information about the invocation, the function, and the execution environment.

The function handler is executed every time your Lambda function is invoked. Static code outside of the handler is executed once per instance of the function. If your handler uses resources like SDK clients and database connections, you can create them outside of the handler method to reuse them for multiple invocations.

Each instance of your function can process multiple invocation events, but it only processes one event at a time. The number of instances processing an event at any given time is your function's *concurrency*. For more information about the Lambda execution environment, see [Lambda execution environment \(p. 14\)](#).

Deploy Ruby Lambda functions with .zip file archives

Your AWS Lambda function's code consists of scripts or compiled programs and their dependencies. You use a *deployment package* to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives.

To create the deployment package for a .zip file archive, you can use a built-in .zip file archive utility or any other .zip file utility (such as [7zip](#)) for your command line tool. Note the following requirements for using a .zip file as your deployment package:

- The .zip file contains your function's code and any dependencies used to run your function's code (if applicable) on Lambda. If your function depends only on standard libraries, or AWS SDK libraries, you don't need to include these libraries in your .zip file. These libraries are included with the supported [Lambda runtime \(p. 37\)](#) environments.
- If the .zip file is larger than 50 MB, we recommend uploading it to your function from an Amazon Simple Storage Service (Amazon S3) bucket.
- If your deployment package contains native libraries, you can build the deployment package with AWS Serverless Application Model (AWS SAM). You can use the AWS SAM CLI `sam build` command with the `--use-container` to create your deployment package. This option builds a deployment package inside a Docker image that is compatible with the Lambda execution environment.

For more information, see [sam build](#) in the *AWS Serverless Application Model Developer Guide*.

- You need to build the deployment package to be compatible with this [instruction set architecture \(p. 29\)](#) of the function.
- Lambda uses POSIX file permissions, so you may need to [set permissions for the deployment package folder](#) before you create the .zip file archive.

Sections

- [Prerequisites \(p. 365\)](#)
- [Tools and libraries \(p. 365\)](#)
- [Updating a function with no dependencies \(p. 366\)](#)
- [Updating a function with additional dependencies \(p. 366\)](#)

Prerequisites

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

Tools and libraries

Lambda provides the following tools and libraries for the Ruby runtime:

Tools and libraries for Ruby

- [AWS SDK for Ruby](#): the official AWS SDK for the Ruby programming language.

Updating a function with no dependencies

To update a function by using the Lambda API, use the [UpdateFunctionCode \(p. 1367\)](#) operation. Create an archive that contains your function code, and upload it using the AWS Command Line Interface (AWS CLI).

To update a Ruby function with no dependencies

1. Create a .zip file archive.

```
zip function.zip function.rb
```

2. To upload the package, use the update-function-code command.

```
aws lambda update-function-code --function-name my-function --zip-file fileb://function.zip
```

You should see the following output:

```
{  
    "FunctionName": "my-function",  
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",  
    "Runtime": "ruby2.5",  
    "Role": "arn:aws:iam::123456789012:role/lambda-role",  
    "Handler": "function.handler",  
    "CodeSha256": "Qf0hMc1I2di6YFMi9aXm3JtGTmcDbjniEuiYonYptAk=",  
    "Version": "$LATEST",  
    "TracingConfig": {  
        "Mode": "Active"  
    },  
    "RevisionId": "983ed1e3-ca8e-434b-8dc1-7d72ebadd83d",  
    ...  
}
```

Updating a function with additional dependencies

If your function depends on libraries other than the AWS SDK for Ruby, install them to a local directory with [Bundler](#), and include them in your deployment package.

To update a Ruby function with dependencies

1. Install libraries in the vendor directory using the bundle command.

```
bundle config set --local path 'vendor/bundle' \  
bundle install
```

You should see the following output:

```
Fetching gem metadata from https://rubygems.org/.....  
Resolving dependencies...  
Fetching aws-eventstream 1.0.1  
Installing aws-eventstream 1.0.1  
...
```

This installs the gems in the project directory instead of the system location, and sets vendor/bundle as the default path for future installations.

Note

The bundle config command in this step may add a trailing space to your BUNDLE_PATH variable. To avoid deployment errors, check that the value of BUNDLE_PATH ("vendor/bundle") does not contain a trailing space.

To later install gems globally, use `bundle config set --local system 'true'`.

2. Create a .zip file archive.

```
zip -r function.zip function.rb vendor
```

You should see the following output:

```
adding: function.rb (deflated 37%)
adding: vendor/ (stored 0%)
adding: vendor/bundle/ (stored 0%)
adding: vendor/bundle/ruby/ (stored 0%)
adding: vendor/bundle/ruby/2.7.0/ (stored 0%)
adding: vendor/bundle/ruby/2.7.0/build_info/ (stored 0%)
adding: vendor/bundle/ruby/2.7.0/cache/ (stored 0%)
adding: vendor/bundle/ruby/2.7.0/cache/aws-eventstream-1.0.1.gem (deflated 36%)
...
```

3. Update the function code.

```
aws lambda update-function-code --function-name my-function --zip-file fileb://
function.zip
```

You should see the following output:

```
{
  "FunctionName": "my-function",
  "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
  "Runtime": "ruby2.5",
  "Role": "arn:aws:iam::123456789012:role/lambda-role",
  "Handler": "function.handler",
  "CodeSize": 300,
  "CodeSha256": "Qf0hMc1I2di6YFMi9aXm3JtGTmcDbjniEuiYonYptAk=",
  "Version": "$LATEST",
  "RevisionId": "983ed1e3-ca8e-434b-8dc1-7d72ebadd83d",
  ...
}
```

Deploy Ruby Lambda functions with container images

You can deploy your Lambda function code as a [container image \(p. 881\)](#). AWS provides the following resources to help you build a container image for your Ruby function:

- AWS base images for Lambda

These base images are preloaded with a language runtime and other components that are required to run the image on Lambda. AWS provides a Dockerfile for each of the base images to help with building your container image.

- Open-source runtime interface clients

If you use a community or private enterprise base image, you must add a [runtime interface client \(p. 883\)](#) to the base image to make it compatible with Lambda.

- Open-source runtime interface emulator

Lambda provides a runtime interface emulator (RIE) for you to test your function locally. The base images for Lambda and base images for custom .runtimes include the RIE. For other base images, you can download the RIE for [testing your image \(p. 884\)](#) locally.

The workflow for a function defined as a container image includes these steps:

1. Build your container image using the resources listed in this topic.
2. Upload the image to your [Amazon Elastic Container Registry \(Amazon ECR\) container registry \(p. 891\)](#).
3. [Create the function \(p. 113\)](#) or [update the function code \(p. 115\)](#) to deploy the image to an existing function.

Topics

- [AWS base images for Ruby \(p. 368\)](#)
- [Using a Ruby base image \(p. 369\)](#)
- [Ruby runtime interface clients \(p. 369\)](#)
- [Create a Ruby image from an AWS base image \(p. 369\)](#)
- [Deploy the container image \(p. 370\)](#)

AWS base images for Ruby

AWS provides the following base images for Ruby:

Tags	Runtime	Operating system	Dockerfile	Deprecation
2.7	Ruby 2.7*	Amazon Linux 2	Dockerfile for Ruby 2.7* on GitHub	Nov 15, 2023

Amazon ECR repository: [gallery.ecr.aws/lambda/ruby](#)

Using a Ruby base image

For instructions on how to use a Ruby base image, choose the **usage** tab on [AWS Lambda base images for Ruby](#) in the *Amazon ECR repository*.

Ruby runtime interface clients

Install the runtime interface client for Ruby using the RubyGems.org package manager:

```
gem install aws_lambda_ric
```

For package details, see [Lambda RIC on RubyGems.org](#).

You can also download the [Ruby runtime interface client](#) from GitHub.

Create a Ruby image from an AWS base image

When you build a container image for Ruby using an AWS base image, you only need to copy the ruby app to the container and install any dependencies.

To build and deploy a Ruby function with the `xruby:2.7` base image.

1. On your local machine, create a project directory for your new function.
2. In your project directory, add a file named `app.rb` containing your function code. The following example shows a simple Ruby handler.

```
module LambdaFunction
  class Handler
    def self.process(event:, context:)
      "Hello from Ruby 2.7 container image!"
    end
  end
end
```

3. Use a text editor to create a Dockerfile in your project directory. The following example shows the Dockerfile for the handler that you created in the previous step. Install any dependencies under the `#{LAMBDA_TASK_ROOT}` directory alongside the function handler to ensure that the Lambda runtime can locate them when the function is invoked.

```
FROM public.ecr.aws/lambda/ruby:2.7

# Copy function code
COPY app.rb ${LAMBDA_TASK_ROOT}

# Copy dependency management file
COPY Gemfile ${LAMBDA_TASK_ROOT}

# Install dependencies under LAMBDA_TASK_ROOT
ENV GEM_HOME=${LAMBDA_TASK_ROOT}
RUN bundle install

# Set the CMD to your handler (could also be done as a parameter override outside of
# the Dockerfile)
CMD [ "app.LambdaFunction::Handler.process" ]
```

4. To create the container image, follow steps 4 through 7 in [Create an image from an AWS base image for Lambda \(p. 888\)](#).

Deploy the container image

For a new function, you deploy the Ruby image when you [create the function \(p. 113\)](#). For an existing function, if you rebuild the container image, you need to redeploy the image by [updating the function code \(p. 115\)](#).

AWS Lambda context object in Ruby

When Lambda runs your function, it passes a context object to the [handler \(p. 364\)](#). This object provides methods and properties that provide information about the invocation, function, and execution environment.

Context methods

- `get_remaining_time_in_millis` – Returns the number of milliseconds left before the execution times out.

Context properties

- `function_name` – The name of the Lambda function.
- `function_version` – The [version \(p. 83\)](#) of the function.
- `invoked_function_arn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memory_limit_in_mb` – The amount of memory that's allocated for the function.
- `aws_request_id` – The identifier of the invocation request.
- `log_group_name` – The log group for the function.
- `log_stream_name` – The log stream for the function instance.
- `deadline_ms` – The date that the execution times out, in Unix time milliseconds.
- `identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
- `client_context` – (mobile apps) Client context that's provided to Lambda by the client application.

AWS Lambda function logging in Ruby

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

This page describes how to produce log output from your Lambda function's code, or access logs using the AWS Command Line Interface, the Lambda console, or the CloudWatch console.

Sections

- [Creating a function that returns logs \(p. 372\)](#)
- [Using the Lambda console \(p. 373\)](#)
- [Using the CloudWatch console \(p. 373\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 373\)](#)
- [Deleting logs \(p. 375\)](#)
- [Logger library \(p. 376\)](#)

Creating a function that returns logs

To output logs from your function code, you can use `puts` statements, or any logging library that writes to `stdout` or `stderr`. The following example logs the values of environment variables and the event object.

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 log format

```
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]3893xmp17fac4485b47bb75b671a283c',
         '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: 16 ms Memory Size: 128 MB Max Memory Used: 56 MB Init Duration: 130.49 ms
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1 SegmentId: 07f5xmpl2d1f6f85 Sampled:
true
```

The Ruby runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details.

Report Log

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TracId** – For traced requests, the [AWS X-Ray trace ID \(p. 807\)](#).
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

For more detailed logs, use the [the section called "Logger library" \(p. 376\)](#).

Using the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

Using the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the [Log groups page](#) on the CloudWatch console.
2. Choose the log group for your function (`/aws/lambda/your-function-name`).
3. Choose a log stream.

Each log stream corresponds to an [instance of your function \(p. 14\)](#). A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see [Error processor sample application for AWS Lambda \(p. 1015\)](#).

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

You can use the [AWS CLI](#) to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the LogResult field for a function named my-function.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{  
    "StatusCode": 200,  
    "LogResult":  
        "U1RBULQgUmVxdWVzdElk0iA4N2QwNDRi0C1mMTU0LTExZTgt0GNkYS0y0Tc0YzV1NGZiMjEgVmVyc21vb...",  
    "ExecutedVersion": "$LATEST"  
}
```

Example decode the logs

In the same command prompt, use the base64 utility to decode the logs. The following example shows how to retrieve base64-encoded logs for my-function.

```
aws lambda invoke --function-name my-function out --log-type Tail \  
--query 'LogResult' --output text | base64 -d
```

You should see the following output:

```
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
```

The base64 utility is available on Linux, macOS, and [Ubuntu on Windows](#). macOS users may need to use base64 -D.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses sed to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the get-log-events command.

Copy the contents of the following code sample and save in your Lambda project directory as get-logs.sh.

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run aws configure set cli-binary-format raw-in-base64-out. For more information, see [AWS CLI supported global command line options](#).

```
#!/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 and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
  "StatusCode": 200,
  "ExecutedVersion": "$LATEST"
}
{
  "events": [
    {
      "timestamp": 1559763003171,
      "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST\n",
      "ingestionTime": 1559763003309
    },
    {
      "timestamp": 1559763003173,
      "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf\tINFO\tENVIRONMENT VARIABLES\r{\r\t\t\"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",\\r ...",
      "ingestionTime": 1559763018353
    },
    {
      "timestamp": 1559763003173,
      "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf\tINFO\tEVENT\t{\r\t\t\"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",
      "ingestionTime": 1559763018353
    }
  ],
  "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
  "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.

Logger library

The Ruby [logger library](#) returns streamlined logs that are easily read. Use the logger utility to output detailed information, messages, and errors codes related to your function.

```
# 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
```

The output from logger includes the log level, timestamp, and request 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: 3 ms Memory Size: 128 MB Max Memory Used: 56 MB Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370 SegmentId: 073cxmpl3e442861 Sampled:
true
```

AWS Lambda function errors in Ruby

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

This page describes how to view Lambda function invocation errors for the Ruby runtime using the Lambda console and the AWS CLI.

Sections

- [Syntax \(p. 377\)](#)
- [How it works \(p. 377\)](#)
- [Using the Lambda console \(p. 378\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 378\)](#)
- [Error handling in other AWS services \(p. 379\)](#)
- [Sample applications \(p. 380\)](#)
- [What's next? \(p. 380\)](#)

Syntax

Example function.rb

```
def handler(event:, context:)
    puts "Processing event..."
    [1, 2, 3].first("two")
    "Success"
end
```

This code results in a type error. Lambda catches the error and generates a JSON document with fields for the error message, the type, and the stack trace.

```
{
  "errorMessage": "no implicit conversion of String into Integer",
  "errorType": "Function<TypeError>",
  "stackTrace": [
    "/var/task/function.rb:3:in `first'",
    "/var/task/function.rb:3:in `handler'"
  ]
}
```

How it works

When you invoke a Lambda function, Lambda receives the invocation request and validates the permissions in your execution role, verifies that the event document is a valid JSON document, and checks parameter values.

If the request passes validation, Lambda sends the request to a function instance. The [Lambda runtime \(p. 37\)](#) environment converts the event document into an object, and passes it to your function handler.

If Lambda encounters an error, it returns an exception type, message, and HTTP status code that indicates the cause of the error. The client or service that invoked the Lambda function can handle the error programmatically, or pass it along to an end user. The correct error handling behavior depends on the type of application, the audience, and the source of the error.

The following list describes the range of status codes you can receive from Lambda.

2xx

A 2xx series error with a X-Amz-Function-Error header in the response indicates a Lambda runtime or function error. A 2xx series status code indicates that Lambda accepted the request, but instead of an error code, Lambda indicates the error by including the X-Amz-Function-Error header in the response.

4xx

A 4xx series error indicates an error that the invoking client or service can fix by modifying the request, requesting permission, or by retrying the request. 4xx series errors other than 429 generally indicate an error with the request.

5xx

A 5xx series error indicates an issue with Lambda, or an issue with the function's configuration or resources. 5xx series errors can indicate a temporary condition that can be resolved without any action by the user. These issues can't be addressed by the invoking client or service, but a Lambda function's owner may be able to fix the issue.

For a complete list of invocation errors, see [InvokeFunction errors \(p. 1262\)](#).

Using the Lambda console

You can invoke your function on the Lambda console by configuring a test event and viewing the output. The output is captured in the function's execution logs and, when [active tracing \(p. 807\)](#) is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to test, and choose **Test**.
3. Under **Test event**, select **New event**.
4. Select a **Template**.
5. For **Name**, enter a name for the test. In the text entry box, enter the JSON test event.
6. Choose **Save changes**.
7. Choose **Test**.

The Lambda console invokes your function [synchronously \(p. 120\)](#) and displays the result. To see the response, logs, and other information, expand the **Details** section.

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

When you invoke a Lambda function in the AWS CLI, the AWS CLI splits the response into two documents. The AWS CLI response is displayed in your command prompt. If an error has occurred, the response contains a `FunctionError` field. The invocation response or error returned by the function is written to an output file. For example, `output.json` or `output.txt`.

The following [invoke](#) command example demonstrates how to invoke a function and write the invocation response to an output.txt file.

```
aws lambda invoke \
--function-name my-function \
--cli-binary-format raw-in-base64-out \
--payload '{"key1": "value1", "key2": "value2", "key3": "value3"}' output.txt
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

You should see the AWS CLI response in your command prompt:

```
{  
    "StatusCode": 200,  
    "FunctionError": "Unhandled",  
    "ExecutedVersion": "$LATEST"  
}
```

You should see the function invocation response in the `output.txt` file. In the same command prompt, you can also view the output in your command prompt using:

```
cat output.txt
```

You should see the invocation response in your command prompt.

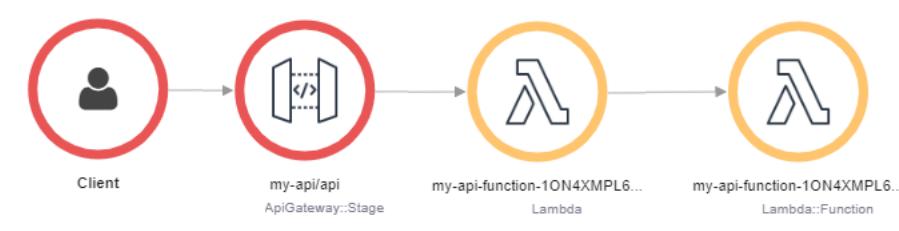
```
{"errorMessage": "no implicit conversion of String into  
Integer", "errorType": "Function<TypeError>", "stackTrace": ["/var/task/function.rb:3:in  
'first'", "/var/task/function.rb:3:in 'handler'"]}
```

Error handling in other AWS services

When another AWS service invokes your function, the service chooses the invocation type and retry behavior. AWS services can invoke your function on a schedule, in response to a lifecycle event on a resource, or to serve a request from a user. Some services invoke functions asynchronously and let Lambda handle errors, while others retry or pass errors back to the user.

For example, API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502 error code. To customize the error response, you must catch errors in your code and format a response in the required format.

We recommend using AWS X-Ray to determine the source of an error and its cause. X-Ray allows you to find out which component encountered an error, and see details about the errors. The following example shows a function error that resulted in a 502 response from API Gateway.



For more information, see [Instrumenting Ruby code in AWS Lambda \(p. 381\)](#).

Sample applications

The following sample code is available for the Ruby runtime.

Sample Lambda applications in Ruby

- [blank-ruby](#) – A Ruby function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.
- [Ruby Code Samples for AWS Lambda](#) – Code samples written in Ruby that demonstrate how to interact with AWS Lambda.

What's next?

- Learn how to show logging events for your Lambda function on the [the section called "Logging" \(p. 372\)](#) page.

Instrumenting Ruby code in AWS Lambda

Lambda integrates with AWS X-Ray to enable you to trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, from the frontend API to storage and database on the backend. By simply adding the X-Ray SDK library to your build configuration, you can record errors and latency for any call that your function makes to an AWS service.

After you've configured active tracing, you can observe specific requests through your application. The [X-Ray service graph](#) shows information about your application and all its components. The following example from the [error processor \(p. 1015\)](#) sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.



To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring and operations tools**.
4. Choose **Edit**.
5. Under **X-Ray**, toggle on **Active tracing**.
6. Choose **Save**.

Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see [AWS X-Ray pricing](#).

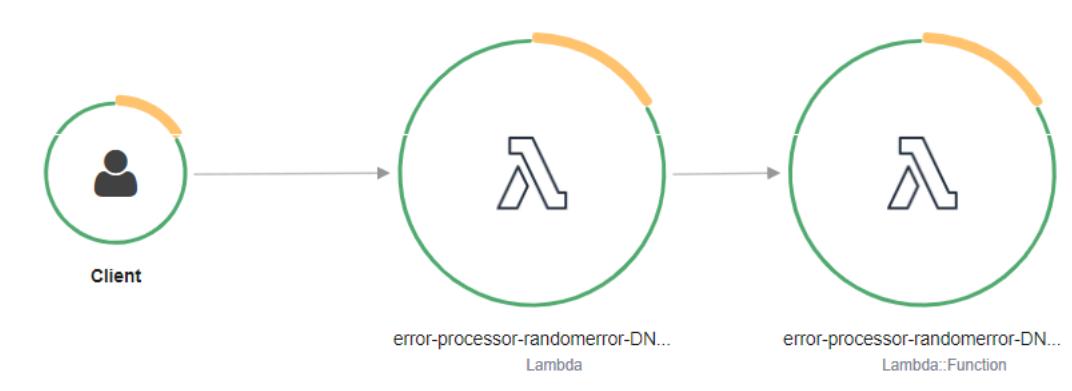
Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's [execution role \(p. 816\)](#). Otherwise, add the [AWSXRayDaemonWriteAccess](#) policy to the execution role.

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

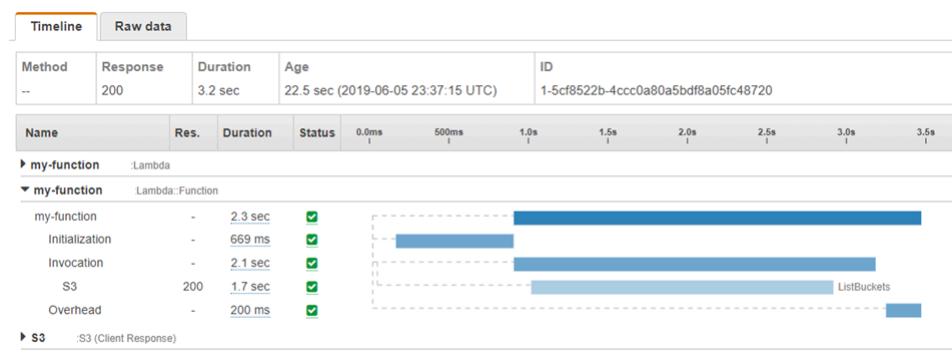
Note

You cannot configure the X-Ray sampling rate for your functions.

When using active tracing, Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes for the primary function from the [error processor sample application \(p. 1015\)](#).



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of AWS::Lambda and the other has origin AWS::Function.



This example expands the function segment to show its three subsegments:

- **Initialization** – Represents time spent loading your function and running [initialization code \(p. 13\)](#). This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

You can instrument your handler code to record metadata and trace downstream calls. To record detail about calls that your handler makes to other resources and services, use the X-Ray SDK for Ruby. To get the SDK, add the `aws-xray-sdk` package to your application's dependencies.

Example [blank-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'
  
```

To instrument AWS SDK clients, require the `aws-xray-sdk/lambda` module after creating a client in initialization code.

Example [blank-ruby/function/lambda_function.rb](#) – Tracing an AWS SDK client

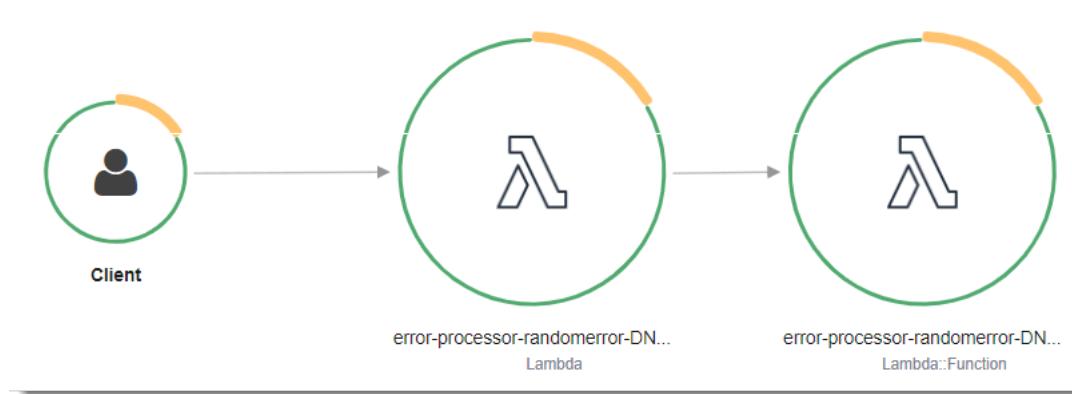
```
# lambda_function.rb
require 'logger'
require 'json'
require 'aws-sdk-lambda'
$client = Aws::Lambda::Client.new()
$client.get_account_settings()

require 'aws-xray-sdk/lambda'

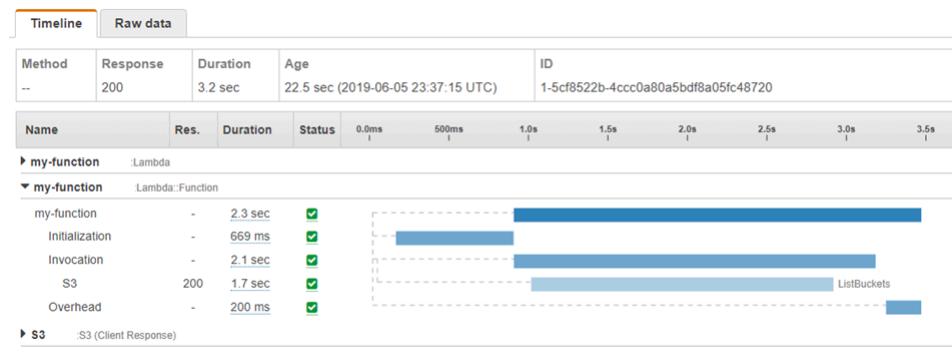
def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  ...

```

When using active tracing, Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes for the primary function from the [error processor sample application \(p. 1015\)](#).



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of `AWS::Lambda` and the other has origin `AWS::Lambda::Function`.



This example expands the function segment to show its three subsegments:

- **Initialization** – Represents time spent loading your function and running [initialization code \(p. 13\)](#). This subsegment only appears for the first event that each instance of your function processes.

- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see [The X-Ray SDK for Ruby](#) in the AWS X-Ray Developer Guide.

Sections

- [Enabling active tracing with the Lambda API \(p. 384\)](#)
- [Enabling active tracing with AWS CloudFormation \(p. 384\)](#)
- [Storing runtime dependencies in a layer \(p. 385\)](#)

Enabling active tracing with the Lambda API

To manage tracing configuration with the AWS CLI or AWS SDK, use the following API operations:

- [UpdateFunctionConfiguration \(p. 1377\)](#)
- [GetFunctionConfiguration \(p. 1229\)](#)
- [CreateFunction \(p. 1165\)](#)

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Enabling active tracing with AWS CloudFormation

To activate tracing on an AWS::Lambda::Function resource in an AWS CloudFormation template, use the TracingConfig property.

Example [function-inline.yml](#) – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...

```

For an AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function resource, use the Tracing property.

Example [template.yml](#) – Tracing configuration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active

```

...

Storing runtime dependencies in a layer

If you use the X-Ray SDK to instrument AWS SDK clients your function code, your deployment package can become quite large. To avoid uploading runtime dependencies every time you update your function code, package the X-Ray SDK in a [Lambda layer \(p. 93\)](#).

The following example shows an `AWS::Serverless::LayerVersion` resource that stores X-Ray SDK for Ruby.

Example `template.yml` – Dependencies layer

```
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
```

With this configuration, you update the library layer only if you change your runtime dependencies. Since the function deployment package contains only your code, this can help reduce upload times.

Creating a layer for dependencies requires build changes to generate the layer archive prior to deployment. For a working example, see the [blank-ruby](#) sample application.

Building Lambda functions with Java

You can run Java code in AWS Lambda. Lambda provides [runtimes \(p. 37\)](#) for Java that run your code to process events. Your code runs in an Amazon Linux environment that includes AWS credentials from an AWS Identity and Access Management (IAM) role that you manage.

Lambda supports the following Java runtimes.

Java

Name	Identifier	Operating system	Architectures	Deprecation (Phase 1)
Java 17	java17	Amazon Linux 2	x86_64, arm64	
Java 11	java11	Amazon Linux 2	x86_64, arm64	
Java 8	java8.al2	Amazon Linux 2	x86_64, arm64	
Java 8	java8	Amazon Linux	x86_64	

Lambda provides the following libraries for Java functions:

- [com.amazonaws:aws-lambda-java-core](#) (required) – Defines handler method interfaces and the context object that the runtime passes to the handler. If you define your own input types, this is the only library that you need.
- [com.amazonaws:aws-lambda-java-events](#) – Input types for events from services that invoke Lambda functions.
- [com.amazonaws:aws-lambda-java-log4j2](#) – An appender library for Apache Log4j 2 that you can use to add the request ID for the current invocation to your [function logs \(p. 410\)](#).
- [AWS SDK for Java 2.0](#) – The official AWS SDK for the Java programming language.

Lambda functions use an [execution role \(p. 816\)](#) to get permission to write logs to Amazon CloudWatch Logs, and to access other services and resources. If you don't already have an execution role for function development, create one.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity** – Lambda.
 - **Permissions** – **AWSLambdaBasicExecutionRole**.
 - **Role name** – **lambda-role**.

The **AWSLambdaBasicExecutionRole** policy has the permissions that the function needs to write logs to CloudWatch Logs.

You can add permissions to the role later, or swap it out for a different role that's specific to a single function.

To create a Java function

1. Open the [Lambda console](#).
2. Choose **Create function**.
3. Configure the following settings:
 - **Name** – **my-function**.
 - **Runtime** – **Java 17**.
 - **Role** – **Choose an existing role**.
 - **Existing role** – **lambda-role**.
4. Choose **Create function**.
5. To configure a test event, choose **Test**.
6. For **Event name**, enter **test**.
7. Choose **Save changes**.
8. To invoke the function, choose **Test**.

The console creates a Lambda function with a handler class named Hello. Since Java is a compiled language, you can't view or edit the source code in the Lambda console, but you can modify its configuration, invoke it, and configure triggers.

Note

To get started with application development in your local environment, deploy one of the [sample applications \(p. 452\)](#) available in this guide's GitHub repository.

The Hello class has a function named handleRequest that takes an event object and a context object. This is the [handler function \(p. 389\)](#) that Lambda calls when the function is invoked. The Java function runtime gets invocation events from Lambda and passes them to the handler. In the function configuration, the handler value is example.Hello::handleRequest.

To update the function's code, you create a deployment package, which is a .zip file archive that contains your function code. As your function development progresses, you will want to store your function code in source control, add libraries, and automate deployments. Start by [creating a deployment package \(p. 393\)](#) and updating your code at the command line.

The function runtime passes a context object to the handler, in addition to the invocation event. The [context object \(p. 407\)](#) contains additional information about the invocation, the function, and the execution environment. More information is available from environment variables.

Your Lambda function comes with a CloudWatch Logs log group. The function runtime sends details about each invocation to CloudWatch Logs. It relays any [logs that your function outputs \(p. 410\)](#) during invocation. If your function [returns an error \(p. 429\)](#), Lambda formats the error and returns it to the invoker.

Topics

- [AWS Lambda function handler in Java \(p. 389\)](#)
- [Deploy Java Lambda functions with .zip or JAR file archives \(p. 393\)](#)
- [Deploy Java Lambda functions with container images \(p. 400\)](#)
- [AWS Lambda context object in Java \(p. 407\)](#)
- [AWS Lambda function logging in Java \(p. 410\)](#)
- [AWS Lambda function errors in Java \(p. 429\)](#)
- [Instrumenting Java code in AWS Lambda \(p. 434\)](#)
- [Creating a deployment package using Eclipse \(p. 449\)](#)
- [Java sample applications for AWS Lambda \(p. 452\)](#)

AWS Lambda function handler in Java

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. When the handler exits or returns a response, it becomes available to handle another event.

In the following example, a class named Handler defines a handler method named handleRequest. The handler method takes an event and context object as input and returns a string.

Example [Handler.java](#)

```
package example;
import com.amazonaws.services.lambda.runtime.Context
import com.amazonaws.services.lambda.runtime.RequestHandler
import com.amazonaws.services.lambda.runtime.LambdaLogger
...

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String, String>, String>{
    Gson gson = new GsonBuilder().setPrettyPrinting().create();
    @Override
    public String handleRequest(Map<String, String> event, Context context)
    {
        LambdaLogger logger = context.getLogger();
        String response = new String("200 OK");
        // 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));
        logger.log("EVENT TYPE: " + event.getClass().toString());
        return response;
    }
}
```

The [Lambda runtime \(p. 37\)](#) receives an event as a JSON-formatted string and converts it into an object. It passes the event object to your function handler along with a context object that provides details about the invocation and the function. You tell the runtime which method to invoke by setting the handler parameter on your function's configuration.

Handler formats

- [`package.Class::method`](#) – Full format. For example: `example.Handler::handleRequest`.
- [`package.Class`](#) – Abbreviated format for functions that implement a [handler interface \(p. 391\)](#). For example: `example.Handler`.

You can add [initialization code](#) outside of your handler method to reuse resources across multiple invocations. When the runtime loads your handler, it runs static code and the class constructor. Resources that are created during initialization stay in memory between invocations, and can be reused by the handler thousands of times.

In the following example, the logger and the serializer are created when the function serves its first event. Subsequent events served by the same function instance are much faster because those resources already exist.

Example [HandlerS3.java](#) – Initialization code

```
public class HandlerS3 implements RequestHandler<S3Event, String>{
    private static final Logger logger = LoggerFactory.getLogger(HandlerS3.class);
```

```
Gson gson = new GsonBuilder().setPrettyPrinting().create();
@Override
public String handleRequest(S3Event event, Context context)
{
    ...
}
```

The GitHub repo for this guide provides easy-to-deploy sample applications that demonstrate a variety of handler types. For details, see the [end of this topic \(p. 392\)](#).

Sections

- [Choosing input and output types \(p. 390\)](#)
- [Handler interfaces \(p. 391\)](#)
- [Sample handler code \(p. 392\)](#)

Choosing input and output types

You specify the type of object that the event maps to in the handler method's signature. In the preceding example, the Java runtime deserializes the event into a type that implements the `Map<String, String>` interface. String-to-string maps work for flat events like the following:

Example [Event.json](#) – Weather data

```
{
    "temperatureK": 281,
    "windKmh": -3,
    "humidityPct": 0.55,
    "pressureHPa": 1020
}
```

However, the value of each field must be a string or number. If the event includes a field that has an object as a value, the runtime can't deserialize it and returns an error.

Choose an input type that works with the event data that your function processes. You can use a basic type, a generic type, or a well-defined type.

Input types

- `Integer`, `Long`, `Double`, etc. – The event is a number with no additional formatting—for example, `3.5`. The runtime converts the value into an object of the specified type.
- `String` – The event is a JSON string, including quotes—for example, `"My string."`. The runtime converts the value (without quotes) into a `String` object.
- `Type`, `Map<String, Type>` etc. – The event is a JSON object. The runtime deserializes it into an object of the specified type or interface.
- `List<Integer>, List<String>, List<Object>`, etc. – The event is a JSON array. The runtime deserializes it into an object of the specified type or interface.
- `InputStream` – The event is any JSON type. The runtime passes a byte stream of the document to the handler without modification. You deserialize the input and write output to an output stream.
- Library type – For events sent by AWS services, use the types in the [aws-lambda-java-events \(p. 393\)](#) library.

If you define your own input type, it should be a deserializable, mutable plain old Java object (POJO), with a default constructor and properties for each field in the event. Keys in the event that don't map to a property as well as properties that aren't included in the event are dropped without error.

The output type can be an object or void. The runtime serializes return values into text. If the output is an object with fields, the runtime serializes it into a JSON document. If it's a type that wraps a primitive value, the runtime returns a text representation of that value.

Handler interfaces

The [aws-lambda-java-core](#) library defines two interfaces for handler methods. Use the provided interfaces to simplify handler configuration and validate the handler method signature at compile time.

- [com.amazonaws.services.lambda.runtime.RequestHandler](#)
- [com.amazonaws.services.lambda.runtime.RequestStreamHandler](#)

The RequestHandler interface is a generic type that takes two parameters: the input type and the output type. Both types must be objects. When you use this interface, the Java runtime deserializes the event into an object with the input type, and serializes the output into text. Use this interface when the built-in serialization works with your input and output types.

Example [Handler.java](#) – Handler interface

```
// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String, String>, String>{
    @Override
    public String handleRequest(Map<String, String> event, Context context)
```

To use your own serialization, implement the RequestStreamHandler interface. With this interface, Lambda passes your handler an input stream and output stream. The handler reads bytes from the input stream, writes to the output stream, and returns void.

The following example uses buffered reader and writer types to work with the input and output streams. It uses the [Gson](#) library for serialization and deserialization.

Example [HandlerStream.java](#)

```
import com.amazonaws.services.lambda.runtime.Context
import com.amazonaws.services.lambda.runtime.RequestStreamHandler
import com.amazonaws.services.lambda.runtime.LambdaLogger
...
// Handler value: example.HandlerStream
public class HandlerStream implements RequestStreamHandler {
    Gson gson = new GsonBuilder().setPrettyPrinting().create();
    @Override
    public void handleRequest(InputStream inputStream, OutputStream outputStream, Context context) throws IOException
    {
        LambdaLogger logger = context.getLogger();
        BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream,
CharsetName("US-ASCII")));
        PrintWriter writer = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(outputStream, Charset.forName("US-ASCII"))));
        try
        {
            HashMap event = gson.fromJson(reader, HashMap.class);
            logger.log("STREAM TYPE: " + inputStream.getClass().toString());
            logger.log("EVENT TYPE: " + event.getClass().toString());
            writer.write(gson.toJson(event));
            if (writer.checkError())
            {
                logger.log("WARNING: Writer encountered an error.");
            }
        }
    }
}
```

```
        }
    }
    catch (IllegalStateException | JsonSyntaxException exception)
    {
        logger.log(exception.toString());
    }
    finally
    {
        reader.close();
        writer.close();
    }
}
```

Sample handler code

The GitHub repository for this guide includes sample applications that demonstrate the use of various handler types and interfaces. Each sample application includes scripts for easy deployment and cleanup, an AWS SAM template, and supporting resources.

Sample Lambda applications in Java

- [java-basic](#) – A collection of minimal Java functions with unit tests and variable logging configuration.
- [java-events](#) – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the [aws-lambda-java-events \(p. 393\)](#) library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- [s3-java](#) – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- [Use API Gateway to invoke a Lambda function](#) – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.

The java-events and s3-java applications take an AWS service event as input and return a string. The java-basic application includes several types of handlers:

- [Handler.java](#) – Takes a `Map<String, String>` as input.
- [HandlerInteger.java](#) – Takes an `Integer` as input.
- [HandlerList.java](#) – Takes a `List<Integer>` as input.
- [HandlerStream.java](#) – Takes an `InputStream` and `OutputStream` as input.
- [HandlerString.java](#) – Takes a `String` as input.
- [HandlerWeatherData.java](#) – Takes a custom type as input.

To test different handler types, just change the handler value in the AWS SAM template. For detailed instructions, see the sample application's `readme` file.

Deploy Java Lambda functions with .zip or JAR file archives

Your AWS Lambda function's code consists of scripts or compiled programs and their dependencies. You use a *deployment package* to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives.

This page describes how to create your deployment package as a .zip file or Jar file, and then use the deployment package to deploy your function code to AWS Lambda using the AWS Command Line Interface (AWS CLI).

Sections

- [Prerequisites \(p. 393\)](#)
- [Tools and libraries \(p. 393\)](#)
- [Building a deployment package with Gradle \(p. 394\)](#)
- [Building a deployment package with Maven \(p. 395\)](#)
- [Uploading a deployment package with the Lambda console \(p. 396\)](#)
- [Uploading a deployment package with the Lambda API \(p. 397\)](#)
- [Uploading a deployment package with AWS SAM \(p. 398\)](#)

Prerequisites

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

Tools and libraries

Lambda provides the following libraries for Java functions:

- [com.amazonaws:aws-lambda-java-core](#) (required) – Defines handler method interfaces and the context object that the runtime passes to the handler. If you define your own input types, this is the only library that you need.
- [com.amazonaws:aws-lambda-java-events](#) – Input types for events from services that invoke Lambda functions.
- [com.amazonaws:aws-lambda-java-log4j2](#) – An appender library for Apache Log4j 2 that you can use to add the request ID for the current invocation to your [function logs \(p. 410\)](#).
- [AWS SDK for Java 2.0](#) – The official AWS SDK for the Java programming language.

These libraries are available through [Maven Central Repository](#). Add them to your build definition as follows:

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'
```

```
}
```

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>
</dependencies>
```

To create a deployment package, compile your function code and dependencies into a single .zip file or Java Archive (JAR) file. For Gradle, [use the Zip build type \(p. 394\)](#). For Apache Maven, [use the Maven Shade plugin \(p. 395\)](#). To upload your deployment package, use the Lambda console, the Lambda API, or AWS Serverless Application Model (AWS SAM).

Note

To keep your deployment package size small, package your function's dependencies in layers. Layers enable you to manage your dependencies independently, can be used by multiple functions, and can be shared with other accounts. For more information, see [Creating and sharing Lambda layers \(p. 93\)](#).

Building a deployment package with Gradle

To create a deployment package with your function's code and dependencies in Gradle, use the Zip build type. Here's an example from a [complete sample build.gradle file](#):

Example build.gradle – Build task

```
task buildZip(type: Zip) {
  from compileJava
  from processResources
  into('lib') {
    from configurations.runtimeClasspath
  }
}
```

This build configuration produces a deployment package in the build/distributions directory. The compileJava task compiles your function's classes. The processResources task copies the Java project resources into their target directory, potentially processing them. The statement into('lib') then copies dependency libraries from the build's classpath into a folder named lib.

Example build.gradle – Dependencies

```
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 loads JAR files in Unicode alphabetical order. If multiple JAR files in the lib directory contain the same class, the first one is used. You can use the following shell script to identify duplicate classes:

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
```

Building a deployment package with Maven

To build a deployment package with Maven, use the [Maven Shade plugin](#). The plugin creates a JAR file that contains the compiled function code and all of its dependencies.

Example pom.xml – Plugin configuration

```
<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>
```

To build the deployment package, use the mvn package command.

```
[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] -----
```

This command generates a JAR file in the target directory.

Note

If you're working with a [multi-release JAR \(MRJAR\)](#), you must include the MRJAR (i.e. the shaded JAR produced by the Maven Shade plugin) in the lib directory and zip it before uploading your deployment package to Lambda. Otherwise, Lambda may not properly unpack your JAR file, causing your MANIFEST.MF file to be ignored.

If you use the appender library (aws-lambda-java-log4j2), you must also configure a transformer for the Maven Shade plugin. The transformer library combines versions of a cache file that appear in both the appender library and in Log4j.

Example pom.xml – Plugin configuration with 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.PluginsCacheFileTransformer">  
                        </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>
```

Uploading a deployment package with the Lambda console

You can upload a deployment package to any existing function using the Lambda console.

To upload a deployment package with the Lambda console

1. Open the [Functions page](#) of the Lambda console.

2. Choose a function.
3. Under **Code source**, choose **Upload from**.
4. Upload the deployment package.
5. Choose **Save**.

Uploading a deployment package with the Lambda API

To update a function's code with the AWS Command Line Interface (AWS CLI) or AWS SDK, use the [UpdateFunctionCode \(p. 1367\)](#) API operation. For the AWS CLI, use the `update-function-code` command. The following command uploads a deployment package named `my-function.zip` in the current directory:

```
aws lambda update-function-code --function-name my-function --zip-file fileb://my-function.zip
```

You should see the following output:

```
{  
    "FunctionName": "my-function",  
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",  
    "Runtime": "java8",  
    "Role": "arn:aws:iam::123456789012:role/lambda-role",  
    "Handler": "example.Handler",  
    "CodeSha256": "Qf0hMc1I2di6YFMi9aXm3JtGTmcDbjniEuiYonYptAk=",  
    "Version": "$LATEST",  
    "TracingConfig": {  
        "Mode": "Active"  
    },  
    "RevisionId": "983ed1e3-ca8e-434b-8dc1-7d72ebadd83d",  
    ...  
}
```

If your deployment package is larger than 50 MB, you can't upload it directly. Upload it to an Amazon Simple Storage Service (Amazon S3) bucket and point Lambda to the object. The following example commands upload a deployment package to an S3 bucket named `my-bucket` and use it to update a function's code:

```
aws s3 cp my-function.zip s3://my-bucket
```

You should see the following output:

```
upload: my-function.zip to s3://my-bucket/my-function
```

```
aws lambda update-function-code --function-name my-function \  
    --s3-bucket my-bucket --s3-key my-function.zip
```

You should see the following output:

```
{  
    "FunctionName": "my-function",  
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",  
    "Runtime": "java8",  
    ...  
}
```

```
"Role": "arn:aws:iam::123456789012:role/lambda-role",
"Handler": "example.Handler",
"CodeSha256": "Qf0hMc1I2di6YFMi9aXm3JtGTmcDbjniEuiYonYptAk=",
"Version": "$LATEST",
"TracingConfig": {
    "Mode": "Active"
},
"RevisionId": "983ed1e3-ca8e-434b-8dc1-7d72ebadd83d",
...
}
```

You can use this method to upload function packages up to 250 MB (decompressed).

Uploading a deployment package with AWS SAM

You can use AWS SAM to automate deployments of your function code, configuration, and dependencies. AWS SAM is an extension of AWS CloudFormation that provides a simplified syntax for defining serverless applications. The following example template defines a function with a deployment package in the build/distributions directory that Gradle uses:

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: java8
      Description: Java function
      MemorySize: 512
      Timeout: 10
      # Function's execution role
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
        - AWSLambdaVPCAccessExecutionRole
      Tracing: Active
```

To create the function, use the package and deploy commands. These commands are customizations to the AWS CLI. They wrap other commands to upload the deployment package to Amazon S3, rewrite the template with the object URL, and update the function's code.

The following example script runs a Gradle build and uploads the deployment package that it creates. It creates an AWS CloudFormation stack the first time you run it. If the stack already exists, the script updates it.

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
```

For a complete working example, see the following sample applications:

Sample Lambda applications in Java

- [java-basic](#) – A collection of minimal Java functions with unit tests and variable logging configuration.
- [java-events](#) – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the [aws-lambda-java-events \(p. 393\)](#) library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- [s3-java](#) – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- [Use API Gateway to invoke a Lambda function](#) – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.

Deploy Java Lambda functions with container images

You can deploy your Lambda function code as a [container image \(p. 881\)](#). To help you build a container image for your Java function, AWS provides the following resources:

- AWS base images for Lambda

These base images are preloaded with a language runtime and other components that are required to run the image on Lambda. AWS provides a Dockerfile for each of the base images to help with building your container image.

- Open-source runtime interface clients

If you use a community or private enterprise base image, you must add a [runtime interface client \(p. 883\)](#) to the base image to make it compatible with Lambda.

- Open-source runtime interface emulator

Lambda provides a runtime interface emulator (RIE) for you to test your function locally. The base images for Lambda and base images for custom .runtimes include the RIE. For other base images, you can download the RIE for [testing your image \(p. 884\)](#) locally.

The workflow for a function defined as a container image includes these steps:

1. Build your container image using the resources listed in this topic.
2. Upload the image to your [Amazon Elastic Container Registry \(Amazon ECR\) container registry \(p. 891\)](#).
3. [Create the function \(p. 113\)](#) or [update the function code \(p. 115\)](#) to deploy the image to an existing function.

Topics

- [AWS base images for Java \(p. 400\)](#)
- [Using a Java base image \(p. 401\)](#)
- [Java runtime interface clients \(p. 406\)](#)

AWS base images for Java

AWS provides the following base images for Java:

Tags	Runtime	Operating system	Dockerfile	Deprecation
17	Java 17	Amazon Linux 2	Dockerfile for Java 17 on GitHub	
11	Java 11	Amazon Linux 2	Dockerfile for Java 11 on GitHub	
8.al2	Java 8	Amazon Linux 2	Dockerfile for Java 8 on GitHub	

Tags	Runtime	Operating system	Dockerfile	Deprecation
8	Java 8	Amazon Linux	Dockerfile for Java 8 on GitHub	

Amazon ECR repository: [gallery.ecr.aws/lambda/java](#)

Using a Java base image

Prerequisites

To complete the steps in this section, you must have the following:

- Java (for example, [Amazon Corretto](#))
- [Docker](#)
- [Apache Maven](#) or [Gradle](#)
- [AWS Command Line Interface \(AWS CLI\) version 2](#)

Creating an image from a base image

Maven

1. Run the following command to create a Maven project using the [archetype for Lambda](#). The following parameters are required:
 - **service** – The AWS service client to use in the Lambda function. For a list of available sources, see [aws-sdk-java-v2/services](#) on GitHub.
 - **region** – The AWS Region where you want to create the Lambda function.
 - **groupId** – The full package namespace of your application.
 - **artifactId** – Your project name. This becomes the name of the directory for your project.

```
mvn -B archetype:generate \
-DarchetypeGroupId=software.amazon.awssdk \
-DarchetypeArtifactId=archetype-lambda -Dservice=s3 -Dregion=US_WEST_2 \
-DgroupId=com.example.myapp \
-DartifactId=myapp
```

The Maven archetype for Lambda is preconfigured to compile with Java SE 8 and includes a dependency to the AWS SDK for Java. If you create your project with a different archetype or by using another method, you must [configure the Java compiler for Maven](#) and [declare the SDK as a dependency](#).

2. Open the `/myapp/src/main/java/com/example/myapp` directory, and find the `App.java` file. This is the code for the Lambda function. You can use the provided sample code for testing, or replace it with your own.
3. Create a new Dockerfile with the following configuration:
 - Set the `FROM` property to the [URI of the base image](#).
 - Set the `CMD` argument to the Lambda function handler.

Example Dockerfile

```
FROM public.ecr.aws/lambda/java:11

# 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. Compile the project and collect the runtime dependencies.

```
mvn compile dependency:copy-dependencies -DincludeScope=runtime
```

5. Build the Docker image with the [docker build](#) command. The following example names the image docker-image and gives it the test [tag](#).

```
docker build -t docker-image:test .
```

(Optional) Test the image locally

1. Start the Docker image with the [docker run](#) command. In this example, docker-image is the image name and test is the tag.

```
docker run -p 9000:8080 docker-image:test
```

This command runs the image as a container and creates a local endpoint at localhost:9000/2015-03-31/functions/function/invocations.

2. Test your application locally using the [RIE \(p. 884\)](#). From a new terminal window, post an event to the following endpoint using a [curl](#) command:

```
curl -XPOST "http://localhost:9000/2015-03-31/functions/function/invocations" -d
'{}'
```

This command invokes the function running in the container image and returns a response.

3. Get the container ID.

```
docker ps
```

4. Use the [docker kill](#) command to stop the container. In this command, replace 3766c4ab331c with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Gradle

1. Create a directory for the project, and then switch to that directory.

```
mkdir example
```

```
cd example
```

2. Run the following command to have Gradle generate a new Java application project in the example directory in your environment. To accept the default options provided in the interactive experience, press Enter.

```
gradle init --type java-application
```

3. Open the `/example/app/src/main/java/example` directory, and find the `App.java` file. This is the code for the Lambda function. You can use the following sample code for testing, or replace it with your own.

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. Open the `build.gradle` file. If you're using the sample function code from the previous step, replace the contents of `build.gradle` with the following. If you're using your own function code, modify your `build.gradle` file as needed.

```
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. The `gradle init` command from step 2 also generated a dummy test case in the `app/test` directory. For the purposes of this tutorial, skip running tests by deleting the `/test` directory.
6. Build the project.

```
gradle build
```

7. In the project's root directory (`/example`), create a `Dockerfile` with the following configuration:
 - Set the `FROM` property to the [URI of the base image](#).
 - Use the `COPY` command to copy the function code and runtime dependencies to `{LAMBDA_TASK_ROOT}`.
 - Set the `CMD` argument to the Lambda function handler.

Example Dockerfile

```
FROM public.ecr.aws/lambda/java:11

# 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. Build the Docker image with the [docker build](#) command. The following example names the image docker-image and gives it the test [tag](#).

```
docker build -t docker-image:test .
```

(Optional) Test the image locally

1. Start the Docker image with the [docker run](#) command. In this example, docker-image is the image name and test is the tag.

```
docker run -p 9000:8080 docker-image:test
```

This command runs the image as a container and creates a local endpoint at localhost:9000/2015-03-31/functions/function/invocations.

2. Test your application locally using the [RIE \(p. 884\)](#). From a new terminal window, post an event to the following endpoint using a [curl](#) command:

```
curl -XPOST "http://localhost:9000/2015-03-31/functions/function/invocations" -d
'{}'
```

This command invokes the function running in the container image and returns a response.

3. Get the container ID.

```
docker ps
```

4. Use the [docker kill](#) command to stop the container. In this command, replace 3766c4ab331c with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the [get-login-password](#) command to authenticate the Docker CLI to your Amazon ECR registry.
 - Set the --region value to the AWS Region where you want to create the Amazon ECR repository.
 - Replace 111122223333 with your AWS account 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. Create a repository in Amazon ECR using the [create-repository](#) command.

```
aws ecr create-repository --repository-name hello-world --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

If successful, you see a response like this:

```
{  
    "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. Copy the `repositoryUri` from the output in the previous step.
4. Run the [docker tag](#) command to tag your local image into your Amazon ECR repository as the latest version. In this command:
 - Replace `docker-image:test` with the name and [tag](#) of your Docker image.
 - Replace the Amazon ECR repository URI with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the [docker push](#) command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.
6. [Create an execution role \(p. 191\)](#) for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.
7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
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
```

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{  
    "ExecutedVersion": "$LATEST",  
    "StatusCode": 200  
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must create a new image version and store the image in the Amazon ECR repository. For more information, see [Updating function code \(p. 115\)](#).

Java runtime interface clients

Install the runtime interface client for Java using the Apache Maven package manager. Add the following to your `pom.xml` file:

```
<dependency>  
    <groupId>com.amazonaws</groupId>  
    <artifactId>aws-lambda-java-runtime-interface-client</artifactId>  
    <version>2.3.1</version>  
</dependency>
```

For package details, see [AWS Lambda Java Runtime Interface Client](#) in Maven Central Repository.

You can also view the Java client source code in the [AWS Lambda Java Support Libraries](#) repository on GitHub.

AWS Lambda context object in Java

When Lambda runs your function, it passes a context object to the [handler \(p. 389\)](#). This object provides methods and properties that provide information about the invocation, function, and execution environment.

Context methods

- `getRemainingTimeInMillis()` – Returns the number of milliseconds left before the execution times out.
- `getFunctionName()` – Returns the name of the Lambda function.
- `getFunctionVersion()` – Returns the [version \(p. 83\)](#) of the function.
- `getInvokedFunctionArn()` – Returns the Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `getMemoryLimitInMB()` – Returns the amount of memory that's allocated for the function.
- `getAwsRequestId()` – Returns the identifier of the invocation request.
- `getLogGroupName()` – Returns the log group for the function.
- `getLogStreamName()` – Returns the log stream for the function instance.
- `getIdentity()` – (mobile apps) Returns information about the Amazon Cognito identity that authorized the request.
- `getClientContext()` – (mobile apps) Returns the client context that's provided to Lambda by the client application.
- `getLogger()` – Returns the [logger object \(p. 410\)](#) for the function.

The following example shows a function that uses the context object to access the Lambda logger.

Example [Handler.java](#)

```
package example;
import com.amazonaws.services.lambda.runtime.Context
import com.amazonaws.services.lambda.runtime.RequestHandler
import com.amazonaws.services.lambda.runtime.LambdaLogger
...

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String, String>, String>{
    Gson gson = new GsonBuilder().setPrettyPrinting().create();
    @Override
    public String handleRequest(Map<String, String> event, Context context)
    {
        LambdaLogger logger = context.getLogger();
        String response = new String("200 OK");
        // 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));
        logger.log("EVENT TYPE: " + event.getClass().toString());
        return response;
    }
}
```

The function serializes the context object into JSON and records it in its log stream.

Example log output

```
START RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Version: $LATEST
...
CONTEXT:
{
    "memoryLimit": 512,
    "awsRequestId": "6bc28136-xmpl-4365-b021-0ce6b2e64ab0",
    "functionName": "java-console",
    ...
}
...
END RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0
REPORT RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Duration: 198.50 ms Billed Duration:
200 ms Memory Size: 512 MB Max Memory Used: 90 MB Init Duration: 524.75 ms
```

The interface for the context object is available in the [aws-lambda-java-core](#) library. You can implement this interface to create a context class for testing. The following example shows a context class that returns dummy values for most properties and a working test logger.

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 LambdaLogger getLogger(){
        return new TestLogger();
    }
}
```

For more information on logging, see [AWS Lambda function logging in Java \(p. 410\)](#).

Context in sample applications

The GitHub repository for this guide includes sample applications that demonstrate the use of the context object. Each sample application includes scripts for easy deployment and cleanup, an AWS Serverless Application Model (AWS SAM) template, and supporting resources.

Sample Lambda applications in Java

- [java-basic](#) – A collection of minimal Java functions with unit tests and variable logging configuration.
- [java-events](#) – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the [aws-lambda-java-events \(p. 393\)](#) library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.

- [**s3-java**](#) – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- [**Use API Gateway to invoke a Lambda function**](#) – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.

All of the sample applications have a test context class for unit tests. The `java-basic` application shows you how to use the context object to get a logger. It uses SLF4J and Log4J 2 to provide a logger that works for local unit tests.

AWS Lambda function logging in Java

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

To output logs from your function code, you can use methods on [java.lang.System](#), or any logging module that writes to stdout or stderr. For simple use cases, this approach might be sufficient.

This page describes how to produce log output from your Lambda function's code, or access logs using the AWS Command Line Interface, the Lambda console, the CloudWatch console, or Infrastructure as code tools such as the AWS Serverless Application Model(AWS SAM).

Sections

- [Tools and libraries \(p. 410\)](#)
- [Creating a function that returns logs \(p. 410\)](#)
- [Using AWS Lambda Powertools for Java and AWS SAM for structured logging \(p. 412\)](#)
- [Using AWS Lambda Powertools for Java and the AWS CDK for structured logging \(p. 415\)](#)
- [Using the Lambda console \(p. 423\)](#)
- [Using the CloudWatch console \(p. 423\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 423\)](#)
- [Deleting logs \(p. 425\)](#)
- [Advanced logging with Log4j 2 and SLF4J \(p. 425\)](#)
- [Sample logging code \(p. 427\)](#)

Tools and libraries

[AWS Lambda Powertools for Java](#) is a developer toolkit to implement Serverless best practices and increase developer velocity. The [Logging utility](#) provides a Lambda optimized logger which includes additional information about function context across all your functions with output structured as JSON. Use this utility to do the following:

- Capture key fields from the Lambda context, cold start and structures logging output as JSON
- Log Lambda invocation events when instructed (disabled by default)
- Print all the logs only for a percentage of invocations via log sampling (disabled by default)
- Append additional keys to structured log at any point in time
- Use a custom log formatter (Bring Your Own Formatter) to output logs in a structure compatible with your organization's Logging RFC

Creating a function that returns logs

To output logs from your function code, you can use methods on [java.lang.System](#), or any logging module that writes to stdout or stderr. The [aws-lambda-java-core \(p. 393\)](#) library provides a logger class named `LambdaLogger` that you can access from the context object. The logger class supports multiline logs.

The following example uses the `LambdaLogger` logger provided by the context object.

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 log format

```
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: 200 ms Memory Size: 512 MB Max Memory Used: 90 MB Init Duration: 524.75 ms
```

The Java runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details:

Report Log

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.

- **Max Memory Used** – The amount of memory used by the function.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY Traceld** – For traced requests, the [AWS X-Ray trace ID \(p. 807\)](#).
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

Using AWS Lambda Powertools for Java and AWS SAM for structured logging

Follow the steps below to download, build, and deploy a sample Hello World Java application with integrated [AWS Lambda Powertools for Java](#) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Java 11
- [AWS CLI version 2](#)
- [AWS SAM CLI version 1.75 or later](#). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World Java template.

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type Zip  
--runtime java11 --no-tracing
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.

Note

For **HelloWorldFunction** may not have authorization defined, Is this okay?, make sure to enter y.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message": "hello world"}
```

7. To get the logs for the function, run [sam logs](#). For more information, see [Working with logs](#) in the [AWS Serverless Application Model Developer Guide](#).

```
sam logs --stack-name sam-app
```

The log output looks like this:

```
2023/02/03/[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
2023/02/03/[LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.114000 Picked up JAVA_TOOL_OPTIONS: -XX:+TieredCompilation -XX:TieredStopAtLevel=1
2023/02/03/[LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.793000
Transforming org/apache/logging/log4j/core/lookup/JndiLookup
(lambdainternal.CustomerClassLoader@1a6c5a9e)
2023/02/03/[LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:35.252000 START
RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765 Version: $LATEST
2023/02/03/[LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.531000 {
    "_aws": {
        "Timestamp": 1675416276051,
        "CloudWatchMetrics": [
            {
                "Namespace": "sam-app-powertools-java",
                "Metrics": [
                    {
                        "Name": "ColdStart",
                        "Unit": "Count"
                    }
                ],
                "Dimensions": [
                    [
                        "Service",
                        "FunctionName"
                    ]
                ]
            }
        ],
        "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
        "traceId":
"Root=1-63dc2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
        "FunctionName": "sam-app-HelloWorldFunction-y9Iu1FLJJBGD",
        "FunctionVersion": "$LATEST",
        "ColdStart": 1.0,
        "Service": "service_undefined",
        "logStreamId": "2023/02/03/[LATEST]851411a899b545eea2cffeba4cfbec81",
        "executionEnvironment": "AWS_Lambda_java11"
    }
}
2023/02/03/[LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.974000 Feb 03,
2023 9:24:36 AM com.amazonaws.xray.AWSXRayRecorder <init>
2023/02/03/[LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000 Feb 03,
2023 9:24:36 AM com.amazonaws.xray.config.DaemonConfiguration <init>
2023/02/03/[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.
```

```

2023/02/03[$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.cloudflare.net (CloudFront)","X-Amz-Cf-Id":"t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRCzD7T1Rdhbtiv1Q==","X-Amzn-Trace-Id":"Root=1-63dc2d1-25f90b9d1c753a783547f4dd","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.cloudflare.net (CloudFront)"], "X-Amz-Cf-Id":["t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRCzD7T1Rdhbtiv1Q=="], "X-Amzn-Trace-Id":["Root=1-63dc2d1-25f90b9d1c753a783547f4dd"], "X-Forwarded-For":["XXX, XXX"], "X-Forwarded-Port":443, "X-Forwarded-Proto":["https"]}, "queryStringParameters":null, "multiValueQueryStringParameters":null, "pathParameters":null, {"accountId": "XXX", "stage": "Prod", "resourceId": "at73a1", "requestId": "ba09ecd2-acf3-40f6-89af-fad32df67597", "operationName": null, "identity": {"cognitoIdentityPoolId": null, "accountId": null, "cognitoIdentityId": null, "caller": null, "apiKey": null}, "hello", "httpMethod": "GET", "apiId": "XXX", "path": "/Prod/hello/", "authorizer": null}, "body": null, "isBase64Encoded": false}
2023/02/03[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.351000
09:24:37.351 [main] INFO helloworld.App - Retrieving https://checkip.amazonaws.com
2023/02/03[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.313000 {
    "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
    "traceId":
    "Root=1-63dc2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
    "xray_trace_id": "1-63dc2d1-25f90b9d1c753a783547f4dd",
    "functionVersion": "$LATEST",
    "Service": "service_undefined",
    "logStreamId": "2023/02/03[$LATEST]851411a899b545eea2cffeba4cfbec81",
    "executionEnvironment": "AWS_Lambda_java11"
}
2023/02/03[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000 END
RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765
2023/02/03[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000 REPORT
RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765 Duration: 4118.98 ms Billed Duration: 4119 ms Memory Size: 512 MB Max Memory Used: 152 MB Init Duration: 1155.47 ms
XRAY TraceId: 1-63dc2d1-25f90b9d1c753a783547f4dd SegmentId: 3a028fee19b895cb
Sampled: true

```

- This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
 sam delete
```

Managing log retention

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which CloudWatch automatically deletes the logs. To set up log retention, add the following to your AWS SAM template:

```
Resources:
HelloWorldFunction:
  Type: AWS::Serverless::Function
```

```
Properties:  
    # Omitting other properties  
  
LogGroup:  
    Type: AWS::Logs::LogGroup  
    Properties:  
        LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"  
        RetentionInDays: 7
```

Using AWS Lambda Powertools for Java and the AWS CDK for structured logging

Follow the steps below to download, build, and deploy a sample Hello World Java application with integrated [AWS Lambda Powertools for Java](#) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Java 11
- [AWS CLI version 2](#)
- [AWS CDK version 2](#)
- [AWS SAM CLI version 1.75 or later](#). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS CDK application

1. Create a project directory for your new application.

```
mkdir hello-world  
cd hello-world
```

2. Initialize the app.

```
cdk init app --language java
```

3. Create a maven project with the following command:

```
mkdir app  
cd app  
mvn archetype:generate -DgroupId=helloworld -DartifactId=Function -  
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

4. Open pom.xml in the hello-world\app\Function directory and replace the existing code with the following code that includes dependencies and maven plugins for Powertools.

```
<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>
        </plugin>
    </plugins>

```

```
</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.PluginsCacheFileTransformer"
                        ></transformer>
                </transformers>
                <createDependencyReducedPom>false</createDependencyReducedPom>
            </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. Create the hello-world\app\src\main\resource directory and create log4j.xml for the log configuration.

```
mkdir -p src/main/resource
cd src/main/resource
touch log4j.xml
```

6. Open log4j.xml and add the following code.

```
<?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. Open App.java from the hello-world\app\Function\src\main\java\helloworld directory and replace the existing code with the following code. This is the code for the Lambda function.

```
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. Open `HelloWorldStack.java` from the `hello-world\src\main\java\com\myorg` directory and replace the existing code with the following code. This code uses [Lambda Constructor](#) and the [ApiGatewayv2 Constructor](#) to create a REST API and a Lambda function.

```
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.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)
    .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. Open pom.xml from the hello-world directory and replace the existing code with the following 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 --&gt;
    &lt;dependency&gt;
        &lt;groupId&gt;software.amazon.awscdk&lt;/groupId&gt;
        &lt;artifactId&gt;aws-cdk-lib&lt;/artifactId&gt;
        &lt;version&gt;${cdk.version}&lt;/version&gt;
    &lt;/dependency&gt;
    &lt;dependency&gt;
        &lt;groupId&gt;software.constructs&lt;/groupId&gt;
        &lt;artifactId&gt;constructs&lt;/artifactId&gt;
        &lt;version&gt;${constructs.version}&lt;/version&gt;
    &lt;/dependency&gt;
    &lt;dependency&gt;
        &lt;groupId&gt;org.junit.jupiter&lt;/groupId&gt;
        &lt;artifactId&gt;junit-jupiter&lt;/artifactId&gt;
        &lt;version&gt;${junit.version}&lt;/version&gt;
        &lt;scope&gt;test&lt;/scope&gt;
    &lt;/dependency&gt;
    &lt;dependency&gt;
        &lt;groupId&gt;software.amazon.awscdk&lt;/groupId&gt;
        &lt;artifactId&gt;apigatewayv2-alpha&lt;/artifactId&gt;
        &lt;version&gt;${cdk.version}-alpha.0&lt;/version&gt;
    &lt;/dependency&gt;
    &lt;dependency&gt;
        &lt;groupId&gt;software.amazon.awscdk&lt;/groupId&gt;
        &lt;artifactId&gt;apigatewayv2-integrations-alpha&lt;/artifactId&gt;
        &lt;version&gt;${cdk.version}-alpha.0&lt;/version&gt;
    &lt;/dependency&gt;
&lt;/dependencies&gt;
&lt;/project&gt;</pre>
```

10. Make sure you're in the hello-world directory and deploy your application.

```
cdk deploy
```

11. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query 'Stacks[0].Outputs[?OutputKey==`HttpApi`].OutputValue' --output text
```

12. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message": "hello world"}
```

13. To get the logs for the function, run [sam logs](#). For more information, see [Working with logs](#) in the *AWS Serverless Application Model Developer Guide*.

```
sam logs --stack-name HelloWorldStack
```

The log output looks like this:

```
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.095000
  INIT_START Runtime Version: java:11.v15      Runtime Version ARN: arn:aws:lambda:eu-
  central-1::runtime:0a25e3e7a1cc9ce404bc435eeb2ad358d8fa64338e618d0c224fe509403583ca
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.114000 Picked
  up JAVA_TOOL_OPTIONS: -XX:+TieredCompilation -XX:TieredStopAtLevel=1
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.793000
  Transforming org/apache/logging/log4j/core/lookup/JndiLookup
  (lambdainternal.CustomerClassLoader@1a6c5a9e)
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:35.252000 START
  RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765 Version: $LATEST
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.531000 {
  "_aws": {
    "Timestamp": 1675416276051,
    "CloudWatchMetrics": [
      {
        "Namespace": "sam-app-powertools-java",
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ],
        "Dimensions": [
          [
            "Service",
            "FunctionName"
          ]
        ]
      }
    ],
    "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
    "traceId":
      "Root=1-63dc2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
    "FunctionName": "sam-app-HelloWorldFunction-y9Iu1FLJJBGD",
    "functionVersion": "$LATEST",
    "ColdStart": 1.0,
    "Service": "service_undefined",
    "logStreamId": "2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
    "executionEnvironment": "AWS_Lambda_java11"
  }
}
```

14. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
cdk destroy
```

Using the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

Using the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the [Log groups page](#) on the CloudWatch console.
2. Choose the log group for your function (`/aws/lambda/your-function-name`).
3. Choose a log stream.

Each log stream corresponds to an [instance of your function \(p. 14\)](#). A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see [Error processor sample application for AWS Lambda \(p. 1015\)](#).

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

You can use the [AWS CLI](#) to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{  
    "StatusCode": 200,  
    "LogResult":  
        "U1RBULQgUmVxdWVzdElk0ia4N2QwNDRi0C1mMTU0LTExZTgt0GNkYS0y0Tc0YzV1NGZiMjEgVmVyc21vb...","  
    "ExecutedVersion": "$LATEST"  
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text | base64 -d
```

You should see the following output:

```
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
```

The base64 utility is available on Linux, macOS, and [Ubuntu on Windows](#). macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses sed to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the get-log-events command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

```
#!/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 and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{  
    "StatusCode": 200,  
    "ExecutedVersion": "$LATEST"  
}
```

```
{
  "events": [
    {
      "timestamp": 1559763003171,
      "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST\n",
      "ingestionTime": 1559763003309
    },
    {
      "timestamp": 1559763003173,
      "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf\n\tINFO\tENVIRONMENT VARIABLES\r{\r\t\t\"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\", \r\t\t...\""
      "ingestionTime": 1559763018353
    },
    {
      "timestamp": 1559763003173,
      "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf\n\tINFO\tEVENT\r{\r\t\t\"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"
}
```

Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.

Advanced logging with Log4j 2 and SLF4J

Note

AWS Lambda does not include Log4j2 in its managed runtimes or base container images. These are therefore not affected by the issues described in CVE-2021-44228, CVE-2021-45046, and CVE-2021-45105.

For cases where a customer function includes an impacted Log4j2 version, we have applied a change to the Lambda Java [managed runtimes \(p. 37\)](#) and [base container images \(p. 400\)](#) that helps to mitigate the issues in CVE-2021-44228, CVE-2021-45046, and CVE-2021-45105.

As a result of this change, customers using Log4j2 may see an additional log entry, similar to "Transforming org/apache/logging/log4j/core/lookup/JndiLookup (java.net.URLClassLoader@...)". Any log strings that reference the jndi mapper in the Log4j2 output will be replaced with "Patched JndiLookup::lookup()".

Independent of this change, we strongly encourage all customers whose functions include Log4j2 to update to the latest version. Specifically, customers using the aws-lambda-java-log4j2 library in their functions should update to version 1.5.0 (or later), and redeploy their functions. This version updates the underlying Log4j2 utility dependencies to version 2.17.0 (or later). The updated aws-lambda-java-log4j2 binary is available at the [Maven repository](#) and its source code is available in [Github](#).

To customize log output, support logging during unit tests, and log AWS SDK calls, use Apache Log4j 2 with SLF4J. Log4j is a logging library for Java programs that enables you to configure log levels and use appender libraries. SLF4J is a facade library that lets you change which library you use without changing your function code.

To add the request ID to your function's logs, use the appender in the [aws-lambda-java-log4j2 \(p. 393\)](#) library. The following example shows a Log4j 2 configuration file that adds a timestamp and request ID to all logs.

Example [src/main/resources/log4j2.xml](#) – Appender configuration

```
<Configuration status="WARN">
    <Appenders>
        <Lambda name="Lambda">
            <PatternLayout>
                <pattern>%d{yyyy-MM-dd HH:mm:ss} %X{AWSRequestId} %-5p %c{1} - %m%n</pattern>
            </PatternLayout>
        </Lambda>
    </Appenders>
    <Loggers>
        <Root level="INFO">
            <AppenderRef ref="Lambda"/>
        </Root>
        <Logger name="software.amazon.awssdk" level="WARN" />
        <Logger name="software.amazon.awssdk.request" level="DEBUG" />
    </Loggers>
</Configuration>
```

With this configuration, each line is prepended with the date, time, request ID, log level, and class name.

Example log format with appender

```
START RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Version: $LATEST
2020-03-18 08:52:43 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 INFO Handler - ENVIRONMENT
VARIABLES:
{
    "_HANDLER": "example.Handler",
    "AWS_EXECUTION_ENV": "AWS_Lambda_java8",
    "AWS_LAMBDA_FUNCTION_MEMORY_SIZE": "512",
    ...
}
2020-03-18 08:52:43 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 INFO Handler - CONTEXT:
{
    "memoryLimit": 512,
    "awsRequestId": "6bc28136-xmpl-4365-b021-0ce6b2e64ab0",
    "functionName": "java-console",
    ...
}
```

SLF4J is a facade library for logging in Java code. In your function code, you use the SLF4J logger factory to retrieve a logger with methods for log levels like `info()` and `warn()`. In your build configuration, you include the logging library and SLF4J adapter in the classpath. By changing the libraries in the build configuration, you can change the logger type without changing your function code. SLF4J is required to capture logs from the SDK for Java.

In the following example, the handler class uses SLF4J to retrieve a logger.

Example [src/main/java/example/HandlerS3.java](#) – Logging with SLF4J

```
import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;

// Handler value: example.Handler
public class HandlerS3 implements RequestHandler<S3Event, String>{
    private static final Logger logger = LoggerFactory.getLogger(HandlerS3.class);
    Gson gson = new GsonBuilder().setPrettyPrinting().create();
    @Override
    public String handleRequest(S3Event event, Context context)
    {
        ...
        logger.info("RECORD: " + record);
        logger.info("SOURCE BUCKET: " + srcBucket);
        logger.info("SOURCE KEY: " + srcKey);
        ...
    }
}
```

The build configuration takes runtime dependencies on the Lambda appender and SLF4J adapter, and implementation dependencies on Log4J 2.

Example [build.gradle](#) – Logging dependencies

```
dependencies {
    ...
    implementation 'org.apache.logging.log4j:log4j-api:[2.17.1,)'
    implementation 'org.apache.logging.log4j:log4j-core:[2.17.1,)'
    implementation 'org.apache.logging.log4j:log4j-slf4j18-impl:[2.17.1,)'
    ...
}
```

When you run your code locally for tests, the context object with the Lambda logger is not available, and there's no request ID for the Lambda appender to use. For example test configurations, see the sample applications in the next section.

Sample logging code

The GitHub repository for this guide includes sample applications that demonstrate the use of various logging configurations. Each sample application includes scripts for easy deployment and cleanup, an AWS SAM template, and supporting resources.

Sample Lambda applications in Java

- [java-basic](#) – A collection of minimal Java functions with unit tests and variable logging configuration.
- [java-events](#) – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the [aws-lambda-java-events \(p. 393\)](#) library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- [s3-java](#) – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- [Use API Gateway to invoke a Lambda function](#) – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.

The [java-basic](#) sample application shows a minimal logging configuration that supports logging tests. The handler code uses the `LambdaLogger` logger provided by the context object. For tests, the application uses a custom `TestLogger` class that implements the `LambdaLogger` interface with a Log4j

2 logger. It uses SLF4J as a facade for compatibility with the AWS SDK. Logging libraries are excluded from build output to keep the deployment package small.

AWS Lambda function errors in Java

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

This page describes how to view Lambda function invocation errors for the Java runtime using the Lambda console and the AWS CLI.

Sections

- [Syntax \(p. 429\)](#)
- [How it works \(p. 430\)](#)
- [Creating a function that returns exceptions \(p. 430\)](#)
- [Using the Lambda console \(p. 431\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 432\)](#)
- [Error handling in other AWS services \(p. 432\)](#)
- [Sample applications \(p. 433\)](#)
- [What's next? \(p. 433\)](#)

Syntax

In the following example, the runtime fails to deserialize the event into an object. The input is a valid JSON type, but it doesn't match the type expected by the handler method.

Example Lambda runtime error

```
{  
  "errorMessage": "An error occurred during JSON parsing",  
  "errorType": "java.lang.RuntimeException",  
  "stackTrace": [],  
  "cause": {  
    "errorMessage": "com.fasterxml.jackson.databind.exc.InvalidFormatException: Can not  
construct instance of java.lang.Integer from String value '1000,10': not a valid Integer  
value\n at [Source: lambdainternal.util.NativeMemoryAsInputStream@35fc6dc4; line: 1,  
column: 1] (through reference chain: java.lang.Object[0])",  
    "errorType": "java.io.UncheckedIOException",  
    "stackTrace": [],  
    "cause": {  
      "errorMessage": "Can not construct instance of java.lang.Integer  
from String value '1000,10': not a valid Integer value\n at [Source:  
lambdainternal.util.NativeMemoryAsInputStream@35fc6dc4; line: 1, column: 1] (through  
reference chain: java.lang.Object[0])",  
      "errorType": "com.fasterxml.jackson.databind.exc.InvalidFormatException",  
      "stackTrace": [  
  
        "com.fasterxml.jackson.databind.exc.InvalidFormatException.from(InvalidFormatException.java:55)",  
  
        "com.fasterxml.jackson.databind.DeserializationContext.weirdStringException(DeserializationContext.ja  
        ...  
      ]  
    }  
  }  
}
```

How it works

When you invoke a Lambda function, Lambda receives the invocation request and validates the permissions in your execution role, verifies that the event document is a valid JSON document, and checks parameter values.

If the request passes validation, Lambda sends the request to a function instance. The [Lambda runtime \(p. 37\)](#) environment converts the event document into an object, and passes it to your function handler.

If Lambda encounters an error, it returns an exception type, message, and HTTP status code that indicates the cause of the error. The client or service that invoked the Lambda function can handle the error programmatically, or pass it along to an end user. The correct error handling behavior depends on the type of application, the audience, and the source of the error.

The following list describes the range of status codes you can receive from Lambda.

2xx

A 2xx series error with a X-Amz-Function-Error header in the response indicates a Lambda runtime or function error. A 2xx series status code indicates that Lambda accepted the request, but instead of an error code, Lambda indicates the error by including the X-Amz-Function-Error header in the response.

4xx

A 4xx series error indicates an error that the invoking client or service can fix by modifying the request, requesting permission, or by retrying the request. 4xx series errors other than 429 generally indicate an error with the request.

5xx

A 5xx series error indicates an issue with Lambda, or an issue with the function's configuration or resources. 5xx series errors can indicate a temporary condition that can be resolved without any action by the user. These issues can't be addressed by the invoking client or service, but a Lambda function's owner may be able to fix the issue.

For a complete list of invocation errors, see [InvokeFunction errors \(p. 1262\)](#).

Creating a function that returns exceptions

You can create a Lambda function that displays human-readable error messages to users.

Note

To test this code, you need to include [InputLengthException.java](#) in your project src folder.

Example [src/main/java/example/HandlerDivide.java](#) – Runtime exception

```
import java.util.List;

// Handler value: example.HandlerDivide
public class HandlerDivide implements RequestHandler<List<Integer>, Integer>{
    Gson gson = new GsonBuilder().setPrettyPrinting().create();
    @Override
    public Integer handleRequest(List<Integer> event, Context context)
    {
        LambdaLogger logger = context.getLogger();
        // process event
        if (event.size() != 2)
        {
```

```
        throw new InputLengthException("Input must be a list that contains 2 numbers.");
    }
    int numerator = event.get(0);
    int denominator = event.get(1);
    logger.log("EVENT: " + gson.toJson(event));
    logger.log("EVENT TYPE: " + event.getClass().toString());
    return numerator/denominator;
}
}
```

When the function throws `InputLengthException`, the Java runtime serializes it into the following document.

Example error document (whitespace added)

```
{
    "errorMessage": "Input must be a list that contains 2 numbers.",
    "errorType": "java.lang.InputLengthException",
    "stackTrace": [
        "example.HandlerDivide.handleRequest(HandlerDivide.java:23)",
        "example.HandlerDivide.handleRequest(HandlerDivide.java:14)"
    ]
}
```

In this example, `InputLengthException` is a `RuntimeException`. The `RequestHandler interface` ([p. 391](#)) does not allow checked exceptions. The `RequestStreamHandler` interface supports throwing `IOException` errors.

The return statement in the previous example can also throw a runtime exception.

```
return numerator/denominator;
```

This code can return an arithmetic error.

```
{"errorMessage": "/ by zero", "errorType": "java.lang.ArithmaticException", "stackTrace": [
    "example.HandlerDivide.handleRequest(HandlerDivide.java:28)",
    "example.HandlerDivide.handleRequest(HandlerDivide.java:14)"
]}
```

Using the Lambda console

You can invoke your function on the Lambda console by configuring a test event and viewing the output. The output is captured in the function's execution logs and, when [active tracing](#) ([p. 807](#)) is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to test, and choose **Test**.
3. Under **Test event**, select **New event**.
4. Select a **Template**.
5. For **Name**, enter a name for the test. In the text entry box, enter the JSON test event.
6. Choose **Save changes**.
7. Choose **Test**.

The Lambda console invokes your function [synchronously](#) ([p. 120](#)) and displays the result. To see the response, logs, and other information, expand the **Details** section.

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- AWS Command Line Interface (AWS CLI) version 2
 - AWS CLI – Quick configuration with aws configure

When you invoke a Lambda function in the AWS CLI, the AWS CLI splits the response into two documents. The AWS CLI response is displayed in your command prompt. If an error has occurred, the response contains a `FunctionError` field. The invocation response or error returned by the function is written to an output file. For example, `output.json` or `output.txt`.

The following [invoke](#) command example demonstrates how to invoke a function and write the invocation response to an output.txt file.

```
aws lambda invoke \
--function-name my-function \
--cli-binary-format raw-in-base64-out \
--payload '{"key1": "value1", "key2": "value2", "key3": "value3"}' output.txt
```

You should see the AWS CLI response in your command prompt:

```
{  
    "StatusCode": 200,  
    "FunctionError": "Unhandled",  
    "ExecutedVersion": "$LATEST"  
}
```

You should see the function invocation response in the output.txt file. In the same command prompt, you can also view the output in your command prompt using:

```
cat output.txt
```

You should see the invocation response in your command prompt.

```
{"errorMessage": "Input must contain 2 numbers.", "errorType": "java.lang.InputLengthException", "stackTrace": ["example.HandlerDivide.handleRequest(HandlerDivide.java:23)", "example.HandlerDivide.handleRequest(Han
```

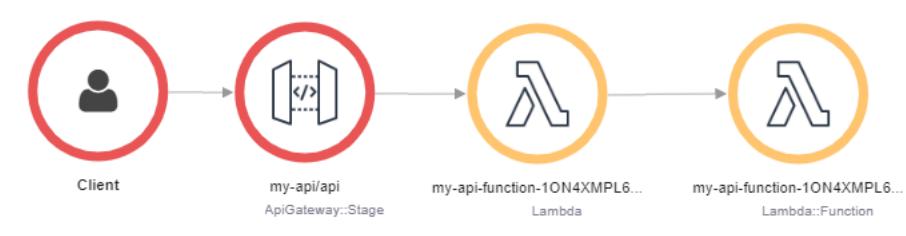
Lambda also records up to 256 KB of the error object in the function's logs. For more information, see [AWS Lambda function logging in Java \(p. 410\)](#).

Error handling in other AWS services

When another AWS service invokes your function, the service chooses the invocation type and retry behavior. AWS services can invoke your function on a schedule, in response to a lifecycle event on a resource, or to serve a request from a user. Some services invoke functions asynchronously and let Lambda handle errors, while others retry or pass errors back to the user.

For example, API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502 error code. To customize the error response, you must catch errors in your code and format a response in the required format.

We recommend using AWS X-Ray to determine the source of an error and its cause. X-Ray allows you to find out which component encountered an error, and see details about the errors. The following example shows a function error that resulted in a 502 response from API Gateway.



For more information, see [Instrumenting Java code in AWS Lambda \(p. 434\)](#).

Sample applications

The GitHub repository for this guide includes sample applications that demonstrate the use of the errors. Each sample application includes scripts for easy deployment and cleanup, an AWS Serverless Application Model (AWS SAM) template, and supporting resources.

Sample Lambda applications in Java

- [java-basic](#) – A collection of minimal Java functions with unit tests and variable logging configuration.
- [java-events](#) – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the [aws-lambda-java-events \(p. 393\)](#) library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- [s3-java](#) – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- [Use API Gateway to invoke a Lambda function](#) – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.

The `java-basic` function includes a handler (`HandlerDivide`) that returns a custom runtime exception. The `HandlerStream` handler implements the `RequestStreamHandler` and can throw an `IException` checked exception.

What's next?

- Learn how to show logging events for your Lambda function on the [the section called "Logging" \(p. 410\)](#) page.

Instrumenting Java code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of two SDK libraries:

- [AWS Distro for OpenTelemetry \(ADOT\)](#) – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- [AWS X-Ray SDK for Java](#) – An SDK for generating and sending trace data to X-Ray.
- [AWS Lambda Powertools for Java](#) – A developer toolkit to implement Serverless best practices and increase developer velocity.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

Important

The X-Ray and AWS Lambda Powertools SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see [Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs](#).

Sections

- [Using AWS Lambda Powertools for Java and AWS SAM for tracing \(p. 434\)](#)
- [Using AWS Lambda Powertools for Java and the AWS CDK for tracing \(p. 436\)](#)
- [Using ADOT to instrument your Java functions \(p. 444\)](#)
- [Using the X-Ray SDK to instrument your Java functions \(p. 444\)](#)
- [Activating tracing with the Lambda console \(p. 445\)](#)
- [Activating tracing with the Lambda API \(p. 445\)](#)
- [Activating tracing with AWS CloudFormation \(p. 445\)](#)
- [Interpreting an X-Ray trace \(p. 446\)](#)
- [Storing runtime dependencies in a layer \(X-Ray SDK\) \(p. 447\)](#)
- [X-Ray tracing in sample applications \(X-Ray SDK\) \(p. 448\)](#)

Using AWS Lambda Powertools for Java and AWS SAM for tracing

Follow the steps below to download, build, and deploy a sample Hello World Java application with integrated [AWS Lambda Powertools for Java](#) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Java 11
- [AWS CLI version 2](#)
- [AWS SAM CLI version 1.75 or later](#). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World Java template.

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type Zip  
--runtime java11 --no-tracing
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.

Note

For **HelloWorldFunction** may not have authorization defined, Is this okay?, make sure to enter y.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message": "hello world"}
```

7. To get the traces for the function, run [sam traces](#).

```
sam traces
```

The trace output looks like this:

```
New XRay Service Graph  
Start time: 2023-02-03 14:31:48+01:00  
End time: 2023-02-03 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 (2023-02-03T14:31:48.500000) 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. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

Using AWS Lambda Powertools for Java and the AWS CDK for tracing

Follow the steps below to download, build, and deploy a sample Hello World Java application with integrated [AWS Lambda Powertools for Java](#) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Java 11
- [AWS CLI version 2](#)
- [AWS CDK version 2](#)
- [AWS SAM CLI version 1.75 or later](#). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS CDK application

1. Create a project directory for your new application.

```
mkdir hello-world
cd hello-world
```

2. Initialize the app.

```
cdk init app --language java
```

3. Create a maven project with the following command:

```
mkdir app
cd app
mvn archetype:generate -DgroupId=helloworld -DartifactId=Function -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

4. Open pom.xml in the hello-world\app\Function directory and replace the existing code with the following code that includes dependencies and maven plugins for Powertools.

```
<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>
    </plugin>
  </plugins>
</build>

```

```
<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.PluginsCacheFileTransformer"
                        ></transformer>
                </transformers>
                <createDependencyReducedPom>false</createDependencyReducedPom>
            <createDependencyReducedPom>
                <finalName>function</finalName>
            </createDependencyReducedPom>
            </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. Create the hello-world\app\src\main\resource directory and create log4j.xml for the log configuration.

```
mkdir -p src/main/resource
cd src/main/resource
touch log4j.xml
```

6. Open log4j.xml and add the following code.

```
<?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. Open App.java from the hello-world\app\Function\src\main\java\helloworld directory and replace the existing code with the following code. This is the code for the Lambda function.

```
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. Open `HelloWorldStack.java` from the `hello-world\src\main\java\com\myorg` directory and replace the existing code with the following code. This code uses [Lambda Constructor](#) and the [ApiGatewayv2 Constructor](#) to create a REST API and a Lambda function.

```

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. Open pom.xml from the hello-world directory and replace the existing code with the following 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>
    </dependencies>

```

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>apigatewayv2-integrations-alpha</artifactId>
    <version>${cdk.version}-alpha.0</version>
</dependency>
</dependencies>
</project>
```

10. Make sure you're in the hello-world directory and deploy your application.

```
cdk deploy
```

11. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
'Stacks[0].Outputs[?OutputKey==`HttpApi`].OutputValue' --output text
```

12. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message": "hello world"}
```

13. To get the traces for the function, run [sam traces](#).

```
sam traces
```

The trace output looks like this:

```
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-YBg8yfYt0c9j -
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-YBg8yfYt0c9j -
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-YBg8yfYt0c9j - 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-YBg8yfYt0c9j [HTTP: 200]
- 0.016s - sam-app-HelloWorldFunction-YBg8yfYt0c9j
- 0.739s - Initialization
- 0.016s - Invocation
- 0.013s - ## lambda_handler
- 0.000s - ## app.hello
- 0.000s - Overhead
```

14. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
cdk destroy
```

Using ADOT to instrument your Java functions

ADOT provides fully managed Lambda [layers \(p. 11\)](#) that package everything you need to collect telemetry data using the OTel SDK. By consuming this layer, you can instrument your Lambda functions without having to modify any function code. You can also configure your layer to do custom initialization of OTel. For more information, see [Custom configuration for the ADOT Collector on Lambda](#) in the ADOT documentation.

For Java runtimes, you can choose between two layers to consume:

- **AWS managed Lambda layer for ADOT Java (Auto-instrumentation Agent)** – This layer automatically transforms your function code at startup to collect tracing data. For detailed instructions on how to consume this layer together with the ADOT Java agent, see [AWS Distro for OpenTelemetry Lambda Support for Java \(Auto-instrumentation Agent\)](#) in the ADOT documentation.
- **AWS managed Lambda layer for ADOT Java** – This layer also provides built-in instrumentation for Lambda functions, but it requires a few manual code changes to initialize the OTel SDK. For detailed instructions on how to consume this layer, see [AWS Distro for OpenTelemetry Lambda Support for Java](#) in the ADOT documentation.

Using the X-Ray SDK to instrument your Java functions

To record data about calls that your function makes to other resources and services in your application, you can add the X-Ray SDK for Java to your build configuration. The following example shows a Gradle build configuration that includes the libraries that activate automatic instrumentation of AWS SDK for Java 2.x clients.

Example [build.gradle](#) – Tracing dependencies

```
dependencies {
    implementation platform('software.amazon.awssdk:bom:2.15.0')
    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-instrumentor'
    ...
}
```

After you add the correct dependencies and make the necessary code changes, activate tracing in your function's configuration via the Lambda console or the API.

Activating tracing with the Lambda console

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring and operations tools**.
4. Choose **Edit**.
5. Under **X-Ray**, toggle on **Active tracing**.
6. Choose **Save**.

Activating tracing with the Lambda API

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- [UpdateFunctionConfiguration \(p. 1377\)](#)
- [GetFunctionConfiguration \(p. 1229\)](#)
- [CreateFunction \(p. 1165\)](#)

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Activating tracing with AWS CloudFormation

To activate tracing on an `AWS::Lambda::Function` resource in an AWS CloudFormation template, use the `TracingConfig` property.

Example [function-inline.yml](#) – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...

```

For an AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` resource, use the `Tracing` property.

Example [template.yml](#) – Tracing configuration

```
Resources:
```

```
function:  
  Type: AWS::Serverless::Function  
Properties:  
  Tracing: Active  
  ...
```

Interpreting an X-Ray trace

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's [execution role \(p. 816\)](#). Otherwise, add the [AWSXRayDaemonWriteAccess](#) policy to the execution role.

After you've configured active tracing, you can observe specific requests through your application. The [X-Ray service graph](#) shows information about your application and all its components. The following example from the [error processor \(p. 1015\)](#) sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.

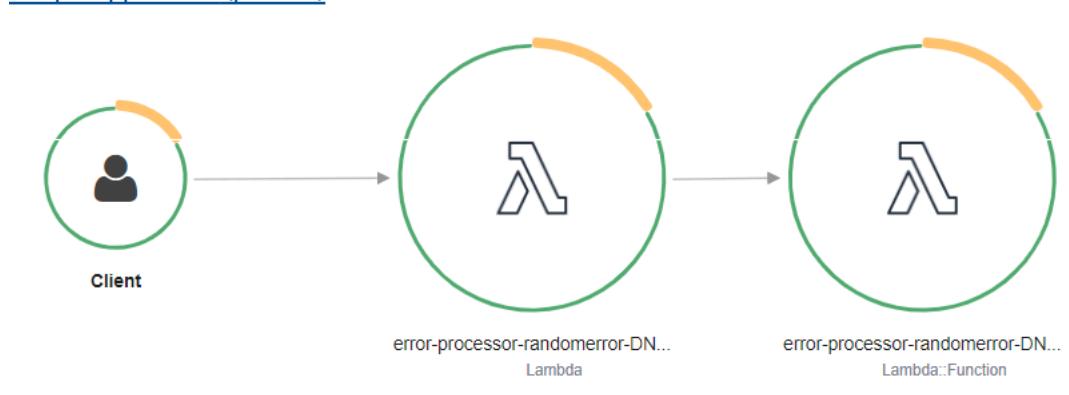


X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

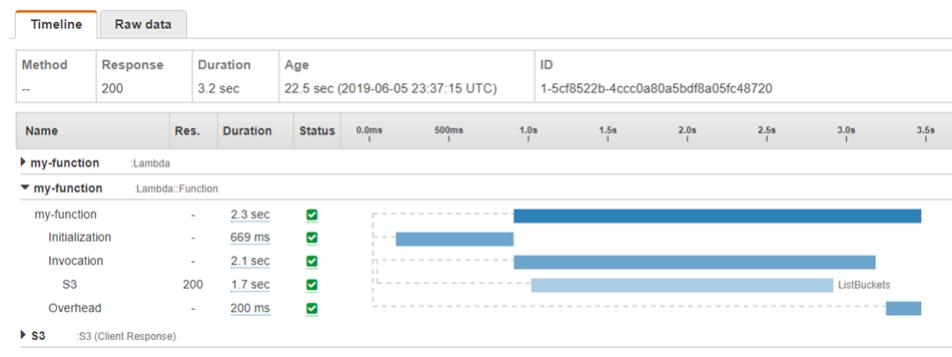
Note

You cannot configure the X-Ray sampling rate for your functions.

When using active tracing, Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes for the primary function from the [error processor sample application \(p. 1015\)](#).



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of AWS::Lambda and the other has origin AWS::Function.



This example expands the function segment to show its three subsegments:

- **Initialization** – Represents time spent loading your function and running [initialization code \(p. 13\)](#). This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see [AWS X-Ray SDK for Java](#) in the [AWS X-Ray Developer Guide](#).

Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see [AWS X-Ray pricing](#).

Storing runtime dependencies in a layer (X-Ray SDK)

If you use the X-Ray SDK to instrument AWS SDK clients your function code, your deployment package can become quite large. To avoid uploading runtime dependencies every time you update your function code, package the X-Ray SDK in a [Lambda layer \(p. 93\)](#).

The following example shows an AWS::Serverless::LayerVersion resource that stores the AWS SDK for Java and X-Ray SDK for Java.

Example [template.yml](#) – Dependencies layer

```
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:  
- java8
```

With this configuration, you update the library layer only if you change your runtime dependencies. Since the function deployment package contains only your code, this can help reduce upload times.

Creating a layer for dependencies requires build configuration changes to generate the layer archive prior to deployment. For a working example, see the [java-basic](#) sample application on GitHub.

X-Ray tracing in sample applications (X-Ray SDK)

The GitHub repository for this guide includes sample applications that demonstrate the use of X-Ray tracing. Each sample application includes scripts for easy deployment and cleanup, an AWS SAM template, and supporting resources.

Sample Lambda applications in Java

- [java-basic](#) – A collection of minimal Java functions with unit tests and variable logging configuration.
- [java-events](#) – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the [aws-lambda-java-events \(p. 393\)](#) library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- [s3-java](#) – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- [Use API Gateway to invoke a Lambda function](#) – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.

All of the sample applications have active tracing enabled for Lambda functions. For example, the [s3-java](#) application shows automatic instrumentation of AWS SDK for Java 2.x clients, segment management for tests, custom subsegments, and the use of Lambda layers to store runtime dependencies.

Creating a deployment package using Eclipse

This section shows how to package your Java code into a deployment package using Eclipse IDE and Maven plugin for Eclipse.

Note

The AWS SDK Eclipse Toolkit provides an Eclipse plugin for you to both create a deployment package and also upload it to create a Lambda function. If you can use Eclipse IDE as your development environment, this plugin enables you to author Java code, create and upload a deployment package, and create your Lambda function. For more information, see the [AWS Toolkit for Eclipse Getting Started Guide](#). For an example of using the toolkit for authoring Lambda functions, see [Using Lambda with the AWS toolkit for Eclipse](#).

Topics

- [Prerequisites \(p. 449\)](#)
- [Create and build a project \(p. 449\)](#)

Prerequisites

Install the **Maven** Plugin for Eclipse.

1. Start Eclipse. From the **Help** menu in Eclipse, choose **Install New Software**.
2. In the **Install** window, type `http://download.eclipse.org/technology/m2e/releases` in the **Work with:** box, and choose **Add**.
3. Follow the steps to complete the setup.

Create and build a project

In this step, you start Eclipse and create a Maven project. You will add the necessary dependencies, and build the project. The build will produce a .jar, which is your deployment package.

1. Create a new Maven project in Eclipse.
 - a. From the **File** menu, choose **New**, and then choose **Project**.
 - b. In the **New Project** window, choose **Maven Project**.
 - c. In the **New Maven Project** window, choose **Create a simple project**, and leave other default selections.
 - d. In the **New Maven Project, Configure project** windows, type the following **Artifact** information:
 - **Group Id:** doc-examples
 - **Artifact Id:** lambda-java-example
 - **Version:** 0.0.1-SNAPSHOT
 - **Packaging:** jar
 - **Name:** lambda-java-example
2. Add the `aws-lambda-java-core` dependency to the `pom.xml` file.

It provides definitions of the `RequestHandler`, `RequestStreamHandler`, and `Context` interfaces. This allows you to compile code that you can use with AWS Lambda.

 - a. Open the context (right-click) menu for the `pom.xml` file, choose **Maven**, and then choose **Add Dependency**.
 - b. In the **Add Dependency** windows, type the following values:

Group Id: com.amazonaws

Artifact Id: aws-lambda-java-core

Version: 1.2.2

Note

If you are following other tutorial topics in this guide, the specific tutorials might require you to add more dependencies. Make sure to add those dependencies as required.

3. Add Java class to the project.
 - a. Open the context (right-click) menu for the `src/main/java` subdirectory in the project, choose **New**, and then choose **Class**.
 - b. In the **New Java Class** window, type the following values:
 - **Package:** example
 - **Name:** Hello

Note

If you are following other tutorial topics in this guide, the specific tutorials might recommend different package name or class name.

- c. Add your Java code. If you are following other tutorial topics in this guide, add the provided code.
4. Build the project.

Open the context (right-click) menu for the project in **Package Explorer**, choose **Run As**, and then choose **Maven Build ...**. In the **Edit Configuration** window, type **package** in the **Goals** box.

Note

The resulting `.jar`, `lambda-java-example-0.0.1-SNAPSHOT.jar`, is not the final standalone `.jar` that you can use as your deployment package. In the next step, you add the Apache maven-shade-plugin to create the standalone `.jar`. For more information, go to [Apache Maven Shade plugin](#).

5. Add the maven-shade-plugin plugin and rebuild.

The maven-shade-plugin will take artifacts (jars) produced by the `package` goal (produces customer code `.jar`), and created a standalone `.jar` that contains the compiled customer code, and the resolved dependencies from the `pom.xml`.

- a. Open the context (right-click) menu for the `pom.xml` file, choose **Maven**, and then choose **Add Plugin**.
- b. In the **Add Plugin** window, type the following values:
 - **Group Id:** org.apache.maven.plugins
 - **Artifact Id:** maven-shade-plugin
 - **Version:** 3.2.2
- c. Now build again.

This time we will create the jar as before, and then use the maven-shade-plugin to pull in dependencies to make the standalone `.jar`.

- i. Open the context (right-click) menu for the project, choose **Run As**, and then choose **Maven build ...**.

- ii. In the **Edit Configuration** windows, type **package shade:shade** in the **Goals** box.
- iii. Choose Run.

You can find the resulting standalone .jar (that is, your deployment package), in the /target subdirectory.

Open the context (right-click) menu for the /target subdirectory, choose **Show In**, choose **System Explorer**, and you will find the `lambda-java-example-0.0.1-SNAPSHOT.jar`.

Java sample applications for AWS Lambda

The GitHub repository for this guide provides sample applications that demonstrate the use of Java in AWS Lambda. Each sample application includes scripts for easy deployment and cleanup, an AWS CloudFormation template, and supporting resources.

Sample Lambda applications in Java

- [java-basic](#) – A collection of minimal Java functions with unit tests and variable logging configuration.
- [java-events](#) – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the [aws-lambda-java-events \(p. 393\)](#) library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- [s3-java](#) – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- [Use API Gateway to invoke a Lambda function](#) – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.

Sample Spring applications in Lambda

- [spring-cloud-function-samples](#) – An example that shows how to use the [Spring Cloud Function](#) framework to create AWS Lambda functions.

If you're new to Lambda functions in Java, start with the `java-basic` examples. To get started with Lambda event sources, see the `java-events` examples. Both of these example sets show the use of Lambda's Java libraries, environment variables, the AWS SDK, and the AWS X-Ray SDK. Each example uses a Lambda layer to package its dependencies separately from the function code, which speeds up deployment times. These examples require minimal setup and you can deploy them from the command line in less than a minute.

Building Lambda functions with Go

The following sections explain how common programming patterns and core concepts apply when authoring Lambda function code in [Go](#).

Go

Name	Identifier	Operating system	Architectures	Deprecation (Phase 1)
Go 1.x	go1.x	Amazon Linux	x86_64	

Note

Runtimes that use the Amazon Linux operating system, such as Go 1.x, do not support the arm64 architecture. To use arm64 architecture, you can run Go with the provided.al2 runtime.

Lambda provides the following tools and libraries for the Go runtime:

Tools and libraries for Go

- [AWS SDK for Go](#): the official AWS SDK for the Go programming language.
- [github.com/aws/aws-lambda-go/lambda](#): The implementation of the Lambda programming model for Go. This package is used by AWS Lambda to invoke your [handler \(p. 454\)](#).
- [github.com/aws/aws-lambda-go/lambdacontext](#): Helpers for accessing context information from the [context object \(p. 458\)](#).
- [github.com/aws/aws-lambda-go/events](#): This library provides type definitions for common event source integrations.
- [github.com/aws/aws-lambda-go/cmd/build-lambda-zip](#): This tool can be used to create a .zip file archive on Windows.

For more information, see [aws-lambda-go](#) on GitHub.

Lambda provides the following sample applications for the Go runtime:

Sample Lambda applications in Go

- [blank-go](#) – A Go function that shows the use of Lambda's Go libraries, logging, environment variables, and the AWS SDK.

Topics

- [AWS Lambda function handler in Go \(p. 454\)](#)
- [AWS Lambda context object in Go \(p. 458\)](#)
- [Deploy Go Lambda functions with .zip file archives \(p. 460\)](#)
- [Deploy Go Lambda functions with container images \(p. 465\)](#)
- [AWS Lambda function logging in Go \(p. 473\)](#)
- [AWS Lambda function errors in Go \(p. 478\)](#)
- [Instrumenting Go code in AWS Lambda \(p. 482\)](#)
- [Using environment variables \(p. 486\)](#)

AWS Lambda function handler in Go

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. When the handler exits or returns a response, it becomes available to handle another event.

A Lambda function written in [Go](#) is authored as a Go executable. In your Lambda function code, you need to include the github.com/aws/aws-lambda-go/lambda package, which implements the Lambda programming model for Go. In addition, you need to implement handler function code and a `main()` function.

```
package main

import (
    "fmt"
    "context"
    "github.com/aws/aws-lambda-go/lambda"
)

type MyEvent struct {
    Name string `json:"name"`
}

func HandleRequest(ctx context.Context, name MyEvent) (string, error) {
    return fmt.Sprintf("Hello %s!", name.Name), nil
}

func main() {
    lambda.Start(HandleRequest)
}
```

Note the following:

- **package main:** In Go, the package containing `func main()` must always be named `main`.
- **import:** Use this to include the libraries your Lambda function requires. In this instance, it includes:
 - **context:** [AWS Lambda context object in Go \(p. 458\)](#).
 - **fmt:** The Go [Formatting](#) object used to format the return value of your function.
 - **github.com/aws/aws-lambda-go/lambda:** As mentioned previously, implements the Lambda programming model for Go.
- **func HandleRequest(ctx context.Context, name MyEvent) (string, error):** This is your Lambda handler signature and includes the code which will be executed. In addition, the parameters included denote the following:
 - **ctx context.Context:** Provides runtime information for your Lambda function invocation. `ctx` is the variable you declare to leverage the information available via [AWS Lambda context object in Go \(p. 458\)](#).
 - **name MyEvent:** An input type with a variable name of `name` whose value will be returned in the `return` statement.
 - **string, error:** Returns two values: `string` for success and standard `error` information. For more information on custom error handling, see [AWS Lambda function errors in Go \(p. 478\)](#).
 - **return fmt.Sprintf("Hello %s!", name), nil:** Simply returns a formatted "Hello" greeting with the name you supplied in the input event. `nil` indicates there were no errors and the function executed successfully.
- **func main():** The entry point that runs your Lambda function code. This is required.

By adding `lambda.Start(HandleRequest)` between `func main() {}` code brackets, your Lambda function will be executed. Per Go language standards, the opening bracket, `{` must be placed directly at end the of the `main()` function signature.

Naming

When you configure a function in Go, the value of the handler setting is the executable file name. For example, if you set the value of the handler to `Handler`, Lambda will call the `main()` function in the `Handler` executable file.

To change the function handler name in the Lambda console, on the **Runtime** settings pane, choose **edit**.

Lambda function handler using structured types

In the example above, the input type was a simple string. But you can also pass in structured events to your function handler:

```
package main

import (
    "fmt"
    "github.com/aws/aws-lambda-go/lambda"
)

type MyEvent struct {
    Name string `json:"What is your name?"`
    Age int     `json:"How old are you?"`
}

type MyResponse struct {
    Message string `json:"Answer:"`
}

func HandleLambdaEvent(event MyEvent) (MyResponse, error) {
    return MyResponse{Message: fmt.Sprintf("%s is %d years old!", event.Name,
    event.Age)}, nil
}

func main() {
    lambda.Start(HandleLambdaEvent)
}
```

Your request would then look like this:

```
# request
{
    "What is your name?": "Jim",
    "How old are you?": 33
}
```

And the response would look like this:

```
# response
{
    "Answer": "Jim is 33 years old!"
```

}

To be exported, field names in the event struct must be capitalized. For more information on handling events from AWS event sources, see [aws-lambda-go/events](#).

Valid handler signatures

You have several options when building a Lambda function handler in Go, but you must adhere to the following rules:

- The handler must be a function.
- The handler may take between 0 and 2 arguments. If there are two arguments, the first argument must implement `context.Context`.
- The handler may return between 0 and 2 arguments. If there is a single return value, it must implement `error`. If there are two return values, the second value must implement `error`. For more information on implementing error-handling information, see [AWS Lambda function errors in Go \(p. 478\)](#).

The following lists valid handler signatures. `TIn` and `TOut` represent types compatible with the `encoding/json` standard library. For more information, see [func Unmarshal](#) to learn how these types are deserialized.

- `func ()`
- `func () error`
- `func (TIn) error`
- `func () (TOut, error)`
- `func (context.Context) error`
- `func (context.Context, TIn) error`
- `func (context.Context) (TOut, error)`
- `func (context.Context, TIn) (TOut, error)`

Using global state

You can declare and modify global variables that are independent of your Lambda function's handler code. In addition, your handler may declare an `init` function that is executed when your handler is loaded. This behaves the same in AWS Lambda as it does in standard Go programs. A single instance of your Lambda function will never handle multiple events simultaneously.

```
package main

import (
    "log"
    "github.com/aws/aws-lambda-go/lambda"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/s3"
```

```
        "github.com/aws/aws-sdk-go/aws"
)

var invokeCount = 0
var myObjects []*s3.Object
func init() {
    svc := s3.New(session.New())
    input := &s3.ListObjectsV2Input{
        Bucket: aws.String("examplebucket"),
    }
    result, _ := svc.ListObjectsV2(input)
    myObjects = result.Contents
}

func LambdaHandler() (int, error) {
    invokeCount = invokeCount + 1
    log.Print(myObjects)
    return invokeCount, nil
}

func main() {
    lambda.Start(LambdaHandler)
}
```

AWS Lambda context object in Go

When Lambda runs your function, it passes a context object to the [handler \(p. 454\)](#). This object provides methods and properties with information about the invocation, function, and execution environment.

The Lambda context library provides the following global variables, methods, and properties.

Global variables

- `FunctionName` – The name of the Lambda function.
- `FunctionVersion` – The [version \(p. 83\)](#) of the function.
- `MemoryLimitInMB` – The amount of memory that's allocated for the function.
- `LogGroupName` – The log group for the function.
- `LogStreamName` – The log stream for the function instance.

Context methods

- `Deadline` – Returns the date that the execution times out, in Unix time milliseconds.

Context properties

- `InvokedFunctionArn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `AwsRequestId` – The identifier of the invocation request.
- `Identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
- `ClientContext` – (mobile apps) Client context that's provided to Lambda by the client application.

Accessing invoke context information

Lambda functions have access to metadata about their environment and the invocation request. This can be accessed at [Package context](#). Should your handler include `context.Context` as a parameter, Lambda will insert information about your function into the context's `Value` property. Note that you need to import the `lambdacontext` library to access the contents of the `context.Context` object.

```
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)
}
```

In the example above, `lc` is the variable used to consume the information that the context object captured and `log.Print(lc.Identity.CognitoIdentityPoolID)` prints that information, in this case, the `CognitoIdentityPoolID`.

The following example introduces how to use the context object to monitor how long your Lambda function takes to complete. This allows you to analyze performance expectations and adjust your function code accordingly, if needed.

```
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.Println("hello!")
            time.Sleep(50 * time.Millisecond)
        }
    }
}

func main() {
    lambda.Start(LongRunningHandler)
}
```

Deploy Go Lambda functions with .zip file archives

Your AWS Lambda function's code consists of scripts or compiled programs and their dependencies. You use a *deployment package* to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives.

This page describes how to create a .zip file as your deployment package for the Go runtime, and then use the .zip file to deploy your function code to AWS Lambda using the AWS Command Line Interface (AWS CLI).

Sections

- [Prerequisites \(p. 460\)](#)
- [Tools and libraries \(p. 460\)](#)
- [Sample applications \(p. 460\)](#)
- [Creating a .zip file on macOS and Linux \(p. 461\)](#)
- [Creating a .zip file on Windows \(p. 461\)](#)
- [Creating a Lambda function using a .zip archive \(p. 462\)](#)
- [Build Go with the provided.al2 runtime \(p. 463\)](#)

Prerequisites

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

Tools and libraries

Lambda provides the following tools and libraries for the Go runtime:

Tools and libraries for Go

- [AWS SDK for Go](#): the official AWS SDK for the Go programming language.
- [github.com/aws/aws-lambda-go/lambda](#): The implementation of the Lambda programming model for Go. This package is used by AWS Lambda to invoke your [handler \(p. 454\)](#).
- [github.com/aws/aws-lambda-go/lambdacontext](#): Helpers for accessing context information from the [context object \(p. 458\)](#).
- [github.com/aws/aws-lambda-go/events](#): This library provides type definitions for common event source integrations.
- [github.com/aws/aws-lambda-go/cmd/build-lambda-zip](#): This tool can be used to create a .zip file archive on Windows.

For more information, see [aws-lambda-go](#) on GitHub.

Sample applications

Lambda provides the following sample applications for the Go runtime:

Sample Lambda applications in Go

- [blank-go](#) – A Go function that shows the use of Lambda's Go libraries, logging, environment variables, and the AWS SDK.

Creating a .zip file on macOS and Linux

The following steps demonstrate how to download the [lambda](#) library from GitHub with go get, and compile your executable with [go build](#).

1. Download the **lambda** library from GitHub.

```
go get github.com/aws/aws-lambda-go/lambda
```

2. Compile your executable.

```
GOOS=linux GOARCH=amd64 go build -o main main.go
```

Setting GOOS to linux ensures that the compiled executable is compatible with the [Go runtime \(p. 37\)](#), even if you compile it in a non-Linux environment.

3. (Optional) If your main package consists of multiple files, use the following [go build](#) command to compile the package:

```
GOOS=linux GOARCH=amd64 go build main
```

4. (Optional) You may need to compile packages with CGO_ENABLED=0 set on Linux:

```
GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build -o main main.go
```

This command creates a stable binary package for standard C library (libc) versions, which may be different on Lambda and other devices.

5. Lambda uses POSIX file permissions, so you may need to [set permissions for the deployment package folder](#) before you create the .zip file archive.
6. Create a deployment package by packaging the executable in a .zip file.

```
zip main.zip main
```

Creating a .zip file on Windows

The following steps demonstrate how to download the [lambda](#) library from GitHub with go get, download the [build-lambda-zip](#) tool for Windows from GitHub with go install, and compile your executable with [go build](#).

Note

If you have not already done so, you must install [git](#) and then add the git executable to your Windows %PATH% environment variable.

1. Download the **lambda** library from GitHub.

```
go get github.com/aws/aws-lambda-go/lambda
```

2. Download the **build-lambda-zip** tool from GitHub.

```
go.exe install github.com/aws/aws-lambda-go/cmd/build-lambda-zip@latest
```

3. Use the tool from your GOPATH to create a .zip file. If you have a default installation of Go, the tool is typically in %USERPROFILE%\Go\bin. Otherwise, navigate to where you installed the Go runtime and do one of the following:

cmd.exe

In cmd.exe, run the following:

```
set GOOS=linux
set GOARCH=amd64
set CGO_ENABLED=0
go build -o main main.go
%USERPROFILE%\Go\bin\build-lambda-zip.exe -o main.zip main
```

PowerShell

In PowerShell, run the following:

```
$env:GOOS = "linux"
$env:GOARCH = "amd64"
$env:CGO_ENABLED = "0"
go build -o main main.go
~\Go\Bin\build-lambda-zip.exe -o main.zip main
```

Creating a Lambda function using a .zip archive

In addition to the main.zip deployment package you have created, to create a Lambda function you also need an execution role. The execution role grants the function permission to use AWS services such as Amazon CloudWatch Logs for log streaming and AWS X-Ray for request tracing. You can create an execution role for your function in the IAM console or using the AWS Command Line Interface (AWS CLI).

The following steps demonstrate how to create the execution role using the AWS CLI and then create a Lambda function using your .zip deployment package.

1. Create a trust policy giving the Lambda service permission to assume the execution role. Copy the following JSON and create a file named trust-policy.json in the current directory.

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

2. Create the execution role.

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

You should see the following output. Make a note of your role's ARN. You will need this to create your function.

```
{  
    "Role": {  
        "Path": "/",  
        "RoleName": "lambda-ex",  
        "RoleId": "AROAQFOXMPL6TZ6ITKWND",  
        "Arn": "arn:aws:iam::123456789012:role/lambda-ex",  
        "CreateDate": "2020-01-17T23:19:12Z",  
        "AssumeRolePolicyDocument": {  
            "Version": "2012-10-17",  
            "Statement": [  
                {  
                    "Effect": "Allow",  
                    "Principal": {  
                        "Service": "lambda.amazonaws.com"  
                    },  
                    "Action": "sts:AssumeRole"  
                }  
            ]  
        }  
    }  
}
```

3. Add permissions to the role using the `attach-role-policy` command. In the example below, you add the `AWSLambdaBasicExecutionRole` managed policy, which allows your Lambda function to upload logs to CloudWatch. If your Lambda function interacts with other AWS services, such as Amazon S3 or DynamoDB, you will need to add policies allowing your Lambda function to access these services. See [AWS managed policies for Lambda features \(p. 818\)](#) for more information.

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/  
service-role/AWSLambdaBasicExecutionRole
```

4. Create the function.

```
aws lambda create-function --function-name my-function --runtime go1.x --  
role arn:aws:iam::123456789012:role/lambda-ex --handler main --zip-file fileb://  
main.zip
```

Note

When you create a Go Lambda function using the AWS CLI, the value of the handler setting you define is the executable file name. For more information, see [AWS Lambda function handler in Go \(p. 454\)](#).

Build Go with the provided.al2 runtime

Go is implemented differently than other native runtimes. Lambda treats Go as a custom runtime, so you can create a Go function on the provided.al2 runtime. You can use the AWS SAM build command to build the .zip file package.

Using AWS SAM to build Go for AL2 function

1. Update the AWS SAM template to use the provided.al2 runtime. Also set the `BuildMethod` to `makefile`.

```
Resources:
```

```
HelloWorldFunction:  
  Type: AWS::Serverless::Function  
  Properties:  
    CodeUri: hello-world/  
    Handler: my.bootstrap.file  
    Runtime: provided.al2  
    Architectures: [arm64]  
  Metadata:  
    BuildMethod: makefile
```

Remove the Architectures property to build the package for the x86_64 instruction set architecture.

2. Add file makefile to the project folder, with the following contents:

```
GOOS=linux go build -o bootstrap  
cp ./bootstrap $(ARTIFACTS_DIR)/.
```

For an example application, download [Go on AL2](#). The readme file contains the instructions to build and run the application. You can also view the blog post [Migrating AWS Lambda functions to Amazon Linux 2](#).

Deploy Go Lambda functions with container images

You can deploy your Lambda function code as a [container image \(p. 881\)](#). To help you build a container image for your Go function, AWS provides the following resources:

- AWS base images for Lambda

These base images are preloaded with a language runtime and other components that are required to run the image on Lambda. AWS provides a Dockerfile for each of the base images to help with building your container image.

- Open-source runtime interface clients

If you use a community or private enterprise base image, you must add a [runtime interface client \(p. 883\)](#) to the base image to make it compatible with Lambda.

- Open-source runtime interface emulator

Lambda provides a runtime interface emulator (RIE) for you to test your function locally. The base images for Lambda and base images for custom .runtimes include the RIE. For other base images, you can download the RIE for [testing your image \(p. 884\)](#) locally.

The workflow for a function defined as a container image includes these steps:

1. Build your container image using the resources listed in this topic.
2. Upload the image to your [Amazon Elastic Container Registry \(Amazon ECR\) container registry \(p. 891\)](#).
3. [Create the function \(p. 113\)](#) or [update the function code \(p. 115\)](#) to deploy the image to an existing function.

Topics

- [AWS base images for Go \(p. 465\)](#)
- [Go runtime interface clients \(p. 466\)](#)
- [Using the Go:1.x base image \(p. 466\)](#)
- [Create a Go image from the provided.al2 base image \(p. 469\)](#)
- [Create a Go image from an alternative base image \(p. 470\)](#)

AWS base images for Go

AWS provides the following base image for Go:

Tags	Runtime	Operating system	Dockerfile	Deprecation
1	Go 1.x	Amazon Linux	Dockerfile for Go 1.x on GitHub	

Amazon ECR repository: [gallery.ecr.aws/lambda/go](#)

Go runtime interface clients

AWS does not provide a separate runtime interface client for Go. The `aws-lambda-go/lambda` package includes an implementation of the runtime interface.

Using the Go:1.x base image

Prerequisites

To complete the steps in this section, you must have the following:

- Go
- [Docker](#)
- [AWS Command Line Interface \(AWS CLI\) version 2](#)

Creating an image from a base image

1. Create a directory for the project, and then switch to that directory.

```
mkdir hello
cd hello
```

2. Initialize a new Go module.

```
go mod init example.com/hello-world
```

3. Add the **lambda** library as a dependency of your new module.

```
go get github.com/aws/aws-lambda-go/lambda
```

4. Create a file named `hello.go` and then open it in a text editor. This is the code for the Lambda function. You can use the following sample code for testing, or replace it with your own.

```
package main

import (
    "context"
    "fmt"
    "github.com/aws/aws-lambda-go/lambda"
)

type MyEvent struct {
    Name string `json:"name"`
}

func HandleRequest(ctx context.Context, name MyEvent) (string, error) {
    return fmt.Sprintf("Hello %s!", name.Name), nil
}

func main() {
    lambda.Start(HandleRequest)
}
```

5. Build the Go project. This command creates an executable called `main`.

```
GOOS=linux GOARCH=amd64 go build -o main hello.go
```

Setting GOOS to linux and GOARCH to amd64 ensures that the compiled executable is compatible with the architecture of the Go base image, even if you compile it in a non-Linux environment.

6. Create a new Dockerfile. The example Dockerfile uses the following configuration:

- FROM: The [URI of the base image](#) that you want to use.
- COPY: Copies the function code into the /var/task directory in your image.
- CMD: The Lambda function handler or executable.

```
FROM public.ecr.aws/lambda/go:1

# Copy function code
COPY main ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside of
# the Dockerfile)
CMD [ "main" ]
```

7. Build the Docker image with the [docker build](#) command. The following example names the image docker-image and gives it the test [tag](#).

```
docker build -t docker-image:test .
```

(Optional) Test the image locally

1. Start the Docker image with the [docker run](#) command. In this example, docker-image is the image name and test is the tag.

```
docker run -p 9000:8080 docker-image:test
```

This command runs the image as a container and creates a local endpoint at localhost:9000/2015-03-31/functions/function/invocations.

2. Test your application locally using the [RIE \(p. 884\)](#). From a new terminal window, post an event to the following endpoint using a [curl](#) command:

```
curl -XPOST "http://localhost:9000/2015-03-31/functions/function/invocations" -d
'{"Name": "World"}'
```

This command invokes the function running in the container image and returns a response.

3. Get the container ID.

```
docker ps
```

4. Use the [docker kill](#) command to stop the container. In this command, replace 3766c4ab331c with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the [get-login-password](#) command to authenticate the Docker CLI to your Amazon ECR registry.
 - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
 - Replace `111122223333` with your AWS account 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. Create a repository in Amazon ECR using the [create-repository](#) command.

```
aws ecr create-repository --repository-name hello-world --image-scanning-configuration  
scanOnPush=true --image-tag-mutability MUTABLE
```

If successful, you see a response like this:

```
{  
  "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. Copy the `repositoryUri` from the output in the previous step.
4. Run the [docker tag](#) command to tag your local image into your Amazon ECR repository as the latest version. In this command:
 - Replace `docker-image:test` with the name and [tag](#) of your Docker image.
 - Replace the Amazon ECR repository URI with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-  
world:latest
```

5. Run the [docker push](#) command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.
6. [Create an execution role \(p. 191\)](#) for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.
7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
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
```

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{  
  "ExecutedVersion": "$LATEST",  
  "StatusCode": 200  
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must create a new image version and store the image in the Amazon ECR repository. For more information, see [Updating function code \(p. 115\)](#).

Create a Go image from the provided.a12 base image

To build a container image for Go that runs on Amazon Linux 2, use the `provided.a12` base image. For more information about this base image, see [provided](#) in the Amazon ECR public gallery.

You include the `aws-lambda-go/lambda` package with your Go handler. This package implements the programming model for Go, including the runtime interface client. The `provided.a12` base image also includes the [RIE \(p. 884\)](#).

To build and deploy a Go function with the `provided.a12` base image.

1. Create a directory for the project, and then switch to that directory.

```
mkdir hello
cd hello
```

2. Initialize a new Go module.

```
go mod init example.com/hello-world
```

3. Add the `lambda` library as a dependency of your new module.

```
go get github.com/aws/aws-lambda-go/lambda
```

4. Create a file named `hello.go` and then open it in a text editor. This is the code for the Lambda function. You can use the following sample code for testing, or replace it with your own.

```
package main
import (
    "context"
    "fmt"
```

```
    "github.com/aws/aws-lambda-go/lambda"
)
type MyEvent struct {
    Name string `json:"name"`
}
func HandleRequest(_ context.Context, name MyEvent) (string, error) {
    return fmt.Sprintf("Hello %s!", name.Name), nil
}
func main() {
    lambda.Start(HandleRequest)
}
```

5. Use a text editor to create a Dockerfile in your project directory. The following example Dockerfile uses a [multi-stage build](#). This allows you to use a different base image in each step. You can use one image, such as a [Go base image](#), to compile your code and build the executable binary. You can then use a different image, such as provided.al2, in the final FROM statement to define the image that you deploy to Lambda. The build process is separated from the final deployment image, so the final image only contains the files needed to run the application.

Example — Multi-stage build Dockerfile

Note

Make sure that the version of Go that you specify in your Dockerfile (for example, golang:1.20) is the same version of Go that you used to create your application.

```
FROM golang:1.20 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
# build
COPY hello.go .
RUN go build -o main hello.go
# copy artifacts to a clean image
FROM public.ecr.aws/lambda/provided:al2
COPY --from=build /helloworld/main /main
ENTRYPOINT [ "/main" ]
```

6. Build your Docker image with the docker build command. The following example names the image docker-image.

```
docker build -t docker-image:test .
```

7. [Deploy the image to Amazon ECR and create the Lambda function \(p. 468\)](#).

Create a Go image from an alternative base image

You can build a container image for Go from an alternative base image. The following example Dockerfile uses [alpine](#) as the base image.

Example Dockerfile with alpine base image

Note

Make sure that the version of Go that you specify in your Dockerfile (for example, golang:1.20.2) is the same version of Go that you used to create your application.

```
FROM golang:1.20.2-alpine3.16 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
```

```
# Build
COPY hello.go .
RUN go build -o main hello.go
# Copy artifacts to a clean image
FROM alpine:3.16
COPY --from=build /helloworld/main /main
ENTRYPOINT [ "/main" ]
```

The steps are the same as [the steps for a provided.al2 base image \(p. 469\)](#), with one additional consideration: If you want to add the RIE to your image, you need to follow these additional steps before you run the `docker build` command. For more information about testing your image locally with the RIE, see [the section called "Testing images" \(p. 884\)](#).

To add RIE to the image

1. In your Dockerfile, replace the `ENTRYPOINT` instruction with the following content:

```
# (Optional) Add Lambda Runtime Interface Emulator and use a script in the ENTRYPOINT
# for simpler local runs
ADD https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie /usr/bin/aws-lambda-rie
RUN chmod 755 /usr/bin/aws-lambda-rie
COPY entry.sh /
RUN chmod 755 /entry.sh
ENTRYPOINT [ "/entry.sh" ]
```

2. Use a text editor to create file `entry.sh` in your project directory, containing the following content:

```
#!/bin/sh
if [ -z "${AWS_LAMBDA_RUNTIME_API}" ]; then
    exec /usr/bin/aws-lambda-rie "$@"
else
    exec "$@"
fi
```

If you do not want to add the RIE to your image, you can test your image locally without adding RIE to the image.

To test locally without adding RIE to the image

1. From your project directory, run the following command to download the RIE from GitHub and install it on your local machine.

```
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
```

2. Run your Lambda image function using the `docker run` command. In the following example, `/main` is the path to the function entry point.

```
docker run -d -v ~/.aws-lambda-rie:/aws-lambda --entrypoint /aws-lambda/aws-lambda-rie
-p 9000:8080 myfunction:latest /main
```

This runs the image as a container and starts up an endpoint locally at `localhost:9000/2015-03-31/functions/function/invocations`.

3. Post an event to the following endpoint using a `curl` command:

```
curl -XPOST "http://localhost:9000/2015-03-31/functions/function/invocations" -d
'{"Name": "World"}'
```

This command invokes the function running in the container image and returns a response.

4. [Deploy the image to Amazon ECR and create the Lambda function \(p. 468\).](#)

AWS Lambda function logging in Go

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

This page describes how to produce log output from your Lambda function's code, or access logs using the AWS Command Line Interface, the Lambda console, or the CloudWatch console.

Sections

- [Creating a function that returns logs \(p. 473\)](#)
- [Using the Lambda console \(p. 474\)](#)
- [Using the CloudWatch console \(p. 474\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 474\)](#)
- [Deleting logs \(p. 477\)](#)

Creating a function that returns logs

To output logs from your function code, you can use methods on [the fmt package](#), or any logging library that writes to `stdout` or `stderr`. The following example uses [the log package](#).

Example [main.go](#) – Logging

```
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 log format

```
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-9DV3XMP6XBC
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: 39 ms Memory Size: 128 MB Max Memory Used: 54 MB Init Duration: 203.69 ms
XRAY TraceId: 1-5e7d7595-212fxmpl9ee07c4884191322 SegmentId: 42ffxmpl0645f474 Sampled: true
```

The Go runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details.

Report Log

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY Traceld** – For traced requests, the [AWS X-Ray trace ID \(p. 807\)](#).
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

Using the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

Using the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the [Log groups page](#) on the CloudWatch console.
2. Choose the log group for your function (`/aws/lambda/your-function-name`).
3. Choose a log stream.

Each log stream corresponds to an [instance of your function \(p. 14\)](#). A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see [Error processor sample application for AWS Lambda \(p. 1015\)](#).

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

You can use the [AWS CLI](#) to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{  
    "StatusCode": 200,  
    "LogResult":  
        "U1RBU1QgUmVxdWVzdElk0iA4N2QwNDRi0C1mMTU0LTEzTgt0GNkYS0y0Tc0YzVlNGZiMjEgVmVyc21vb...",  
    "ExecutedVersion": "$LATEST"  
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \  
--query 'LogResult' --output text | base64 -d
```

You should see the following output:

```
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
```

The `base64` utility is available on Linux, macOS, and [Ubuntu on Windows](#). macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

```
#!/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 and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod +R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
  "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",
      "ingestionTime": 1559763018353
    }
  ],
  "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
  "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
```

}

Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.

AWS Lambda function errors in Go

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

This page describes how to view Lambda function invocation errors for the Go runtime using the Lambda console and the AWS CLI.

Sections

- [Creating a function that returns exceptions \(p. 478\)](#)
- [How it works \(p. 478\)](#)
- [Using the Lambda console \(p. 479\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 479\)](#)
- [Error handling in other AWS services \(p. 480\)](#)
- [What's next? \(p. 481\)](#)

Creating a function that returns exceptions

The following code sample demonstrates custom error handling that raises an exception directly from a Lambda function and handles it directly. Note that custom errors in Go must import the `errors` module.

```
package main

import (
    "errors"
    "github.com/aws/aws-lambda-go/lambda"
)

func OnlyErrors() error {
    return errors.New("something went wrong!")
}

func main() {
    lambda.Start(OnlyErrors)
}
```

Which returns the following:

```
{
    "errorMessage": "something went wrong!",
    "errorType": "errorString"
}
```

How it works

When you invoke a Lambda function, Lambda receives the invocation request and validates the permissions in your execution role, verifies that the event document is a valid JSON document, and checks parameter values.

If the request passes validation, Lambda sends the request to a function instance. The [Lambda runtime \(p. 37\)](#) environment converts the event document into an object, and passes it to your function handler.

If Lambda encounters an error, it returns an exception type, message, and HTTP status code that indicates the cause of the error. The client or service that invoked the Lambda function can handle the

error programmatically, or pass it along to an end user. The correct error handling behavior depends on the type of application, the audience, and the source of the error.

The following list describes the range of status codes you can receive from Lambda.

2xx

A 2xx series error with a X-Amz-Function-Error header in the response indicates a Lambda runtime or function error. A 2xx series status code indicates that Lambda accepted the request, but instead of an error code, Lambda indicates the error by including the X-Amz-Function-Error header in the response.

4xx

A 4xx series error indicates an error that the invoking client or service can fix by modifying the request, requesting permission, or by retrying the request. 4xx series errors other than 429 generally indicate an error with the request.

5xx

A 5xx series error indicates an issue with Lambda, or an issue with the function's configuration or resources. 5xx series errors can indicate a temporary condition that can be resolved without any action by the user. These issues can't be addressed by the invoking client or service, but a Lambda function's owner may be able to fix the issue.

For a complete list of invocation errors, see [InvokeFunction errors \(p. 1262\)](#).

Using the Lambda console

You can invoke your function on the Lambda console by configuring a test event and viewing the output. The output is captured in the function's execution logs and, when [active tracing \(p. 807\)](#) is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to test, and choose **Test**.
3. Under **Test event**, select **New event**.
4. Select a **Template**.
5. For **Name**, enter a name for the test. In the text entry box, enter the JSON test event.
6. Choose **Save changes**.
7. Choose **Test**.

The Lambda console invokes your function [synchronously \(p. 120\)](#) and displays the result. To see the response, logs, and other information, expand the **Details** section.

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

When you invoke a Lambda function in the AWS CLI, the AWS CLI splits the response into two documents. The AWS CLI response is displayed in your command prompt. If an error has occurred, the

response contains a `FunctionError` field. The invocation response or error returned by the function is written to an output file. For example, `output.json` or `output.txt`.

The following [invoke](#) command example demonstrates how to invoke a function and write the invocation response to an `output.txt` file.

```
aws lambda invoke \
--function-name my-function \
--cli-binary-format raw-in-base64-out \
--payload '{"key1": "value1", "key2": "value2", "key3": "value3"}' output.txt
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

You should see the AWS CLI response in your command prompt:

```
{  
  "StatusCode": 200,  
  "FunctionError": "Unhandled",  
  "ExecutedVersion": "$LATEST"  
}
```

You should see the function invocation response in the `output.txt` file. In the same command prompt, you can also view the output in your command prompt using:

```
cat output.txt
```

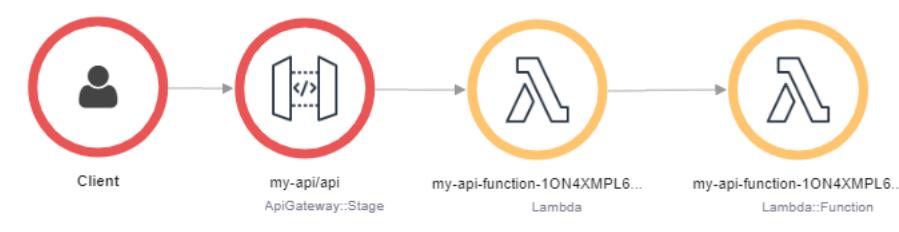
You should see the invocation response in your command prompt.

Error handling in other AWS services

When another AWS service invokes your function, the service chooses the invocation type and retry behavior. AWS services can invoke your function on a schedule, in response to a lifecycle event on a resource, or to serve a request from a user. Some services invoke functions asynchronously and let Lambda handle errors, while others retry or pass errors back to the user.

For example, API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502 error code. To customize the error response, you must catch errors in your code and format a response in the required format.

We recommend using AWS X-Ray to determine the source of an error and its cause. X-Ray allows you to find out which component encountered an error, and see details about the errors. The following example shows a function error that resulted in a 502 response from API Gateway.



For more information, see [Instrumenting Go code in AWS Lambda \(p. 482\)](#).

What's next?

- Learn how to show logging events for your Lambda function on the [the section called "Logging" \(p. 473\)](#) page.

Instrumenting Go code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of two SDK libraries:

- [AWS Distro for OpenTelemetry \(ADOT\)](#) – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- [AWS X-Ray SDK for Go](#) – An SDK for generating and sending trace data to X-Ray.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

Important

The X-Ray and AWS Lambda Powertools SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see [Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs](#).

Sections

- [Using ADOT to instrument your Go functions \(p. 482\)](#)
- [Using the X-Ray SDK to instrument your Go functions \(p. 482\)](#)
- [Activating tracing with the Lambda console \(p. 483\)](#)
- [Activating tracing with the Lambda API \(p. 483\)](#)
- [Activating tracing with AWS CloudFormation \(p. 483\)](#)
- [Interpreting an X-Ray trace \(p. 484\)](#)

Using ADOT to instrument your Go functions

ADOT provides fully managed Lambda [layers \(p. 11\)](#) that package everything you need to collect telemetry data using the OTel SDK. By consuming this layer, you can instrument your Lambda functions without having to modify any function code. You can also configure your layer to do custom initialization of OTel. For more information, see [Custom configuration for the ADOT Collector on Lambda](#) in the ADOT documentation.

For Go runtimes, you can add the **AWS managed Lambda layer for ADOT Go** to automatically instrument your functions. For detailed instructions on how to add this layer, see [AWS Distro for OpenTelemetry Lambda Support for Go](#) in the ADOT documentation.

Using the X-Ray SDK to instrument your Go functions

To record details about calls that your Lambda function makes to other resources in your application, you can also use the AWS X-Ray SDK for Go. To get the SDK, download the SDK from its [GitHub repository](#) with go get:

```
go get github.com/aws/aws-xray-sdk-go
```

To instrument AWS SDK clients, pass the client to the `xray.AWS()` method. You can then trace calls by using the `WithContext` version of the method.

```
svc := s3.New(session.New())
xray.AWS(svc.Client)
...
svc.ListBucketsWithContext(ctx aws.Context, input *ListBucketsInput)
```

After you add the correct dependencies and make the necessary code changes, activate tracing in your function's configuration via the Lambda console or the API.

Activating tracing with the Lambda console

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring and operations tools**.
4. Choose **Edit**.
5. Under **X-Ray**, toggle on **Active tracing**.
6. Choose **Save**.

Activating tracing with the Lambda API

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- [UpdateFunctionConfiguration \(p. 1377\)](#)
- [GetFunctionConfiguration \(p. 1229\)](#)
- [CreateFunction \(p. 1165\)](#)

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Activating tracing with AWS CloudFormation

To activate tracing on an `AWS::Lambda::Function` resource in an AWS CloudFormation template, use the `TracingConfig` property.

Example [function-inline.yml – Tracing configuration](#)

```
Resources:
```

```
function:
  Type: AWS::Lambda::Function
  Properties:
    TracingConfig:
      Mode: Active
    ...
  ...
```

For an AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function resource, use the Tracing property.

Example [template.yml](#) – Tracing configuration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
    ...
  ...
```

Interpreting an X-Ray trace

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's [execution role \(p. 816\)](#). Otherwise, add the [AWSXRayDaemonWriteAccess](#) policy to the execution role.

After you've configured active tracing, you can observe specific requests through your application. The [X-Ray service graph](#) shows information about your application and all its components. The following example from the [error processor \(p. 1015\)](#) sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.

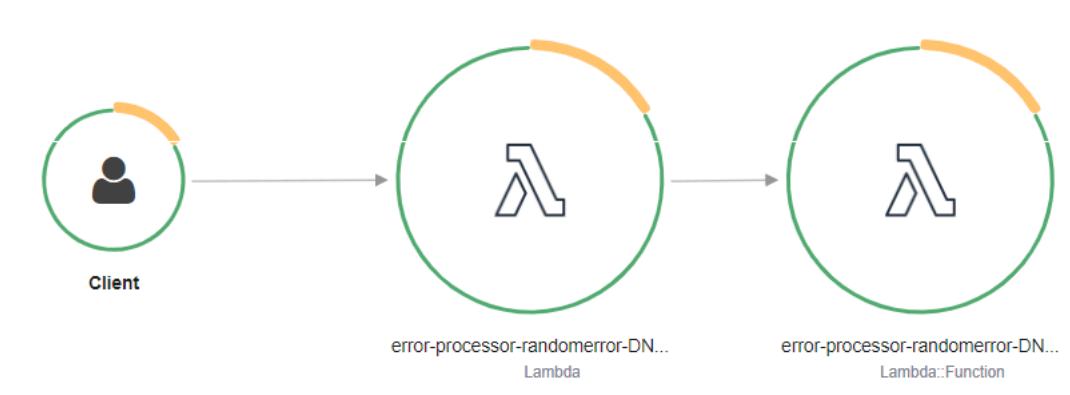


X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

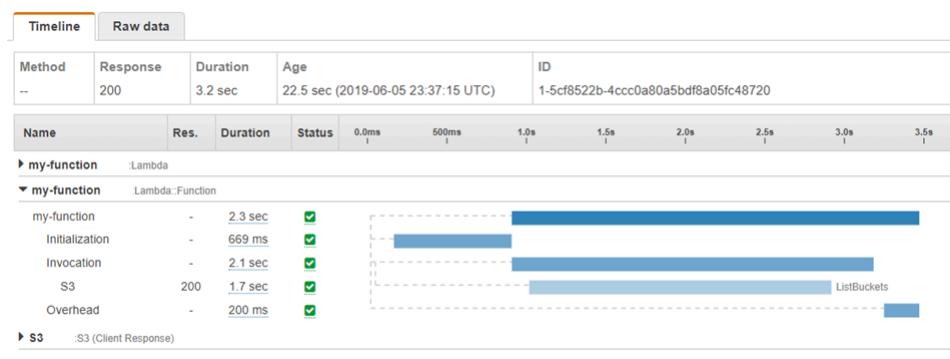
Note

You cannot configure the X-Ray sampling rate for your functions.

When using active tracing, Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes for the primary function from the [error processor sample application \(p. 1015\)](#).



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of AWS : :Lambda and the other has origin AWS : :Lambda : :Function.



This example expands the function segment to show its three subsegments:

- **Initialization** – Represents time spent loading your function and running [initialization code \(p. 13\)](#). This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see the [AWS X-Ray SDK for Go](#) in the [AWS X-Ray Developer Guide](#).

Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see [AWS X-Ray pricing](#).

Using environment variables

To access [environment variables \(p. 76\)](#) in Go, use the [Getenv](#) function.

The following explains how to do this. Note that the function imports the [fmt](#) package to format the printed results and the [os](#) package, a platform-independent system interface that allows you to access environment variables.

```
package main

import (
    "fmt"
    "os"
    "github.com/aws/aws-lambda-go/lambda"
)

func main() {
    fmt.Printf("%s is %s. years old\n", os.Getenv("NAME"), os.Getenv("AGE"))
}
```

For a list of environment variables that are set by the Lambda runtime, see [Defined runtime environment variables \(p. 79\)](#).

Building Lambda functions with C#

The following sections explain how common programming patterns and core concepts apply when authoring Lambda function code in C#.

AWS Lambda provides the following libraries for C# functions:

- **Amazon.Lambda.Core** – This library provides a static Lambda logger, serialization interfaces and a context object. The Context object ([AWS Lambda context object in C# \(p. 505\)](#)) provides runtime information about your Lambda function.
- **Amazon.Lambda.Serialization.Json** – This is an implementation of the serialization interface in **Amazon.Lambda.Core**.
- **Amazon.Lambda.Logging.AspNetCore** – This provides a library for logging from ASP.NET.
- Event objects (POCOs) for several AWS services, including:
 - **Amazon.Lambda.APIGatewayEvents**
 - **Amazon.Lambda.CognitoEvents**
 - **Amazon.Lambda.ConfigEvents**
 - **Amazon.Lambda.DynamoDBEvents**
 - **Amazon.Lambda.KinesisEvents**
 - **Amazon.Lambda.S3Events**
 - **Amazon.Lambda.SQSEvents**
 - **Amazon.Lambda.SNSEvents**

These packages are available at [Nuget packages](#).

.NET

Name	Identifier	Operating system	Architectures	Deprecation (Phase 1)
.NET Core 3.1	dotnetcore3.1	Amazon Linux 2	x86_64, arm64	Apr 3, 2023
.NET 7	dotnet7	Amazon Linux 2	x86_64, arm64	
.NET 6	dotnet6	Amazon Linux 2	x86_64, arm64	
.NET 5	dotnet5.0	Amazon Linux 2	x86_64	

Note

For end of support information about .NET Core 2.1, see [the section called “Runtime deprecation policy” \(p. 39\)](#).

To get started with application development in your local environment, deploy one of the sample applications available in this guide's GitHub repository.

Sample Lambda applications in C#

- [blank-csharp](#) – A C# function that shows the use of Lambda's .NET libraries, logging, environment variables, AWS X-Ray tracing, unit tests, and the AWS SDK.
- [ec2-spot](#) – A function that manages spot instance requests in Amazon EC2.

Topics

- [Lambda function handler in C# \(p. 489\)](#)
- [Deploy C# Lambda functions with .zip file archives \(p. 497\)](#)
- [Deploy .NET Lambda functions with container images \(p. 502\)](#)
- [AWS Lambda context object in C# \(p. 505\)](#)
- [Lambda function logging in C# \(p. 506\)](#)
- [AWS Lambda function errors in C# \(p. 514\)](#)
- [Instrumenting C# code in AWS Lambda \(p. 519\)](#)
- [.NET functions with native AOT compilation \(p. 525\)](#)

Lambda function handler in C#

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. When the handler exits or returns a response, it becomes available to handle another event.

You define a Lambda function handler as an instance or static method in a class. For access to the Lambda context object, you can define a method parameter of type *ILambdaContext*. You can use this to access information about the current invocation, such as the name of the function, memory limit, remaining execution time, and logging.

```
returnType handler-name(inputType input, ILambdaContext context) {  
    ...  
}
```

In the syntax, note the following:

- *inputType* – The first handler parameter is the input to the handler. This can be event data (that an event source publishes) or custom input that you provide, such as a string or any custom data object.
- *returnType* – If you plan to invoke the Lambda function synchronously (using the RequestResponse invocation type), you can return the output of your function using any of the supported data types. For example, if you use a Lambda function as a mobile application backend, you are invoking it synchronously. Your output data type is serialized into JSON.

If you plan to invoke the Lambda function asynchronously (using the Event invocation type), the *returnType* should be *void*. For example, if you use Lambda with event sources such as Amazon Simple Storage Service (Amazon S3) or Amazon Simple Notification Service (Amazon SNS), these event sources invoke the Lambda function using the Event invocation type.

- *ILambdaContext context* – The second argument in the handler signature is optional. It provides access to the [context object \(p. 505\)](#), which has information about the function and request.

Handling streams

By default, Lambda supports only the *System.IO.Stream* type as an input parameter.

For example, consider the following C# example code.

```
using System.IO;  
  
namespace Example  
{  
    public class Hello  
    {  
        public Stream MyHandler(Stream stream)  
        {  
            //function logic  
        }  
    }  
}
```

In the example C# code, the first handler parameter is the input to the handler (*MyHandler*). This can be event data (published by an event source such as Amazon S3) or custom input that you provide, such as a *Stream* (as in this example) or any custom data object. The output is of type *Stream*.

Handling standard data types

All the following other types require you to specify a serializer:

- Primitive .NET types (such as string or int)
- Collections and maps – IList, IEnumerable, IList<T>, Array, IDictionary, IDictionary<TKey, TValue>
- POCO types (Plain old CLR objects)
- Predefined AWS event types
- For asynchronous invocations, Lambda ignores the return type. In such cases, the return type may be set to void.
- If you are using .NET asynchronous programming, the return type can be Task and Task<T> types and use `async` and `await` keywords. For more information, see [Using async in C# functions with Lambda \(p. 495\)](#).

Unless your function input and output parameters are of type `System.IO.Stream`, you must serialize them. Lambda provides default serializers that can be applied at the assembly or method level of your application, or you can define your own by implementing the `ILambdaSerializer` interface provided by the `Amazon.Lambda.Core` library. For more information, see [Deploy C# Lambda functions with .zip file archives \(p. 497\)](#).

To add the default serializer attribute to a method, first add a dependency on `Amazon.Lambda.Serialization.SystemTextJson` in your `.csproj` file.

```
<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <TargetFramework>net6.0</TargetFramework>
        <ImplicitUsings>enable</ImplicitUsings>
        <Nullable>enable</Nullable>
        <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
        <AWSProjectType>Lambda</AWSProjectType>
        <!-- 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="2.1.0 " />
        <PackageReference Include="Amazon.Lambda.Serialization.SystemTextJson"
Version="2.2.0" />
    </ItemGroup>
</Project>
```

The example below illustrates the flexibility you can leverage by specifying the default `System.Text.Json` serializer on one method and another of your choosing on a different method:

```
public class ProductService
{
    [LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer))]
    public Product DescribeProduct(DescribeProductRequest request)
    {
        return catalogService.DescribeProduct(request.Id);
    }
}
```

```
[LambdaSerializer(typeof(MyJsonSerializer))]
public Customer DescribeCustomer(DescribeCustomerRequest request)
{
    return customerService.DescribeCustomer(request.Id);
}
```

Source generation for JSON serialization

C# 9 provides source generators that can generate code during compilation. Starting with .NET 6, the native JSON library `System.Text.Json` can use source generators to parse JSON without the need for reflection APIs. This can help improve cold start performance.

To use the source generator

1. In your project, define an empty, partial class that derives from `System.Text.Json.Serialization.JsonSerializerContext`.
2. Add the `JsonSerializable` attribute for each .NET type that the source generator must generate serialization code for.

Example API Gateway integration leveraging source generation

```
using System.Collections.Generic;
using System.Net;
using System.Text.Json.Serialization;

using Amazon.Lambda.Core;
using Amazon.Lambda.APIGatewayEvents;
using Amazon.Lambda.Serialization.SystemTextJson;

[assembly:
LambdaSerializer(typeof(SourceGeneratorLambdaJsonSerializer<SourceGeneratorExample.HttpApiJsonSerializerContext>))]

namespace SourceGeneratorExample;

[JsonSerializable(typeof(APIGatewayHttpApiV2ProxyRequest))]
[JsonSerializable(typeof(APIGatewayHttpApiV2ProxyResponse))]
public partial class HttpApiJsonSerializerContext : JsonSerializerContext
{

public class Functions
{
    public APIGatewayProxyResponse Get(APIGatewayHttpApiV2ProxyRequest
request, ILambdaContext context)
    {
        context.Logger.LogInformation("Get Request");
        var response = new APIGatewayHttpApiV2ProxyResponse
        {
            StatusCode = (int) HttpStatusCode.OK,
            Body = "Hello AWS Serverless",
            Headers = new Dictionary<string, string> { { "Content-Type",
"text/plain" } }
        };
        return response;
    }
}
```

When you invoke your function, Lambda uses the source-generated JSON serialization code to handle the serialization of Lambda events and responses.

Handler signatures

When creating Lambda functions, you have to provide a handler string that tells Lambda where to look for the code to invoke. In C#, the format is:

ASSEMBLY::TYPE::METHOD where:

- **ASSEMBLY** is the name of the .NET assembly file for your application. When using the .NET Core CLI to build your application, if you haven't set the assembly name using the AssemblyName property in the .csproj file, the **ASSEMBLY** name is the .csproj file name. For more information, see [.NET Core CLI \(p. 497\)](#). In this case, let's assume that the .csproj file is HelloWorldApp.csproj.
- **TYPE** is the full name of the handler type, which consists of the **Namespace** and the **ClassName**. In this case Example.Hello.
- **METHOD** is name of the function handler, in this case MyHandler.

Ultimately, the signature is of this format: **Assembly::Namespace.ClassName::MethodName**

Consider the following example:

```
using System.IO;

namespace Example
{
    public class Hello
    {
        public Stream MyHandler(Stream stream)
        {
            //function logic
        }
    }
}
```

The handler string would be: HelloWorldApp::Example.Hello::MyHandler

Important

If the method specified in your handler string is overloaded, you must provide the exact signature of the method that Lambda should invoke. If the resolution would require selecting among multiple (overloaded) signatures, Lambda will reject an otherwise valid signature.

Using top-level statements

In .NET 6, you can write functions using C# 9's [top-level statements](#). Top-level statements remove some of the initial boilerplate code for .NET projects, reducing the number of lines of code that you write.

Top-level statements require the build output to be an executable. To configure this setting, specify Exe as the output type in your .csproj file:

Example .csproj file

```
<PropertyGroup>
<TargetFramework>net6.0</TargetFramework>
<OutputType>Exe</OutputType>
...
</PropertyGroup>
```

Here's an example of how you can rewrite the [previous function code example \(p. 492\)](#) using top-level statements:

Example – Using top-level statements

```
using Amazon.Lambda.RuntimeSupport;

var handler = (Stream stream) =>
{
    //function logic
};

await LambdaBootstrapBuilder.Create(handler).Build().RunAsync();
```

When using top-level statements, you only include the ASSEMBLY name when providing the handler signature. Continuing from the [previous example \(p. 492\)](#), the handler string would be HelloWorldApp.

By setting the handler to the assembly name Lambda will treat the assembly as an executable and execute it at startup. You must add the NuGet package Amazon.Lambda.RuntimeSupport to the project so that the executable that runs at startup starts the Lambda runtime client.

For more information about using top-level statements, see [Introducing the .NET 6 runtime for AWS Lambda](#) on the AWS Compute Blog.

Using Lambda Annotations (Preview)

Note

The Lambda Annotations feature is in preview release and is subject to change.

The Lambda Annotations framework can simplify the process of writing handler code, updating CloudFormation templates, and configuring dependency injection for .NET 6 Lambda functions. When you use Lambda Annotations, you must deploy with CloudFormation.

The following examples demonstrate the benefits of using Lambda Annotations. This function adds two numbers.

Example – Without Lambda Annotations

```
public class Functions
{
    public APIGatewayProxyResponse LambdaMathPlus(APIGatewayProxyRequest request,
ILambdaContext context)
    {
        if (!request.PathParameters.TryGetValue("x", out var xs))
        {
            return new APIGatewayProxyResponse
            {
                StatusCode = (int) HttpStatusCode.BadRequest
            };
        }
        if (!request.PathParameters.TryGetValue("y", out var ys))
        {
            return new APIGatewayProxyResponse
            {
                StatusCode = (int) HttpStatusCode.BadRequest
            };
        }

        var x = int.Parse(xs);
```

```

        var y = int.Parse(yS);
        return new APIGatewayProxyResponse
        {
            StatusCode = (int) HttpStatusCode.OK,
            Body = (x + y).ToString(),
            Headers = new Dictionary<string, string> { { "Content-Type", "text/plain" } }
        };
    }
}

```

Example – With Lambda Annotations

```

public class Functions
{
    [LambdaFunction]
    [RestApi("/plus/{x}/{y}")]
    public int Plus(int x, int y)
    {
        return x + y;
    }
}

```

For details about how to use the Lambda Annotations framework, see the following resources:

- The [aws/aws-lambda-dotnet](#) GitHub repository.
- The [preview blog](#) for Lambda Annotations.
- The [Amazon.Lambda.Annotations](#) NuGet package.

Serializing Lambda functions

For any Lambda functions that use input or output types other than a `Stream` object, you must add a serialization library to your application. You can do this in the following ways:

- Use the `Amazon.Lambda.Serialization.SystemTextJson` NuGet package. This library uses the native .NET Core JSON serializer to handle serialization. This package provides a performance improvement over `Amazon.Lambda.Serialization.Json`, but note the limitations described in the [Microsoft documentation](#). This library is available for .NET Core 3.1 and later runtimes.
- Use the `Amazon.Lambda.Serialization.Json` NuGet package. This library uses JSON.NET to handle serialization.
- Create your own serialization library by implementing the `ILambdaSerializer` interface, which is available as part of the `Amazon.Lambda.Core` library. The interface defines two methods:
 - `T Deserialize<T>(Stream requestStream);`

You implement this method to deserialize the request payload from the Invoke API into the object that is passed to the Lambda function handler.

- `T Serialize<T>(T response, Stream responseStream);`

You implement this method to serialize the result returned from the Lambda function handler into the response payload that the Invoke API operation returns.

To use the serializer, you must add a dependency to your `MyProject.csproj` file.

```

...
<ItemGroup>

```

```
<PackageReference Include="Amazon.Lambda.Serialization.SystemTextJson" Version="2.1.0" />
<!-- or -->
<PackageReference Include="Amazon.Lambda.Serialization.Json" Version="2.0.0" />
</ItemGroup>
```

Next, you must define the serializer. The following example defines the `Amazon.Lambda.Serialization.SystemTextJson` serializer in the `AssemblyInfo.cs` file.

```
[assembly: LambdaSerializer(typeof(DefaultLambdaJsonSerializer))]
```

The following example defines the `Amazon.Lambda.Serialization.Json` serializer in the `AssemblyInfo.cs` file.

```
[assembly: LambdaSerializer(typeof(JsonSerializer))]
```

You can define a custom serialization attribute at the method level, which overrides the default serializer specified at the assembly level.

```
public class ProductService{
    [LambdaSerializer(typeof(JsonSerializer))]
    public Product DescribeProduct(DescribeProductRequest request)
    {
        return catalogService.DescribeProduct(request.Id);
    }

    [LambdaSerializer(typeof(MyJsonSerializer))]
    public Customer DescribeCustomer(DescribeCustomerRequest request)
    {
        return customerService.DescribeCustomer(request.Id);
    }
}
```

Lambda function handler restrictions

Note that there are some restrictions on the handler signature.

- It may not be unsafe and use pointer types in the handler signature, though you can use unsafe context inside the handler method and its dependencies. For more information, see [unsafe \(C# Reference\)](#) on the Microsoft Docs website.
- It may not pass a variable number of parameters using the `params` keyword, or use `ArgIterator` as an input or a return parameter, which is used to support a variable number of parameters.
- The handler may not be a generic method, for example, `IList<T> Sort<T>(IList<T> input)`.
- Async handlers with signature `async void` are not supported.

Using `async` in C# functions with Lambda

If you know that your Lambda function will require a long-running process, such as uploading large files to Amazon S3 or reading a large stream of records from Amazon DynamoDB, you can take advantage of the `async/await` pattern. When you use this signature, Lambda invokes the function synchronously and waits for the function to return a response or for execution to time out.

```
public async Task<Response> ProcessS3ImageResizeAsync(SimpleS3Event input)
{
    var response = await client.DoAsyncWork(input);
    return response;
}
```

If you use this pattern, consider the following:

- Lambda does not support `async void` methods.
- If you create an `async` Lambda function without implementing the `await` operator, .NET will issue a compiler warning and you will observe unexpected behavior. For example, some `async` actions will run while others won't. Or some `async` actions won't complete before the function invocation completes.

```
public async Task ProcessS3ImageResizeAsync(SimpleS3Event event) // Compiler warning
{
    client.DoAsyncWork(input);
}
```

- Your Lambda function can include multiple `async` calls, which can be invoked in parallel. You can use the `Task.WhenAll` and `Task.WhenAny` methods to work with multiple tasks. To use the `Task.WhenAll` method, you pass a list of the operations as an array to the method. Note that in the following example, if you neglect to include any operation to the array, that call may return before its operation completes.

```
public async Task DoesNotWaitForAllTasks1()
{
    // In Lambda, Console.WriteLine goes to CloudWatch Logs.
    var task1 = Task.Run(() => Console.WriteLine("Test1"));
    var task2 = Task.Run(() => Console.WriteLine("Test2"));
    var task3 = Task.Run(() => Console.WriteLine("Test3"));

    // Lambda may return before printing "Test2" since we never wait on task2.
    await Task.WhenAll(task1, task3);
}
```

To use the `Task.WhenAny` method, you again pass a list of operations as an array to the method. The call returns as soon as the first operation completes, even if the others are still running.

```
public async Task DoesNotWaitForAllTasks2()
{
    // In Lambda, Console.WriteLine goes to CloudWatch Logs.
    var task1 = Task.Run(() => Console.WriteLine("Test1"));
    var task2 = Task.Run(() => Console.WriteLine("Test2"));
    var task3 = Task.Run(() => Console.WriteLine("Test3"));

    // Lambda may return before printing all tests since we're waiting for only one to
    // finish.
    await Task.WhenAny(task1, task2, task3);
}
```

Deploy C# Lambda functions with .zip file archives

A .NET Core deployment package (.zip file archive) contains your function's compiled assembly along with all of its assembly dependencies. The package also contains a `proj.deps.json` file. This signals to the .NET Core runtime all of your function's dependencies and a `proj.runtimeconfig.json` file, which is used to configure the runtime. The .NET command line interface (CLI) publish command can create a folder with all of these files. By default, the `proj.runtimeconfig.json` is not included because a Lambda project is typically configured to be a class library. To force the `proj.runtimeconfig.json` to be written as part of the publish process, pass in the command line argument `/p:GenerateRuntimeConfigurationFiles=true` to the publish command.

Although it is possible to create the deployment package with the `dotnet publish` command, we recommend that you create the deployment package with either the [.NET Core CLI \(p. 497\)](#) or the [AWS Toolkit for Visual Studio \(p. 500\)](#). These are tools optimized specifically for Lambda to ensure that the `Lambda-project.runtimeconfig.json` file exists and optimizes the package bundle, including the removal of any non-Linux-based dependencies.

Topics

- [.NET Core CLI \(p. 497\)](#)
- [AWS Toolkit for Visual Studio \(p. 500\)](#)

.NET Core CLI

The .NET Core CLI offers a cross-platform way for you to create .NET-based Lambda applications. This section assumes that you have installed the .NET Core CLI. If you haven't, see [Download .NET](#) on the Microsoft website.

In the .NET CLI, you use the new command to create .NET projects from a command line. This is useful if you want to create a project outside of Visual Studio. To view a list of the available project types, open a command line, navigate to where you installed the .NET Core runtime, and then run the following command:

```
dotnet new -all
Usage: new [options]
...
Templates                                     Short Name      Language      Tags
-----
Console Application                           console        [C#], F#, VB
Common/Console                               classlib       [C#], F#, VB
Class library                                classlib       [C#], F#, VB
Common/Library                               mstest         [C#], F#, VB
Unit Test Project                           mstest         [C#], F#, VB
Test/MSTest                                 xunit          [C#], F#, VB
xUnit Test Project                          xunit          [C#], F#, VB
Test/xUnit
...
Examples:
  dotnet new mvc --auth Individual
  dotnet new viewstart
  dotnet new --help
```

Lambda offers additional templates via the [Amazon.Lambda.Templates](#) NuGet package. To install this package, run the following command:

```
dotnet new -i Amazon.Lambda.Templates
```

Once the install is complete, the Lambda templates show up as part of `dotnet new`. To examine details about a template, use the `help` option.

```
dotnet new lambda.EmptyFunction --help
```

The `lambda.EmptyFunction` template supports the following options:

- `--name` – The name of the function
- `--profile` – The name of a profile in your [AWS SDK for .NET credentials file](#)
- `--region` – The AWS Region to create the function in

These options are saved to a file named `aws-lambda-tools-defaults.json`.

Create a function project with the `lambda.EmptyFunction` template.

```
dotnet new lambda.EmptyFunction --name MyFunction
```

Under the `src/myfunction` directory, examine the following files:

- **aws-lambda-tools-defaults.json**: This is where you specify the command line options when deploying your Lambda function. For example:

```
"profile" : "default",
"region" : "us-east-2",
"configuration" : "Release",
"function-runtime": "dotnet6",
"function-memory-size" : 256,
"function-timeout" : 30,
"function-handler" : "MyFunction::MyFunction.Function::FunctionHandler"
```

- **Function.cs**: Your Lambda handler function code. It's a C# template that includes the default `Amazon.Lambda.Core` library and a default `LambdaSerializer` attribute. For more information on serialization requirements and options, see [Serializing Lambda functions \(p. 494\)](#). It also includes a sample function that you can edit to apply your Lambda function code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

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.DefaultLambdaJsonSerializer))]

namespace MyFunction
{
    public class Function
    {

        public string FunctionHandler(string input, ILambdaContext context)
        {
            return input.ToUpper();
        }
    }
}
```

- **MyFunction.csproj:** An [MSBuild](#) file that lists the files and assemblies that comprise your application.

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
    <!-- 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="2.1.0 " />
    <PackageReference Include="Amazon.Lambda.Serialization.SystemTextJson"
Version="2.2.0" />
</ItemGroup>

</Project>
```

- **Readme:** Use this file to document your Lambda function.

Under the myfunction/test directory, examine the following files:

- **myFunction.Tests.csproj:** As noted previously, this is an [MSBuild](#) file that lists the files and assemblies that comprise your test project. Note also that it includes the Amazon.Lambda.Core library, so you can seamlessly integrate any Lambda templates required to test your function.

```
<Project Sdk="Microsoft.NET.Sdk">
    ...
    <PackageReference Include="Amazon.Lambda.Core" Version="2.1.0 " />
    ...
```

- **FunctionTest.cs:** The same C# code template file that it is included in the src directory. Edit this file to mirror your function's production code and test it before uploading your Lambda function to a production environment.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

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);  
    }  
}  
}
```

Once your function has passed its tests, you can build and deploy using the Amazon.Lambda.Tools [.NET Core Global Tool](#). To install the .NET Core Global Tool, run the following command:

```
dotnet tool install -g Amazon.Lambda.Tools
```

If you already have the tool installed, you can make sure that it is the latest version using the following command:

```
dotnet tool update -g Amazon.Lambda.Tools
```

For more information about the Amazon.Lambda.Tools .NET Core Global Tool, see the [AWS Extensions for .NET CLI](#) repository on GitHub.

With the Amazon.Lambda.Tools installed, you can deploy your function using the following command:

```
dotnet lambda deploy-function MyFunction --function-role role
```

After deployment, you can re-test it in a production environment using the following command, and pass in a different value to your Lambda function handler:

```
dotnet lambda invoke-function MyFunction --payload "Just Checking If Everything is OK"
```

If everything is successful, you see the following:

```
dotnet lambda invoke-function MyFunction --payload "Just Checking If Everything is OK"  
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
```

AWS Toolkit for Visual Studio

You can build .NET-based Lambda applications using the Lambda plugin for the [AWS Toolkit for Visual Studio](#). The toolkit is available as a [Visual Studio extension](#).

1. Launch Microsoft Visual Studio and choose **New project**.
 - a. From the **File** menu, choose **New**, and then choose **Project**.
 - b. In the **New Project** window, choose **Lambda Project (.NET Core)**, and then choose **OK**.
 - c. In the **Select Blueprint** window, select from the list of sample applications with sample code to help you get started with creating a .NET-based Lambda application.

- d. To create a Lambda application from scratch, choose **Empty Function**, and then choose **Finish**.
2. Review the `aws-lambda-tools-defaults.json` file, which is created as part of your project. You can set the options in this file, which the Lambda tooling reads by default. The project templates created in Visual Studio set many of these fields with default values. Note the following fields:
- **profile** – The name of a profile in your [AWS SDK for .NET credentials file](#)
 - **function-handler** – The field where you specify the function handler. (This is why you don't have to set it in the wizard.) However, whenever you rename the *Assembly*, *Namespace*, *Class*, or *Function* in your function code, you must update the corresponding fields in the `aws-lambda-tools-defaults.json` file.

```
{  
    "profile": "default",  
    "region": "us-east-2",  
    "configuration": "Release",  
    "function-runtime": "dotnet6",  
    "function-memory-size": 256,  
    "function-timeout": 30,  
    "function-handler": "Assembly::Namespace.Class::Function"  
}
```

3. Open the **Function.cs** file. You are provided with a template to implement your Lambda function handler code.
4. After writing the code that represents your Lambda function, upload it by opening the context (right-click) menu for the **Project** node in your application and then choosing **Publish to AWS Lambda**.
5. In the **Upload Lambda Function** window, enter a name for the function, or select a previously published function to republish. Then choose **Next**.
6. In the **Advanced Function Details** window, configure the following options:
- **Role Name** (required) – The [AWS Identity and Access Management \(IAM\) role \(p. 816\)](#) that Lambda assumes when it runs your function.
 - **Environment** – Key-value pairs that Lambda sets in the execution environment. To extend your function's configuration outside of code, [use environment variables \(p. 76\)](#).
 - **Memory** – The amount of memory available to the function at runtime. Choose an amount [between 128 MB and 10,240 MB \(p. 1131\)](#) in 1-MB increments.
 - **Timeout** – The amount of time that Lambda allows a function to run before stopping it. The default is three seconds. The maximum allowed value is 900 seconds.
 - **VPC** – If your function needs network access to resources that are not available over the internet, [configure it to connect to a virtual private cloud \(VPC\) \(p. 222\)](#).
 - **DLQ** – If your function is invoked asynchronously, [choose a dead-letter queue \(p. 129\)](#) to receive failed invocations.
 - **Enable active tracing** – Sample incoming requests and [trace sampled requests with AWS X-Ray \(p. 807\)](#).
7. Choose **Next**, and then choose **Upload** to deploy your application.

For more information, see [Deploying an AWS Lambda Project with the .NET Core CLI](#).

Deploy .NET Lambda functions with container images

You can deploy your Lambda function code as a [container image \(p. 881\)](#). To help you build a container image for your .NET function, AWS provides the following resources:

- AWS base images for Lambda

These base images are preloaded with a language runtime and other components that are required to run the image on Lambda. AWS provides a Dockerfile for each of the base images to help with building your container image.

- Open-source runtime interface clients

If you use a community or private enterprise base image, you must add a [runtime interface client \(p. 883\)](#) to the base image to make it compatible with Lambda.

- Open-source runtime interface emulator

Lambda provides a runtime interface emulator (RIE) for you to test your function locally. The base images for Lambda and base images for custom .runtimes include the RIE. For other base images, you can download the RIE for [testing your image \(p. 884\)](#) locally.

The workflow for a function defined as a container image includes these steps:

1. Build your container image using the resources listed in this topic.
2. Upload the image to your [Amazon Elastic Container Registry \(Amazon ECR\) container registry \(p. 891\)](#).
3. [Create the function \(p. 113\)](#) or [update the function code \(p. 115\)](#) to deploy the image to an existing function.

Topics

- [AWS base images for .NET \(p. 502\)](#)
- [Using a .NET base image \(p. 503\)](#)
- [.NET runtime interface clients \(p. 504\)](#)

AWS base images for .NET

AWS provides the following base images for .NET:

Tags	Runtime	Operating system	Dockerfile	Deprecation
core3.1	.NET Core 3.1	Amazon Linux 2	Dockerfile for .NET Core 3.1 on GitHub	Apr 3, 2023
7	.NET 7	Amazon Linux 2	Dockerfile for .NET 7 on GitHub	
6	.NET 6	Amazon Linux 2	Dockerfile for .NET 6 on GitHub	

Tags	Runtime	Operating system	Dockerfile	Deprecation
5.0	.NET 5	Amazon Linux 2	Dockerfile for .NET 5 on GitHub	

Amazon ECR repository: [gallery.ecr.aws/lambda/dotnet](#)

Using a .NET base image

Prerequisites

To complete the steps in this section, you must have the following:

- [.NET SDK](#) – The following steps use the .NET 7 base image. Make sure that your .NET version matches the version of the [base image](#) that you specify in your Dockerfile.
- [Docker](#)

Creating and deploying an image using a base image

In the following steps, you use [Amazon.Lambda.Templates](#) and [Amazon.Lambda.Tools](#) to create a .NET project. Then, you build a Docker image, upload the image to Amazon ECR, and deploy it to a Lambda function.

1. Install the Amazon.Lambda.Templates NuGet package.

```
dotnet new install Amazon.Lambda.Templates
```

For more information about this package, see the [AWS Lambda for .NET Core](#) repository on GitHub.

2. Create a .NET project using the `lambda.image.EmptyFunction` template.

```
dotnet new lambda.image.EmptyFunction --name MyFunction --region us-east-1
```

3. In the `src/MyFunction` directory, examine the following files:

- **aws-lambda-tools-defaults.json** – This file is where you specify the command line options when deploying your Lambda function.
- **Function.cs** – Your Lambda handler function code. This is a C# template that includes the default `Amazon.Lambda.Core` library and a default `LambdaSerializer` attribute. For more information about serialization requirements and options, see [Serializing Lambda functions \(p. 494\)](#). You can use the provided code for testing, or replace it with your own.
- **MyFunction.csproj** – A .NET [project file](#), which lists the files and assemblies that comprise your application.
- **Readme** – This file contains more information about the sample Lambda function.

4. Examine the Dockerfile in the `src/MyFunction` directory. You can use the provided Dockerfile for testing, or replace it with your own. If you use your own, make sure to:

- Set the `FROM` property to the [URI of the base image](#). Your .NET version must match the version of the base image.
- Set the `CMD` argument to the Lambda function handler. This should match the `image-command` in `aws-lambda-tools-defaults.json`.

Example Dockerfile

```
FROM public.ecr.aws/lambda/dotnet:7
#You can also pull these images from DockerHub amazon/aws-lambda-dotnet:7

# Copy function code
COPY publish/* ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside of
# the Dockerfile)
CMD [ "MyProject::MyFunction.FunctionHandler" ]
```

5. Install the Amazon.Lambda.Tools [.NET Core Global Tool](#).

```
dotnet tool install -g Amazon.Lambda.Tools
```

If Amazon.Lambda.Tools is already installed, make sure that you have the latest version.

```
dotnet tool update -g Amazon.Lambda.Tools
```

6. Change the directory to `MyFunction/src/MyFunction`, if you're not there already.

```
cd src/MyFunction
```

7. Use Amazon.Lambda.Tools to build the Docker image, push it to a new Amazon ECR repository, and deploy the Lambda function.

For `--function-role`, specify the role name—not the Amazon Resource Name (ARN)—of the [execution role \(p. 816\)](#) for the function. For example, `lambda-role`.

```
dotnet lambda deploy-function MyFunction --function-role lambda-role
```

For more information about the Amazon.Lambda.Tools .NET Core Global Tool, see the [AWS Extensions for .NET CLI](#) repository on GitHub.

8. Invoke the function.

```
dotnet lambda invoke-function MyFunction --payload "Testing the function"
```

If everything is successful, you see the following:

```
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
```

.NET runtime interface clients

You can download the .NET runtime interface client from the [AWS Lambda for .NET Core](#) repository on GitHub.

AWS Lambda context object in C#

When Lambda runs your function, it passes a context object to the [handler \(p. 489\)](#). This object provides properties with information about the invocation, function, and execution environment.

Context properties

- `FunctionName` – The name of the Lambda function.
- `FunctionVersion` – The [version \(p. 83\)](#) of the function.
- `InvokedFunctionArn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `MemoryLimitInMB` – The amount of memory that's allocated for the function.
- `AwsRequestId` – The identifier of the invocation request.
- `LogGroupName` – The log group for the function.
- `LogStreamName` – The log stream for the function instance.
- `RemainingTime (TimeSpan)` – The number of milliseconds left before the execution times out.
- `Identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
- `ClientContext` – (mobile apps) Client context that's provided to Lambda by the client application.
- `Logger` The [logger object \(p. 506\)](#) for the function.

The following C# code snippet shows a simple handler function that prints some of the context information.

```
public async Task Handler(ILambdaContext context)
{
    Console.WriteLine("Function name: " + context.FunctionName);
    Console.WriteLine("RemainingTime: " + context.RemainingTime);
    await Task.Delay(TimeSpan.FromSeconds(0.42));
    Console.WriteLine("RemainingTime after sleep: " + context.RemainingTime);
}
```

Lambda function logging in C#

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

This page describes how to produce log output from your Lambda function's code, or access logs using the AWS Command Line Interface, the Lambda console, the CloudWatch console, or Infrastructure as code tools such as the AWS Serverless Application Model(AWS SAM).

Sections

- [Tools and libraries \(p. 506\)](#)
- [Creating a function that returns logs \(p. 506\)](#)
- [Using log levels \(p. 508\)](#)
- [Using AWS Lambda Powertools for .NET and AWS SAM for structured logging \(p. 508\)](#)
- [Using the Lambda console \(p. 510\)](#)
- [Using the CloudWatch console \(p. 510\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 511\)](#)
- [Deleting logs \(p. 513\)](#)

Tools and libraries

[AWS Lambda Powertools for .NET](#) is a developer toolkit to implement Serverless best practices and increase developer velocity. The [Logging utility](#) provides a Lambda optimized logger which includes additional information about function context across all your functions with output structured as JSON. Use this utility to do the following:

- Capture key fields from the Lambda context, cold start and structures logging output as JSON
- Log Lambda invocation events when instructed (disabled by default)
- Print all the logs only for a percentage of invocations via log sampling (disabled by default)
- Append additional keys to structured log at any point in time
- Use a custom log formatter (Bring Your Own Formatter) to output logs in a structure compatible with your organization's Logging RFC

Creating a function that returns logs

To output logs from your function code, you can use methods on the [Console class](#), or any logging library that writes to `stdout` or `stderr`. The following example uses the `LambdaLogger` class from the [Amazon.Lambda.Core \(p. 487\)](#) library.

Example [src/blank-csharp/Function.cs](#) – Logging

```
public async Task<AccountUsage> FunctionHandler(SQSEvent invocationEvent, ILambdaContext context)
{
    GetAccountSettingsResponse accountSettings;
    try
```

```

    {
        accountSettings = await callLambda();
    }
    catch (AmazonLambdaException ex)
    {
        throw ex;
    }
    AccountUsage accountUsage = accountSettings.AccountUsage;
    LambdaLogger.Log("ENVIRONMENT VARIABLES: " +
JsonConvert.SerializeObject(System.Environment.GetEnvironmentVariables()));
    LambdaLogger.Log("CONTEXT: " + JsonConvert.SerializeObject(context));
    LambdaLogger.Log("EVENT: " + JsonConvert.SerializeObject(invocationEvent));
    return accountUsage;
}

```

Example log format

```

START RequestId: d1cf0ccb-xmpl-46e6-950d-04c96c9b1c5d Version: $LATEST
ENVIRONMENT VARIABLES:
{
    "AWS_EXECUTION_ENV": "AWS_Lambda_dotnet6",
    "AWS_LAMBDA_FUNCTION_MEMORY_SIZE": "256",
    "AWS_LAMBDA_LOG_GROUP_NAME": "/aws/lambda/blank-csharp-function-WU56XmplV2XA",
    "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
    "AWS_LAMBDA_LOG_STREAM_NAME": "2020/03/27/[${LATEST}]5296xmpl084f411d9fb73b258393f30c",
    "AWS_LAMBDA_FUNCTION_NAME": "blank-csharp-function-WU56XmplV2XA",
    ...
EVENT:
{
    "Records": [
        {
            "MessageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
            "ReceiptHandle": "MessageReceiptHandle",
            "Body": "Hello from SQS!",
            "Md5OfBody": "7b270e59b47ff90a553787216d55d91d",
            "Md5OfMessageAttributes": null,
            "EventSourceArn": "arn:aws:sqs:us-west-2:123456789012:MyQueue",
            "EventSource": "aws:sqs",
            "AwsRegion": "us-west-2",
            "Attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1523232000000",
                "SenderId": "123456789012",
                "ApproximateFirstReceiveTimestamp": "152323200001"
            },
            ...
        }
    ]
}
END RequestId: d1cf0ccb-xmpl-46e6-950d-04c96c9b1c5d
REPORT RequestId: d1cf0ccb-xmpl-46e6-950d-04c96c9b1c5d Duration: 4157.16 ms Billed Duration: 4200 ms Memory Size: 256 MB Max Memory Used: 99 MB Init Duration: 841.60 ms XRAY TraceId: 1-5e7e8131-7ff0xmpl32bfb31045d0a3bb SegmentId: 0152xmpl6016310f Sampled: true

```

The .NET runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details.

Report Log

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function.

- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TraceId** – For traced requests, the [AWS X-Ray trace ID \(p. 807\)](#).
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

Using log levels

Starting with .NET 6, you can use log levels for additional logging from Lambda functions. Log levels provide filtering and categorization for the logs that your function writes to Amazon EventBridge (CloudWatch Events).

The log levels available are:

- `LogCritical`
- `.LogError`
- `.LogWarning`
- `.LogInformation`
- `LogDebug`
- `LogTrace`

By default, Lambda writes `.LogInformation` level logs and above to CloudWatch. You can adjust the level of logs that Lambda writes using the `AWS_LAMBDA_HANDLER_LOG_LEVEL` environment variable. Set the value of the environment variable to the string enum value for the level desired, as outlined in the [LogLevel enum](#). For example, if you set `AWS_LAMBDA_HANDLER_LOG_LEVEL` to `Error`, Lambda writes `.LogError` and `LogCritical` messages to CloudWatch.

Lambda writes `Console.WriteLine` calls as info level messages, and `Console.Error.WriteLine` calls as error level messages.

If you prefer the previous style of logging in .NET, set `AWS_LAMBDA_HANDLER_LOG_FORMAT` to `Unformatted`.

Using AWS Lambda Powertools for .NET and AWS SAM for structured logging

Follow the steps below to download, build, and deploy a sample Hello World C# application with integrated [AWS Lambda Powertools for .NET](#) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- .NET 6
- [AWS CLI version 2](#)
- [AWS SAM CLI version 1.75 or later](#). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.

```
 sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type Zip  
--runtime dotnet6 --no-tracing
```

- ## 2. Build the app.

```
cd sam-app && sam build
```

- ### 3. Deploy the app.

```
 sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press **Enter**.

Note

For HelloWorldFunction may not have authorization defined, Is this okay?, make sure to enter y.

- ## 5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

- ## 6. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

For more information about the study, please contact Dr. John Smith at (555) 123-4567 or via email at john.smith@researchinstitute.org.

7. To get the logs for the function, run `sam logs`. For more information, see [Working with logs](#) in the AWS Cloud9 Application Model Guide.

```
 sam logs --stack-name sam-app
```

The log output looks like this:

```
2023/02/20/[LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8
2023-02-20T14:15:27.988000 INIT_START Runtime Version:
dotnet:6.v13           Runtime Version ARN: arn:aws:lambda:ap-
southeast-2::runtime:699f346a05dae24c58c45790bc4089f252bf17da3997e79b17d939a288aa1ec
2023/02/20/[LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:28.229000 START
RequestID: bed25b38-d012-42e7-ba28-f272535fb80e Version: $LATEST
2023/02/20/[LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:29.259000
2023-02-20T14:15:29.201Z      bed25b38-d012-42e7-ba28-f272535fb80e  info
    {"_aws": [{"Timestamp": "1676902528962", "CloudWatchMetrics": [{"Namespace": "sam-
app-logging", "Metrics": [{"Name": "ColdStart", "Unit": "Count"}]}, {"Dimensions": [
        {"FunctionName": "HelloWorldFunction", "Service": "HelloWorld"}, {"FunctionName": "HelloWorldFunction", "Service": "HelloWorld"}]}], "FunctionName": "HelloWorldFunction", "ColdStart": 1}
2023/02/20/[LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.479000
2023-02-20T14:15:30.479Z      bed25b38-d012-42e7-ba28-f272535fb80e  info
    {"ColdStart": true, "XrayTraceId": "1-63f3807f-5dbcb9910c96f50742707542", "CorrelationId": "d3d4de7f-4a549-4d67b2fdc015", "FunctionName": "HelloWorldFunction", "FunctionVersion": "$LATEST", "FunctionMemorySize": 256, "FunctionArn": "arn:aws:lambda:ap-
```

```
southeast-2:123456789012:function:sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionRequestId":"bed25b38-d012-42e7-ba28-
f272535fb80e","Timestamp":"2023-02-20T14:15:30.4602970Z","Level":"Information","Service":"Powertools
world API - HTTP 200"}  
2023/02/20/[${LATEST}]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.599000
2023-02-20T14:15:30.599Z      bed25b38-d012-42e7-ba28-f272535fb80e    info
  {"_aws":{"Timestamp":1676902528922,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ApiRequestCount","Unit":"Count"}],"Dimensions":
[{"Service"}]}],"Service":"PowertoolsHelloWorld","ApiRequestCount":1}  
2023/02/20/[${LATEST}]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.680000 END
RequestId: bed25b38-d012-42e7-ba28-f272535fb80e
2023/02/20/[${LATEST}]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.680000 REPORT
RequestId: bed25b38-d012-42e7-ba28-f272535fb80e Duration: 2450.99 ms Billed
Duration: 2451 ms Memory Size: 256 MB Max Memory Used: 74 MB Init Duration:
240.05 ms
XRAY TraceId: 1-63f3807f-5dbc9910c96f50742707542 SegmentId: 16b362cd5f52cba0
```

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
 sam delete
```

Managing log retention

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which CloudWatch automatically deletes the logs. To set up log retention, add the following to your AWS SAM template:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Using the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

Using the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the [Log groups page](#) on the CloudWatch console.
2. Choose the log group for your function (`/aws/lambda/your-function-name`).
3. Choose a log stream.

Each log stream corresponds to an [instance of your function \(p. 14\)](#). A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple concurrent

invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see [Error processor sample application for AWS Lambda \(p. 1015\)](#).

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

You can use the [AWS CLI](#) to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{  
  "StatusCode": 200,  
  "LogResult":  
    "U1RBULQgUmVxdWVzdElkOia4N2QwNDRiOC1mMTU0LTEzTgt0GNkYS0y0Tc0YzVlNGZiMjEgVmVyc21vb...",  
  "ExecutedVersion": "$LATEST"  
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \  
--query 'LogResult' --output text | base64 -d
```

You should see the following output:

```
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
```

The `base64` utility is available on Linux, macOS, and [Ubuntu on Windows](#). macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses sed to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the get-log-events command.

Copy the contents of the following code sample and save in your Lambda project directory as get-logs.sh.

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

```
#!/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 and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf \tINFO\tENVIRONMENT VARIABLES\r{\r\t\t\"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\", \r\t\t...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf \tINFO\tEVENT\r{\r\t\t\"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        }
    ]
}
```

```
        "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",
        "ingestionTime": 1559763018353
    }
],
"nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
"nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.

AWS Lambda function errors in C#

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

This page describes how to view Lambda function invocation errors for the C# runtime using the Lambda console and the AWS CLI.

Sections

- [Syntax \(p. 514\)](#)
- [How it works \(p. 516\)](#)
- [Using the Lambda console \(p. 517\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 517\)](#)
- [Error handling in other AWS services \(p. 518\)](#)
- [What's next? \(p. 518\)](#)

Syntax

In the initialization phase, exceptions can be thrown for invalid handler strings, a rule-breaking type or method (see [Lambda function handler restrictions \(p. 495\)](#)), or any other validation method (such as forgetting the serializer attribute and having a POCO as your input or output type). These exceptions are of type `LambdaException`. For example:

```
{  
  "errorType": "LambdaException",  
  "errorMessage": "Invalid lambda function handler: 'http://this.is.not.a.valid.handler/'.  
  The valid format is 'ASSEMBLY::TYPE::METHOD'."  
}
```

If your constructor throws an exception, the error type is also of type `LambdaException`, but the exception thrown during construction is provided in the `cause` property, which is itself a modeled exception object:

```
{  
  "errorType": "LambdaException",  
  "errorMessage": "An exception was thrown when the constructor for type  
'LambdaExceptionTestFunction.ThrowExceptionInConstructor'  
  was invoked. Check inner exception for more details.",  
  "cause": {  
    "errorType": "TargetInvocationException",  
    "errorMessage": "Exception has been thrown by the target of an invocation.",  
    "stackTrace": [  
      "at System.RuntimeTypeHandle.CreateInstance(RuntimeType type, Boolean publicOnly,  
Boolean noCheck, Boolean&canBeCached,  
      RuntimeMethodHandleInternal&ctor, Boolean& bNeedSecurityCheck)",  
      "at System.RuntimeType.CreateInstanceSlow(Boolean publicOnly, Boolean skipCheckThis,  
Boolean fillCache, StackCrawlMark& stackMark)",  
      "at System.Activator.CreateInstance(Type type, Boolean nonPublic)",  
      "at System.Activator.CreateInstance(Type type)"  
    ],  
    "cause": {  
      "errorType": "AritheticException",  
      "errorMessage": "Sorry, 2 + 2 = 5",  
      "stackTrace": [  
        "at LambdaExceptionTestFunction.ThrowExceptionInConstructor..ctor()"  
      ]  
    }  
  }  
}
```

```
    }
}
```

As the example shows, the inner exceptions are always preserved (as the cause property), and can be deeply nested.

Exceptions can also occur during invocation. In this case, the exception type is preserved and the exception is returned directly as the payload and in the CloudWatch logs. For example:

```
{
  "errorType": "AggregateException",
  "errorMessage": "One or more errors occurred. (An unknown web exception occurred!)",
  "stackTrace": [
    "at System.Threading.Tasks.Task.ThrowIfExceptional(Boolean
includeTaskCanceledExceptions)",
    "at System.Threading.Tasks.Task`1.GetResultCore(Boolean waitCompletionNotification)",
    "at lambda_method(Closure , Stream , Stream , ContextInfo )"
  ],
  "cause": {
    "errorType": "UnknownWebException",
    "errorMessage": "An unknown web exception occurred!",
    "stackTrace": [
      "at LambdaDemo107.LambdaEntryPoint.<GetUriResponse>d__1.MoveNext()",
      "--- End of stack trace from previous location where exception was thrown ---",
      "at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
      "at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
task)",
      "at System.Runtime.CompilerServices.TaskAwaiter`1.GetResult()",  

      "at LambdaDemo107.LambdaEntryPoint.<CheckWebsiteStatus>d__0.MoveNext()"
    ],
    "cause": {
      "errorType": "WebException",
      "errorMessage": "An error occurred while sending the request. SSL peer certificate or
SSH remote key was not OK",
      "stackTrace": [
        "at System.Net.HttpWebRequest.EndGetResponse(IAsyncResult asyncResult)",
        "at System.Threading.Tasks.TaskFactory`1.FromAsyncCoreLogic(IAsyncResult iar,
Func`2 endFunction, Action`1 endAction, Task`1 promise, Boolean requiresSynchronization)",
        "--- End of stack trace from previous location where exception was thrown ---",
        "at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
        "at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
task)",
        "at System.Runtime.CompilerServices.TaskAwaiter`1.GetResult()",  

        "at LambdaDemo107.LambdaEntryPoint.<GetUriResponse>d__1.MoveNext()"
      ],
      "cause": {
        "errorType": "HttpRequestException",
        "errorMessage": "An error occurred while sending the request.",
        "stackTrace": [
          "at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
          "at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
task)",
          "at System.Net.Http.HttpClient.<FinishSendAsync>d__58.MoveNext()",
          "--- End of stack trace from previous location where exception was thrown ---",
          "at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",
          "at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
task)",
          "at System.Net.HttpWebRequest.<SendRequest>d__63.MoveNext()",
          "--- End of stack trace from previous location where exception was thrown ---",
          "at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)",  

          "at System.Net.HttpWebRequest.<SendRequest>d__63.MoveNext()",
          "--- End of stack trace from previous location where exception was thrown ---",
          "at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)"
        ]
      }
    }
}
```

```
"at
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task
task)",
    "at System.Net.HttpWebRequest.EndGetResponse(IAsyncResult asyncResult)"
],
"cause": {
    "errorType": "CurlException",
    "errorMessage": "SSL peer certificate or SSH remote key was not OK",
    "stackTrace": [
        "at System.Net.Http.CurlHandler.ThrowIfCURLError(CURLcode error)",
        "at
System.Net.Http.CurlHandler.MultiAgent.FinishRequest(StrongToWeakReference`1 easyWrapper,
CURLcode messageResult)"
    ]
}
}
```

The method in which error information is conveyed depends on the invocation type:

- RequestResponse invocation type (that is, synchronous execution): In this case, you get the error message back.

For example, if you invoke a Lambda function using the Lambda console, the RequestResponse is always the invocation type and the console displays the error information returned by AWS Lambda in the **Execution result** section of the console.

- Event invocation type (that is, asynchronous execution): In this case AWS Lambda does not return anything. Instead, it logs the error information in CloudWatch Logs and CloudWatch metrics.

How it works

When you invoke a Lambda function, Lambda receives the invocation request and validates the permissions in your execution role, verifies that the event document is a valid JSON document, and checks parameter values.

If the request passes validation, Lambda sends the request to a function instance. The [Lambda runtime \(p. 37\)](#) environment converts the event document into an object, and passes it to your function handler.

If Lambda encounters an error, it returns an exception type, message, and HTTP status code that indicates the cause of the error. The client or service that invoked the Lambda function can handle the error programmatically, or pass it along to an end user. The correct error handling behavior depends on the type of application, the audience, and the source of the error.

The following list describes the range of status codes you can receive from Lambda.

2xx

A 2xx series error with a X-Amz-Function-Error header in the response indicates a Lambda runtime or function error. A 2xx series status code indicates that Lambda accepted the request, but instead of an error code, Lambda indicates the error by including the X-Amz-Function-Error header in the response.

4xx

A 4xx series error indicates an error that the invoking client or service can fix by modifying the request, requesting permission, or by retrying the request. 4xx series errors other than 429 generally indicate an error with the request.

5xx

A 5xx series error indicates an issue with Lambda, or an issue with the function's configuration or resources. 5xx series errors can indicate a temporary condition that can be resolved without any action by the user. These issues can't be addressed by the invoking client or service, but a Lambda function's owner may be able to fix the issue.

For a complete list of invocation errors, see [InvokeFunction errors \(p. 1262\)](#).

Using the Lambda console

You can invoke your function on the Lambda console by configuring a test event and viewing the output. The output is captured in the function's execution logs and, when [active tracing \(p. 807\)](#) is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to test, and choose **Test**.
3. Under **Test event**, select **New event**.
4. Select a **Template**.
5. For **Name**, enter a name for the test. In the text entry box, enter the JSON test event.
6. Choose **Save changes**.
7. Choose **Test**.

The Lambda console invokes your function [synchronously \(p. 120\)](#) and displays the result. To see the response, logs, and other information, expand the **Details** section.

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

When you invoke a Lambda function in the AWS CLI, the AWS CLI splits the response into two documents. The AWS CLI response is displayed in your command prompt. If an error has occurred, the response contains a `FunctionError` field. The invocation response or error returned by the function is written to an output file. For example, `output.json` or `output.txt`.

The following `invoke` command example demonstrates how to invoke a function and write the invocation response to an `output.txt` file.

```
aws lambda invoke \
--function-name my-function \
--cli-binary-format raw-in-base64-out \
--payload '{"key1": "value1", "key2": "value2", "key3": "value3"}' output.txt
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

You should see the AWS CLI response in your command prompt:

```
{  
    "StatusCode": 200,  
    "FunctionError": "Unhandled",  
    "ExecutedVersion": "$LATEST"  
}
```

You should see the function invocation response in the `output.txt` file. In the same command prompt, you can also view the output in your command prompt using:

```
cat output.txt
```

You should see the invocation response in your command prompt.

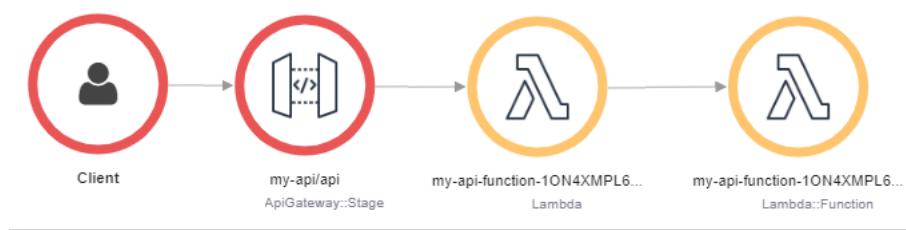
Lambda also records up to 256 KB of the error object in the function's logs. For more information, see [Lambda function logging in C# \(p. 506\)](#).

Error handling in other AWS services

When another AWS service invokes your function, the service chooses the invocation type and retry behavior. AWS services can invoke your function on a schedule, in response to a lifecycle event on a resource, or to serve a request from a user. Some services invoke functions asynchronously and let Lambda handle errors, while others retry or pass errors back to the user.

For example, API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502 error code. To customize the error response, you must catch errors in your code and format a response in the required format.

We recommend using AWS X-Ray to determine the source of an error and its cause. X-Ray allows you to find out which component encountered an error, and see details about the errors. The following example shows a function error that resulted in a 502 response from API Gateway.



For more information, see [Instrumenting C# code in AWS Lambda \(p. 519\)](#).

What's next?

- Learn how to show logging events for your Lambda function on the [the section called "Logging" \(p. 506\)](#) page.

Instrumenting C# code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of three SDK libraries:

- [AWS Distro for OpenTelemetry \(ADOT\)](#) – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- [AWS X-Ray SDK for .NET](#) – An SDK for generating and sending trace data to X-Ray.
- [AWS Lambda Powertools for .NET](#) – A developer toolkit to implement Serverless best practices and increase developer velocity.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

Important

The X-Ray and AWS Lambda Powertools SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see [Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs](#).

Sections

- [Using AWS Lambda Powertools for .NET and AWS SAM for tracing \(p. 519\)](#)
- [Using the X-Ray SDK to instrument your .NET functions \(p. 521\)](#)
- [Activating tracing with the Lambda console \(p. 522\)](#)
- [Activating tracing with the Lambda API \(p. 522\)](#)
- [Activating tracing with AWS CloudFormation \(p. 522\)](#)
- [Interpreting an X-Ray trace \(p. 523\)](#)

Using AWS Lambda Powertools for .NET and AWS SAM for tracing

Follow the steps below to download, build, and deploy a sample Hello World C# application with integrated [AWS Lambda Powertools for .NET](#) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- .NET 6
- [AWS CLI version 2](#)
- [AWS SAM CLI version 1.75 or later](#). If you have an older version of the AWS SAM CLI, see [Upgrading the AWS SAM CLI](#).

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type Zip  
--runtime dotnet6 --no-tracing
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press Enter.

Note

For **HelloWorldFunction** may not have authorization defined, Is this okay?, make sure to enter y.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message": "hello world"}
```

7. To get the traces for the function, run [sam traces](#).

```
sam traces
```

The trace output looks like this:

```
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. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

sam delete

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

Note

You cannot configure the X-Ray sampling rate for your functions.

Using the X-Ray SDK to instrument your .NET functions

You can instrument your function code to record metadata and trace downstream calls. To record detail about calls that your function makes to other resources and services, use the X-Ray SDK for .NET. To get the SDK, add the `AWSXRayRecorder` packages to your project file.

Example [src/blank-csharp/blank-csharp.csproj](#)

```

<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
    <TargetFramework>net6.0</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>
```

To instrument AWS SDK clients, call the `RegisterXRayForAllServices` method in your initialization code.

Example [src/blank-csharp/Function.cs](#) – Initialize X-Ray

```
{
    AWSSDKHandler.RegisterXRayForAllServices();
    lambdaClient = new AmazonLambdaClient();
    await callLambda();
}
```

Activating tracing with the Lambda console

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring and operations tools**.
4. Choose **Edit**.
5. Under **X-Ray**, toggle on **Active tracing**.
6. Choose **Save**.

Activating tracing with the Lambda API

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- [UpdateFunctionConfiguration \(p. 1377\)](#)
- [GetFunctionConfiguration \(p. 1229\)](#)
- [CreateFunction \(p. 1165\)](#)

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Activating tracing with AWS CloudFormation

To activate tracing on an `AWS::Lambda::Function` resource in an AWS CloudFormation template, use the `TracingConfig` property.

Example [function-inline.yml](#) – Tracing configuration

```
Resources:  
  function:  
    Type: AWS::Lambda::Function  
    Properties:  
      TracingConfig:  
        Mode: Active  
      ...
```

For an AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function resource, use the Tracing property.

Example [template.yml](#) – Tracing configuration

```
Resources:  
  function:  
    Type: AWS::Serverless::Function  
    Properties:  
      Tracing: Active  
      ...
```

Interpreting an X-Ray trace

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's [execution role \(p. 816\)](#). Otherwise, add the [AWSXRayDaemonWriteAccess](#) policy to the execution role.

After you've configured active tracing, you can observe specific requests through your application. The [X-Ray service graph](#) shows information about your application and all its components. The following example from the [error processor \(p. 1015\)](#) sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.

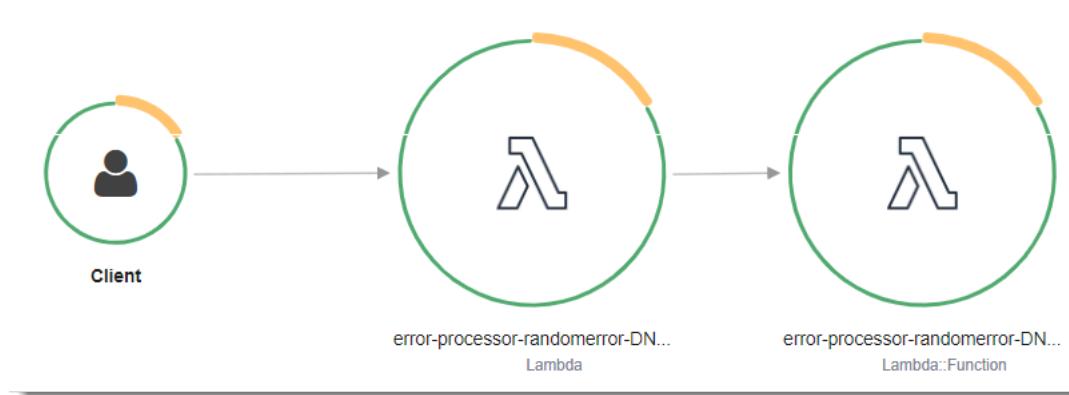


X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

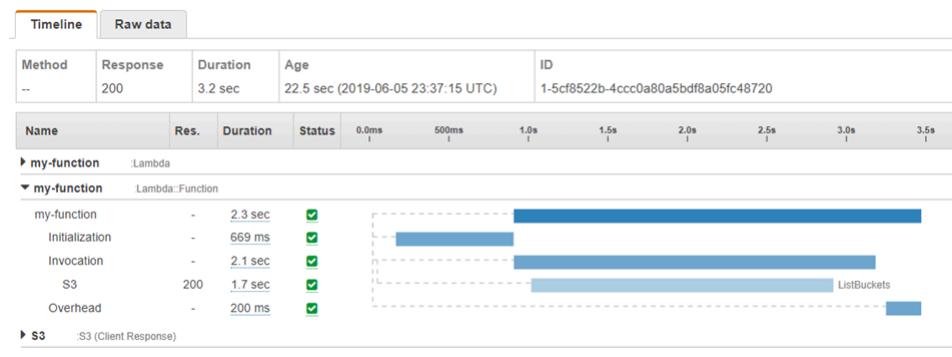
Note

You cannot configure the X-Ray sampling rate for your functions.

When using active tracing, Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes for the primary function from the [error processor sample application \(p. 1015\)](#).



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of AWS : : Lambda and the other has origin AWS : : Lambda : : Function.



This example expands the function segment to show its three subsegments:

- **Initialization** – Represents time spent loading your function and running [initialization code \(p. 13\)](#). This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see the [AWS X-Ray SDK for .NET](#) in the [AWS X-Ray Developer Guide](#).

Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see [AWS X-Ray pricing](#).

.NET functions with native AOT compilation

.NET 7 supports native ahead-of-time (AOT) compilation. With *native AOT*, you can compile your Lambda function code to a native runtime format, which removes the need to compile .NET code at runtime. Native AOT compilation can reduce the cold start time for Lambda functions that you write in .NET. For more information, see [Building serverless .NET applications on AWS Lambda using .NET 7](#) on the AWS Compute Blog.

Sections

- [Limitations \(p. 525\)](#)
- [Prerequisites \(p. 525\)](#)
- [Lambda runtime \(p. 526\)](#)
- [Set up your project \(p. 526\)](#)
- [Edit your Lambda function code \(p. 526\)](#)
- [Deploy your Lambda function \(p. 526\)](#)
- [Add support for complex types \(p. 526\)](#)
- [Troubleshooting \(p. 527\)](#)

Limitations

There are limitations to functionality that you can include in native AOT functions. For more information, see [Limitations of Native AOT deployment](#) on the Microsoft Learn website.

Prerequisites

Docker

You must compile your function with native AOT on the same operating system that your code will run on. As a result, on any operating system other than Amazon Linux 2, you need Docker to develop Lambda functions that use native AOT.

.NET 7 SDK

Native AOT compilation is a feature of .NET 7. You must install the [.NET 7 SDK](#) on your build machine, not only the runtime.

Amazon.Lambda.Tools

To create your Lambda functions, use the [Amazon.Lambda.Tools .NET Core global tool](#). The current version of the .NET Core global tool for Lambda supports using Docker for native AOT. To install Amazon.Lambda.Tools, run the following command:

```
dotnet tool install -g Amazon.Lambda.Tools
```

For more information about the Amazon.Lambda.Tools .NET Core global tool, see the [AWS Extensions for .NET CLI](#) repository on GitHub.

Amazon.Lambda.Templates

To generate your Lambda function code, use the [Amazon.Lambda.Templates](#) NuGet package. To install this template package, run the following command:

```
dotnet new install Amazon.Lambda.Templates
```

Lambda runtime

Use the provided .a12 custom runtime with the x86_64 architecture to deploy a Lambda function that you build with native AOT compilation. When you use a .NET Lambda runtime, your application is compiled into Intermediate Language (IL) code. At runtime, the just-in-time (JIT) compiler takes the IL code and compiles it into machine code as needed. With a Lambda function that is compiled ahead of time with native AOT, the runtime environment doesn't include the .NET SDK or .NET runtime. You compile your code into machine code before it runs.

Set up your project

Use the .NET Core global tool for Lambda to create your new environment. To initialize your project, run the following command:

```
dotnet new lambda.NativeAOT
```

Edit your Lambda function code

The .NET Core global tool for Lambda generates a basic Lambda function that accepts a `String` and returns a `String`. Edit the function code as required for your use case. For how to change the parameters and return types of your function, see [the section called "Add support for complex types" \(p. 526\)](#).

Deploy your Lambda function

If you're using Windows or macOS, make sure that Docker is running.

Then, to compile and deploy your Lambda function, run the following command:

```
dotnet lambda deploy-function
```

The `deploy-function` process automatically downloads a Docker image of Amazon Linux 2 to perform the native AOT compilation for your function. After this container image has downloaded, the deploy process builds your function and then creates a `.zip` file that gets deployed into your AWS account.

Add support for complex types

The default Lambda function provides a basic starting point of a `String` parameter and a `String` return type. To accept or return complex types, update your code to include functionality that generates serialization code at compile time, rather than at runtime.

Add more `JsonSerializable` attributes to your custom serializer object definition as needed.

API Gateway example

For example, to use Amazon API Gateway events, add a reference to the NuGet package `Amazon.Lambda.ApiGatewayEvents`. Then, add the following using statement to your Lambda function source code:

```
using Amazon.Lambda.APIGatewayEvents;
```

Add the following attributes to your class definition of your custom serializer:

```
[JsonSerializable(typeof(APIGatewayHttpApiV2ProxyRequest))]  
[JsonSerializable(typeof(APIGatewayHttpApiV2ProxyResponse))]
```

Update your function signature to the following:

```
public static async Task<APIGatewayHttpApiV2ProxyResponse>  
    FunctionHandler(APIGatewayHttpApiV2ProxyRequest input, ILambdaContext context)
```

Troubleshooting

Error: Cross-OS native compilation is not supported.

Your version of the Amazon.Lambda.Tools .NET Core global tool is out of date. Update to the latest version and try again.

Docker: image operating system "linux" cannot be used on this platform.

Docker on your system is configured to use Windows containers. Swap to Linux containers to run the native AOT build environment.

Unhandled Exception: System.ApplicationException: The serializer NativeAoT.MyCustomJsonSerializerContext is missing a constructor that takes in JsonSerializerOptions object

If you encounter this error when you invoke your Lambda function, add an `rd.xml` file to your project, and then redeploy.

For more information about common errors, see the [AWS NativeAOT for .NET](#) repository on GitHub.

Building Lambda functions with PowerShell

The following sections explain how common programming patterns and core concepts apply when you author Lambda function code in PowerShell.

.NET

Name	Identifier	Operating system	Architectures	Deprecation (Phase 1)
.NET Core 3.1	dotnetcore3.1	Amazon Linux 2	x86_64, arm64	Apr 3, 2023
.NET 7	dotnet7	Amazon Linux 2	x86_64, arm64	
.NET 6	dotnet6	Amazon Linux 2	x86_64, arm64	
.NET 5	dotnet5.0	Amazon Linux 2	x86_64	

Lambda provides the following sample applications for the PowerShell runtime:

- [blank-powershell](#) – A PowerShell function that shows the use of logging, environment variables, and the AWS SDK.

Before you get started, you must first set up a PowerShell development environment. For instructions on how to do this, see [Setting Up a PowerShell Development Environment \(p. 529\)](#).

To learn about how to use the AWSLambdaPSCore module to download sample PowerShell projects from templates, create PowerShell deployment packages, and deploy PowerShell functions to the AWS Cloud, see [Deploy PowerShell Lambda functions with .zip file archives \(p. 530\)](#).

Topics

- [Setting Up a PowerShell Development Environment \(p. 529\)](#)
- [Deploy PowerShell Lambda functions with .zip file archives \(p. 530\)](#)
- [AWS Lambda function handler in PowerShell \(p. 532\)](#)
- [AWS Lambda context object in PowerShell \(p. 533\)](#)
- [AWS Lambda function logging in PowerShell \(p. 534\)](#)
- [AWS Lambda function errors in PowerShell \(p. 539\)](#)

Setting Up a PowerShell Development Environment

Lambda provides a set of tools and libraries for the PowerShell runtime. For installation instructions, see [Lambda tools for PowerShell](#) on GitHub.

The AWSLambdaPSCore module includes the following cmdlets to help author and publish PowerShell Lambda functions:

- **Get-AWSPowerShellLambdaTemplate** – Returns a list of getting started templates.
- **New-AWSPowerShellLambda** – Creates an initial PowerShell script based on a template.
- **Publish-AWSPowerShellLambda** – Publishes a given PowerShell script to Lambda.
- **New-AWSPowerShellLambdaPackage** – Creates a Lambda deployment package that you can use in a CI/CD system for deployment.

Deploy PowerShell Lambda functions with .zip file archives

A deployment package for the PowerShell runtime contains your PowerShell script, PowerShell modules that are required for your PowerShell script, and the assemblies needed to host PowerShell Core.

Creating the Lambda function

To get started writing and invoking a PowerShell script with Lambda, you can use the `New-AWSPowerShellLambda` cmdlet to create a starter script based on a template. You can use the `Publish-AWSPowerShellLambda` cmdlet to deploy your script to Lambda. Then you can test your script either through the command line or the Lambda console.

To create a new PowerShell script, upload it, and test it, do the following:

1. To view the list of available templates, run the following command:

```
PS C:\> Get-AWSPowerShellLambdaTemplate

Template          Description
-----          -----
Basic            Bare bones script
CodeCommitTrigger Script to process AWS CodeCommit Triggers
...
```

2. To create a sample script based on the Basic template, run the following command:

```
New-AWSPowerShellLambda -ScriptName MyFirstPSScript -Template Basic
```

A new file named `MyFirstPSScript.ps1` is created in a new subdirectory of the current directory. The name of the directory is based on the `-ScriptName` parameter. You can use the `-Directory` parameter to choose an alternative directory.

You can see that the new file has the following contents:

```
# 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. To see how log messages from your PowerShell script are sent to Amazon CloudWatch Logs, uncomment the `Write-Host` line of the sample script.

To demonstrate how you can return data back from your Lambda functions, add a new line at the end of the script with `$PSVersionTable`. This adds the `$PSVersionTable` to the PowerShell pipeline. After the PowerShell script is complete, the last object in the PowerShell pipeline is the return data for the Lambda function. `$PSVersionTable` is a PowerShell global variable that also provides information about the running environment.

After making these changes, the last two lines of the sample script look like this:

```
Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)  
$PSVersionTable
```

4. After editing the `MyFirstPSScript.ps1` file, change the directory to the script's location. Then run the following command to publish the script to Lambda:

```
Publish-AWSPowerShellLambda -ScriptPath .\MyFirstPSScript.ps1 -Name MyFirstPSScript -  
Region us-east-2
```

Note that the `-Name` parameter specifies the Lambda function name, which appears in the Lambda console. You can use this function to invoke your script manually.

5. Invoke your function using the AWS Command Line Interface (AWS CLI) `invoke` command.

```
> aws lambda invoke --function-name MyFirstPSScript out
```

AWS Lambda function handler in PowerShell

When a Lambda function is invoked, the Lambda handler invokes the PowerShell script.

When the PowerShell script is invoked, the following variables are predefined:

- **\$LambdaInput** – A PSObject that contains the input to the handler. This input can be event data (published by an event source) or custom input that you provide, such as a string or any custom data object.
- **\$LambdaContext** – An Amazon.Lambda.Core.ILambdaContext object that you can use to access information about the current invocation—such as the name of the current function, the memory limit, execution time remaining, and logging.

For example, consider the following PowerShell example code.

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function Name:' $LambdaContext.FunctionName
```

This script returns the `FunctionName` property that's obtained from the `$LambdaContext` variable.

Note

You're required to use the `#Requires` statement within your PowerShell scripts to indicate the modules that your scripts depend on. This statement performs two important tasks. 1) It communicates to other developers which modules the script uses, and 2) it identifies the dependent modules that AWS PowerShell tools need to package with the script, as part of the deployment. For more information about the `#Requires` statement in PowerShell, see [About requires](#). For more information about PowerShell deployment packages, see [Deploy PowerShell Lambda functions with .zip file archives \(p. 530\)](#).

When your PowerShell Lambda function uses the AWS PowerShell cmdlets, be sure to set a `#Requires` statement that references the `AWSPowerShell.NetCore` module, which supports PowerShell Core—and not the `AWSPowerShell` module, which only supports Windows PowerShell. Also, be sure to use version 3.3.270.0 or newer of `AWSPowerShell.NetCore` which optimizes the cmdlet import process. If you use an older version, you'll experience longer cold starts. For more information, see [AWS Tools for PowerShell](#).

Returning data

Some Lambda invocations are meant to return data back to their caller. For example, if an invocation was in response to a web request coming from API Gateway, then our Lambda function needs to return back the response. For PowerShell Lambda, the last object that's added to the PowerShell pipeline is the return data from the Lambda invocation. If the object is a string, the data is returned as is. Otherwise the object is converted to JSON by using the `ConvertTo-Json` cmdlet.

For example, consider the following PowerShell statement, which adds `$PSVersionTable` to the PowerShell pipeline:

```
$PSVersionTable
```

After the PowerShell script is finished, the last object in the PowerShell pipeline is the return data for the Lambda function. `$PSVersionTable` is a PowerShell global variable that also provides information about the running environment.

AWS Lambda context object in PowerShell

When Lambda runs your function, it passes context information by making a `$LambdaContext` variable available to the [handler \(p. 532\)](#). This variable provides methods and properties with information about the invocation, function, and execution environment.

Context properties

- `FunctionName` – The name of the Lambda function.
- `FunctionVersion` – The [version \(p. 83\)](#) of the function.
- `InvokedFunctionArn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `MemoryLimitInMB` – The amount of memory that's allocated for the function.
- `AwsRequestId` – The identifier of the invocation request.
- `LogGroupName` – The log group for the function.
- `LogStreamName` – The log stream for the function instance.
- `RemainingTime` – The number of milliseconds left before the execution times out.
- `Identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
- `ClientContext` – (mobile apps) Client context that's provided to Lambda by the client application.
- `Logger` – The [logger object \(p. 534\)](#) for the function.

The following PowerShell code snippet shows a simple handler function that prints some of the context information.

```
#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
```

AWS Lambda function logging in PowerShell

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

This page describes how to produce log output from your Lambda function's code, or access logs using the AWS Command Line Interface, the Lambda console, or the CloudWatch console.

Sections

- [Creating a function that returns logs \(p. 534\)](#)
- [Using the Lambda console \(p. 535\)](#)
- [Using the CloudWatch console \(p. 535\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 536\)](#)
- [Deleting logs \(p. 538\)](#)

Creating a function that returns logs

To output logs from your function code, you can use cmdlets on [Microsoft.PowerShell.Utility](#), or any logging module that writes to `stdout` or `stderr`. The following example uses `Write-Host`.

Example [function/Handler.ps1](#) – Logging

```
#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 log format

```
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: 4000 ms Memory Size: 512 MB Max Memory Used: 367 MB Init Duration: 5960.19 ms
XRAY TraceId: 1-5e843da6-733cxmple7d0c3c020510040 SegmentId: 3913xmpl20999446 Sampled: true
```

The .NET runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details.

Report Log

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY Traceld** – For traced requests, the [AWS X-Ray trace ID \(p. 807\)](#).
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

Using the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#).

Using the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the [Log groups page](#) on the CloudWatch console.
2. Choose the log group for your function (`/aws/lambda/your-function-name`).
3. Choose a log stream.

Each log stream corresponds to an [instance of your function \(p. 14\)](#). A log stream appears when you update your Lambda function, and when additional instances are created to handle multiple concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

To use a sample application that correlates logs and traces with X-Ray, see [Error processor sample application for AWS Lambda \(p. 1015\)](#).

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

You can use the [AWS CLI](#) to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{  
    "StatusCode": 200,  
    "LogResult":  
        "U1RBULQgUmVxdWVzdElk0iA4N2QwNDRi0C1mMTU0LTExZTgt0GNkYS0y0Tc0YzVlNGZiMjEgVmVyc21vb...",  
    "ExecutedVersion": "$LATEST"  
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \  
--query 'LogResult' --output text | base64 -d
```

You should see the following output:

```
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
```

The `base64` utility is available on Linux, macOS, and [Ubuntu on Windows](#). macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

```
#!/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 and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf \tINFO\tENVIRONMENT VARIABLES\r{\r\t\t\"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\", \r\t\t...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf \tINFO\tEVENT\r{\r\t\t\"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\nDuration:  
26.73 ms\nBilled Duration: 27 ms \nMemory Size: 128 MB\nMax Memory Used: 75 MB\n",  
        "ingestionTime": 1559763018353  
    }  
],  
"nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",  
"nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"  
}
```

Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or [configure a retention period](#) after which logs are deleted automatically.

AWS Lambda function errors in PowerShell

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

This page describes how to view Lambda function invocation errors for the PowerShell runtime using the Lambda console and the AWS CLI.

Sections

- [Syntax \(p. 539\)](#)
- [How it works \(p. 540\)](#)
- [Using the Lambda console \(p. 540\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 541\)](#)
- [Error handling in other AWS services \(p. 541\)](#)
- [What's next? \(p. 542\)](#)

Syntax

Consider the following PowerShell script example statement:

```
throw 'The Account is not found'
```

When you invoke this Lambda function, it throws a terminating error, and AWS Lambda returns the following error message:

```
{  
  "errorMessage": "The Account is not found",  
  "errorType": "RuntimeException"  
}
```

Note the `errorType` is `RuntimeException`, which is the default exception thrown by PowerShell. You can use custom error types by throwing the error like this:

```
throw @{'Exception'='AccountNotFound'; 'Message'='The Account is not found'}
```

The error message is serialized with `errorType` set to `AccountNotFound`:

```
{  
  "errorMessage": "The Account is not found",  
  "errorType": "AccountNotFound"  
}
```

If you don't need an error message, you can throw a string in the format of an error code. The error code format requires that the string starts with a character and only contain letters and digits afterwards, with no spaces or symbols.

For example, if your Lambda function contains the following:

```
throw 'AccountNotFound'
```

The error is serialized like this:

```
{  
  "errorMessage": "AccountNotFound",  
  "errorType": "AccountNotFound"  
}
```

How it works

When you invoke a Lambda function, Lambda receives the invocation request and validates the permissions in your execution role, verifies that the event document is a valid JSON document, and checks parameter values.

If the request passes validation, Lambda sends the request to a function instance. The [Lambda runtime \(p. 37\)](#) environment converts the event document into an object, and passes it to your function handler.

If Lambda encounters an error, it returns an exception type, message, and HTTP status code that indicates the cause of the error. The client or service that invoked the Lambda function can handle the error programmatically, or pass it along to an end user. The correct error handling behavior depends on the type of application, the audience, and the source of the error.

The following list describes the range of status codes you can receive from Lambda.

2xx

A 2xx series error with a X-Amz-Function-Error header in the response indicates a Lambda runtime or function error. A 2xx series status code indicates that Lambda accepted the request, but instead of an error code, Lambda indicates the error by including the X-Amz-Function-Error header in the response.

4xx

A 4xx series error indicates an error that the invoking client or service can fix by modifying the request, requesting permission, or by retrying the request. 4xx series errors other than 429 generally indicate an error with the request.

5xx

A 5xx series error indicates an issue with Lambda, or an issue with the function's configuration or resources. 5xx series errors can indicate a temporary condition that can be resolved without any action by the user. These issues can't be addressed by the invoking client or service, but a Lambda function's owner may be able to fix the issue.

For a complete list of invocation errors, see [InvokeFunction errors \(p. 1262\)](#).

Using the Lambda console

You can invoke your function on the Lambda console by configuring a test event and viewing the output. The output is captured in the function's execution logs and, when [active tracing \(p. 807\)](#) is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the [Functions page](#) of the Lambda console.
2. Choose the function to test, and choose **Test**.
3. Under **Test event**, select **New event**.
4. Select a **Template**.

5. For **Name**, enter a name for the test. In the text entry box, enter the JSON test event.
6. Choose **Save changes**.
7. Choose **Test**.

The Lambda console invokes your function [synchronously \(p. 120\)](#) and displays the result. To see the response, logs, and other information, expand the **Details** section.

Using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

When you invoke a Lambda function in the AWS CLI, the AWS CLI splits the response into two documents. The AWS CLI response is displayed in your command prompt. If an error has occurred, the response contains a `FunctionError` field. The invocation response or error returned by the function is written to an output file. For example, `output.json` or `output.txt`.

The following [invoke](#) command example demonstrates how to invoke a function and write the invocation response to an `output.txt` file.

```
aws lambda invoke \
--function-name my-function \
--cli-binary-format raw-in-base64-out \
--payload '{"key1": "value1", "key2": "value2", "key3": "value3"}' output.txt
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

You should see the AWS CLI response in your command prompt:

```
{  
  "StatusCode": 200,  
  "FunctionError": "Unhandled",  
  "ExecutedVersion": "$LATEST"  
}
```

You should see the function invocation response in the `output.txt` file. In the same command prompt, you can also view the output in your command prompt using:

```
cat output.txt
```

You should see the invocation response in your command prompt.

Lambda also records up to 256 KB of the error object in the function's logs. For more information, see [AWS Lambda function logging in PowerShell \(p. 534\)](#).

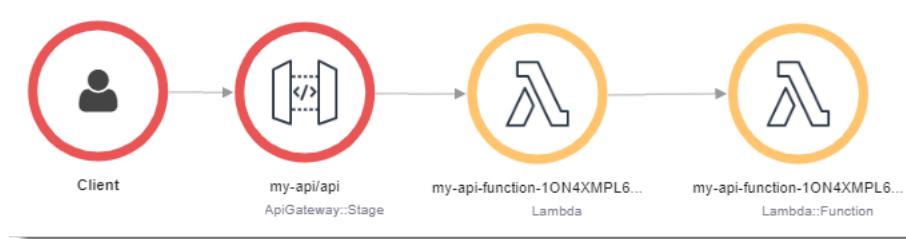
Error handling in other AWS services

When another AWS service invokes your function, the service chooses the invocation type and retry behavior. AWS services can invoke your function on a schedule, in response to a lifecycle event on a

resource, or to serve a request from a user. Some services invoke functions asynchronously and let Lambda handle errors, while others retry or pass errors back to the user.

For example, API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502 error code. To customize the error response, you must catch errors in your code and format a response in the required format.

We recommend using AWS X-Ray to determine the source of an error and its cause. X-Ray allows you to find out which component encountered an error, and see details about the errors. The following example shows a function error that resulted in a 502 response from API Gateway.



For more information, see [Using AWS Lambda with AWS X-Ray \(p. 807\)](#).

What's next?

- Learn how to show logging events for your Lambda function on the [the section called "Logging" \(p. 534\)](#) page.

Building Lambda functions with Rust

Because Rust compiles to native code, you don't need a dedicated runtime to run Rust code on Lambda. Instead, use the [Rust runtime client](#) to build your project locally, and then deploy it to Lambda using the provided .a12 runtime. When you use provided.a12, Lambda automatically keeps the operating system up to date with the latest patches.

Note

The [Rust runtime client](#) is an experimental package. It is subject to change and intended only for evaluation purposes.

Rust

Name	Identifier	Operating system	Architectures
Rust	provided.a12	Amazon Linux	x86_64, arm64

Lambda functions use an [execution role \(p. 816\)](#) to get permission to write logs to Amazon CloudWatch Logs, and to access other services and resources. If you don't already have an execution role for function development, create one.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – Lambda.**
 - **Permissions – AWSLambdaBasicExecutionRole.**
 - **Role name – lambda-role.**

The **AWSLambdaBasicExecutionRole** policy has the permissions that the function needs to write logs to CloudWatch Logs.

You can add permissions to the role later, or swap it out for a different role that's specific to a single function.

Tools and libraries for Rust

- [AWS SDK for Rust](#): The AWS SDK for Rust is in developer preview release. Do not use it in production as it is subject to breaking changes.
- [Rust runtime client for Lambda](#): The Rust runtime client is an experimental package. It is subject to breaking changes and not recommended for production.
- [Cargo Lambda](#): This library provides a command line application to work with Lambda functions built with Rust.
- [Lambda HTTP](#): This library provides a wrapper to work with HTTP events.
- [Lambda Extension](#): This library provides support to write Lambda Extensions with Rust.
- [AWS Lambda Events](#): This library provides type definitions for common event source integrations.

Sample Lambda applications for Rust

- [Basic Lambda function](#): A Rust function that shows how to process basic events.
- [Lambda function with error handling](#): A Rust function that shows how to handle custom Rust errors in Lambda.
- [Lambda function with shared resources](#): A Rust project that initializes shared resources before creating the Lambda function.
- [Lambda HTTP events](#): A Rust function that handles HTTP events.
- [Lambda HTTP events with CORS headers](#): A Rust function that uses Tower to inject CORS headers.
- [Lambda REST API](#): A REST API that uses Axum and Diesel to connect to a PostgreSQL database.
- [Serverless Rust demo](#): A Rust project that shows the use of Lambda's Rust libraries, logging, environment variables, and the AWS SDK.
- [Basic Lambda Extension](#): A Rust extension that shows how to process basic extension events.
- [Lambda Logs Amazon Kinesis Data Firehose Extension](#): A Rust extension that shows how to send Lambda logs to Kinesis Data Firehose.

Topics

- [Lambda function handler in Rust \(p. 545\)](#)
- [Lambda context object in Rust \(p. 547\)](#)
- [Processing HTTP events with Rust \(p. 548\)](#)
- [Deploy Rust Lambda functions with .zip file archives \(p. 550\)](#)
- [Lambda function logging in Rust \(p. 553\)](#)
- [Lambda function errors in Rust \(p. 555\)](#)

Lambda function handler in Rust

Note

The [Rust runtime client](#) is an experimental package. It is subject to change and intended only for evaluation purposes.

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. When the handler exits or returns a response, it becomes available to handle another event.

Write your Lambda function code as a Rust executable. Implement the handler function code and a main function and include the following:

- The [lambda_runtime](#) crate from crates.io, which implements the Lambda programming model for Rust.
- Include [Tokio](#) in your dependencies. The [Rust runtime client for Lambda](#) uses Tokio to handle asynchronous calls.

Example — Rust handler that processes JSON events

The following example uses the [serde_json](#) crate to process basic JSON events:

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};

async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    Ok(json!({ "message": format!("Hello, {}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

Note the following:

- `use`: Imports the libraries that your Lambda function requires.
- `async fn main`: The entry point that runs the Lambda function code. The Rust runtime client uses [Tokio](#) as an async runtime, so you must annotate the main function with `#[tokio::main]`.
- `async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error>`: This is the Lambda handler signature. It includes the code that runs when the function is invoked.
 - `LambdaEvent<Value>`: This is a generic type that describes the event received by the Lambda runtime as well as the [Lambda function context \(p. 547\)](#).
 - `Result<Value, Error>`: The function returns a Result type. If the function is successful, the result is a JSON value. If the function is not successful, the result is an error.

Using shared state

You can declare shared variables that are independent of your Lambda function's handler code. These variables can help you load state information during the [Init phase \(p. 15\)](#), before your function receives any events.

Example — Share Amazon S3 client across function instances

Note the following:

- `use aws_sdk_s3::Client`: This example requires you to add `aws-sdk-s3 = "0.26.0"` to the list of dependencies in your `Cargo.toml` file.
- `aws_config::from_env`: This example requires you to add `aws-config = "0.55.1"` to the list of dependencies in your `Cargo.toml` file.

```
use aws_sdk_s3::Client;
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde::{Deserialize, Serialize};

#[derive(Deserialize)]
struct Request {
    bucket: String,
}

#[derive(Serialize)]
struct Response {
    keys: Vec<String>,
}

async fn handler(client: &Client, event: LambdaEvent<Request>) -> Result<Response, Error> {
    let bucket = event.payload.bucket;
    let objects = client.list_objects_v2().bucket(bucket).send().await?;
    let keys = objects
        .contents()
        .map(|s| s.iter().flat_map(|o| o.key().map(String::from)).collect())
        .unwrap_or_default();
    Ok(Response { keys })
}

#[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
}
```

Lambda context object in Rust

Note

The [Rust runtime client](#) is an experimental package. It is subject to change and intended only for evaluation purposes.

When Lambda runs your function, it adds a context object to the LambdaEvent that the [handler \(p. 545\)](#) receives. This object provides properties with information about the invocation, function, and execution environment.

Context properties

- `request_id`: The AWS request ID generated by the Lambda service.
- `deadline`: The execution deadline for the current invocation in milliseconds.
- `invoked_function_arn`: The Amazon Resource Name (ARN) of the Lambda function being invoked.
- `xray_trace_id`: The AWS X-Ray trace ID for the current invocation.
- `client_content`: The client context object sent by the AWS mobile SDK. This field is empty unless the function is invoked using an AWS mobile SDK.
- `identity`: The Amazon Cognito identity that invoked the function. This field is empty unless the invocation request to the Lambda APIs was made using AWS credentials issued by Amazon Cognito identity pools.
- `env_config`: The Lambda function configuration from the local environment variables. This property includes information such as the function name, memory allocation, version, and log streams.

Accessing invoke context information

Lambda functions have access to metadata about their environment and the invocation request. The LambdaEvent object that your function handler receives includes the context metadata:

```
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
}
```

Processing HTTP events with Rust

Note

The [Rust runtime client](#) is an experimental package. It is subject to change and intended only for evaluation purposes.

Amazon API Gateway APIs, Application Load Balancers, and [Lambda function URLs \(p. 166\)](#) can send HTTP events to Lambda. You can use the [aws_lambda_events](#) crate from crates.io to process events from these sources.

Example — Handle API Gateway proxy request

Note the following:

- use `aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse}`: The [aws_lambda_events](#) crate includes many Lambda events. To reduce compilation time, use feature flags to activate the events you need. Example:
`aws_lambda_events = { version = "0.8.3", default-features = false, features = ["apigw"] }`.
- use `http::HeaderMap`: This import requires you to add the [http](#) crate to your dependencies.

```
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: Some(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
}
```

The [Rust runtime client for Lambda](#) also provides an abstraction over these event types that allows you to work with native HTTP types, regardless of which service sends the events. The following code is equivalent to the previous example, and it works out of the box with Lambda function URLs, Application Load Balancers, and API Gateway.

Note

The [lambda_http](#) crate uses the [lambda_runtime](#) crate underneath. You don't have to import `lambda_runtime` separately.

Example — Handle HTTP requests

```
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
}
```

For another example of how to use `lambda_http`, see the [http-axum code sample](#) on the AWS Labs GitHub repository.

Sample HTTP Lambda events for Rust

- [Lambda HTTP events](#): A Rust function that handles HTTP events.
- [Lambda HTTP events with CORS headers](#): A Rust function that uses Tower to inject CORS headers.
- [Lambda HTTP events with shared resources](#): A Rust function that uses shared resources initialized before the function handler is created.

Deploy Rust Lambda functions with .zip file archives

Note

The [Rust runtime client](#) is an experimental package. It is subject to change and intended only for evaluation purposes.

This page describes how to compile your Rust function, and then deploy the compiled binary to AWS Lambda using [Cargo Lambda](#). It also shows how to deploy the compiled binary with the AWS Command Line Interface and the AWS Serverless Application Model CLI.

Sections

- [Prerequisites \(p. 550\)](#)
- [Building Rust functions on macOS, Windows, or Linux \(p. 550\)](#)
- [Deploying the Rust function binary with Cargo Lambda \(p. 551\)](#)
- [Invoking your Rust function with Cargo Lambda \(p. 552\)](#)

Prerequisites

- [Rust](#)
- [AWS Command Line Interface \(AWS CLI\) version 2](#)

Building Rust functions on macOS, Windows, or Linux

The following steps demonstrate how to create the project for your first Lambda function with Rust and compile it with [Cargo Lambda](#).

1. Install Cargo Lambda, a Cargo subcommand, that compiles Rust functions for Lambda on macOS, Windows, and Linux.

To install Cargo Lambda on any system that has Python 3 installed, use pip:

```
pip3 install cargo-lambda
```

To install Cargo Lambda on macOS or Linux, use Homebrew:

```
brew tap cargo-lambda/cargo-lambda
brew install cargo-lambda
```

To install Cargo Lambda on Windows, use [Scoop](#):

```
scoop bucket add cargo-lambda
scoop install cargo-lambda/cargo-lambda
```

For other options, see [Installation](#) in the Cargo Lambda documentation.

2. Create the package structure. This command creates some basic function code in `src/main.rs`. You can use this code for testing or replace it with your own.

```
cargo lambda new my-function
```

3. Inside the package's root directory, run the [build](#) subcommand to compile the code in your function.

```
cargo lambda build --release
```

(Optional) If you want to use AWS Graviton2 on Lambda, add the `--arm64` flag to compile your code for ARM CPUs.

```
cargo lambda build --release --arm64
```

4. Before deploying your Rust function, configure AWS credentials on your machine.

```
aws configure
```

Deploying the Rust function binary with Cargo Lambda

Use the [deploy](#) subcommand to deploy the compiled binary to Lambda. This command creates an [execution role \(p. 816\)](#) and then creates the Lambda function. To specify an existing execution role, use the [--iam-role](#) flag.

```
cargo lambda deploy my-function
```

Deploying your Rust function binary with the AWS CLI

You can also deploy your binary with the AWS CLI.

1. Use the [build](#) subcommand to build the .zip deployment package.

```
cargo lambda build --release --output-format zip
```

2. Deploy the .zip package to Lambda. For `--role`, specify the ARN of the execution role.

```
aws lambda create-function --function-name my-function \
    --runtime provided.al2 \
    --role arn:aws:iam::111122223333:role/lambda-role \
    --handler rust.handler \
    --zip-file fileb://target/lambda/my-function/bootstrap.zip
```

Deploying your Rust function binary with the AWS SAM CLI

You can also deploy your binary with the AWS SAM CLI.

1. Create an AWS SAM template with the resource and property definition. For more information, see [AWS::Serverless::Function](#) in the *AWS Serverless Application Model Developer Guide*.

Example SAM resource and property definition for a Rust binary

```
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.al2
Outputs:
  RustFunction:
    Description: "Lambda Function ARN"
    Value: !GetAtt RustFunction.Arn
```

2. Use the [build](#) subcommand to compile the function.

```
cargo lambda build --release
```

3. Use the [sam deploy](#) command to deploy the function to Lambda.

```
sam deploy --guided
```

For more information about building Rust functions with the AWS SAM CLI, see [Building Rust Lambda functions with Cargo Lambda](#) in the *AWS Serverless Application Model Developer Guide*.

Invoking your Rust function with Cargo Lambda

Use the [invoke](#) subcommand to test your function with a payload.

```
cargo lambda invoke --remote --data-ascii '{"command": "Hello world"}' my-function
```

Invoking your Rust function with the AWS CLI

You can also use the AWS CLI to invoke the function.

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"command": "Hello world"}' /tmp/out.txt
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

Lambda function logging in Rust

Note

The [Rust runtime client](#) is an experimental package. It is subject to change and intended only for evaluation purposes.

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#). This page describes how to produce log output from your Lambda function's code.

Creating a function that writes logs

To output logs from your function code, you can use any logging function that writes to `stdout` or `stderr`, such as the `println!` macro. The following example uses `println!` to print a message when the function handler starts and before it finishes.

```
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, {}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

Advanced logging with the Tracing crate

[Tracing](#) is a framework for instrumenting Rust programs to collect structured, event-based diagnostic information. This framework provides utilities to customize logging output levels and formats, like creating structured JSON log messages. To use this framework, you must initialize a subscriber before implementing the function handler. Then, you can use tracing macros like `debug`, `info`, and `error`, to specify the level of logging that you want for each scenario.

Example — Using the Tracing crate

Note the following:

- `tracing_subscriber::fmt().json()`: When this option is included, logs are formatted in JSON. To use this option, you must include the `json` feature in the `tracing-subscriber` dependency (for example, `tracing-subscriber = { version = "0.3.11", features = ["json"] }`).
- `#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]`: This annotation generates a span every time the handler is invoked. The span adds the request ID to each log line.
- `{ %first_name }`: This construct adds the `first_name` field to the log line where it's used. The value for this field corresponds to the variable with the same name.

```
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
}
```

When this Rust function is invoked, it prints two log lines similar to the following:

```
{"level": "INFO", "fields": {"message": "Rust function invoked"}, "spans": [{"req_id": "45daaa7-1a72-470c-9a62-e79860044bb5", "name": "handler"}]}
{"level": "INFO", "fields": {"message": "Rust function responds to event", "first_name": "David"}, "spans": [{"req_id": "45daaa7-1a72-470c-9a62-e79860044bb5", "name": "handler"}]}
```

Lambda function errors in Rust

Note

The [Rust runtime client](#) is an experimental package. It is subject to change and intended only for evaluation purposes.

When your code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output. The [Rust runtime client](#) also writes the error in the log. The error appears in Amazon CloudWatch Logs by default. This page demonstrates how to return errors in your Lambda function's output.

Creating a function that returns errors

The following code sample shows a Lambda function that returns an error. The Rust Runtime handles this error directly.

Example

```
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde_json::{json, Value};
async fn handler(_event: LambdaEvent<Value>) -> Result<Value, String> {
    Err("something went wrong!".into())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

This code returns the following error payload:

```
{
    "errorType": "&alloc::string::String",
    "errorMessage": "something went wrong!"
}
```

For a more advanced error handling example, see the [sample application](#) in the AWS Labs GitHub repository.

Using AWS Lambda with other services

AWS Lambda integrates with other AWS services to invoke functions or take other actions. These are some common use cases:

- Invoke a function in response to resource lifecycle events, such as with Amazon Simple Storage Service (Amazon S3). For more information, see [Using AWS Lambda with Amazon S3 \(p. 741\)](#).
- Respond to incoming HTTP requests. For more information, see [Tutorial: Using Lambda with API Gateway \(p. 568\)](#).
- Consume events from a queue. For more information, see [Using Lambda with Amazon SQS \(p. 778\)](#).
- Run a function on a schedule. For more information, see [Using AWS Lambda with Amazon EventBridge \(CloudWatch Events\) \(p. 591\)](#).

Depending on which service you're using with Lambda, the invocation generally works in one of two ways. An event drives the invocation or Lambda polls a queue or data stream and invokes the function in response to activity in the queue or data stream. Lambda integrates with Amazon Elastic File System and AWS X-Ray in a way that doesn't involve invoking functions.

For more information, see [Event-driven invocation \(p. 558\)](#) and [Lambda polling \(p. 558\)](#). Or, look up the service that you want to work with in the following section to find a link to information about using that service with Lambda.

You can also use Lambda functions to interact programmatically with other AWS services using one of the AWS Software Development Kits (SDKs). For example, you can have a Lambda function create an Amazon S3 bucket or write data to a DynamoDB table using an API call from within your function. To learn more about AWS SDKs, see [Tools to build on AWS](#).

Listing of services and links to more information

Find the service that you want to work with in the following table, to determine which method of invocation you should use. Follow the link from the service name to find information about how to set up the integration between the services. These topics also include example events that you can use to test your function.

Tip

Entries in this table are alphabetical by service name, excluding the "Amazon" or "AWS" prefix. You can also use your browser's search functionality to find your service in the list.

Service	Method of invocation
Amazon Alexa (p. 561)	Event-driven; synchronous invocation
Amazon Managed Streaming for Apache Kafka (p. 716)	Lambda polling
Self-managed Apache Kafka (p. 671)	Lambda polling
Amazon API Gateway (p. 562)	Event-driven; synchronous invocation
AWS CloudFormation (p. 598)	Event-driven; asynchronous invocation

Service	Method of invocation
Amazon CloudFront (Lambda@Edge) (p. 601)	Event-driven; synchronous invocation
Amazon EventBridge (CloudWatch Events) (p. 591)	Event-driven; asynchronous invocation
Amazon CloudWatch Logs (p. 597)	Event-driven; asynchronous invocation
AWS CodeCommit (p. 603)	Event-driven; asynchronous invocation
AWS CodePipeline (p. 604)	Event-driven; asynchronous invocation
Amazon Cognito (p. 607)	Event-driven; synchronous invocation
AWS Config (p. 608)	Event-driven; asynchronous invocation
Amazon Connect (p. 609)	Event-driven; synchronous invocation
Amazon DynamoDB (p. 635)	Lambda polling
Amazon Elastic File System (p. 666)	Special integration
Elastic Load Balancing (Application Load Balancer) (p. 664)	Event-driven; synchronous invocation
AWS IoT (p. 668)	Event-driven; asynchronous invocation
AWS IoT Events (p. 669)	Event-driven; asynchronous invocation
Amazon Kinesis (p. 684)	Lambda polling
Amazon Kinesis Data Firehose (p. 683)	Event-driven; synchronous invocation
Amazon Lex (p. 706)	Event-driven; synchronous invocation
Amazon MQ (p. 708)	Lambda polling
Amazon Simple Email Service (p. 768)	Event-driven; asynchronous invocation
Amazon Simple Notification Service (p. 770)	Event-driven; asynchronous invocation
Amazon Simple Queue Service (p. 778)	Lambda polling
Amazon Simple Storage Service (Amazon S3) (p. 741)	Event-driven; asynchronous invocation
Amazon Simple Storage Service Batch (p. 764)	Event-driven; synchronous invocation
Secrets Manager (p. 767)	Event-driven; synchronous invocation
AWS X-Ray (p. 807)	Special integration

Event-driven invocation

Some services generate events that can invoke your Lambda function. For more information about designing these types of architectures , see [Event driven architectures](#) in the *Lambda operator guide*.

When you implement an event-driven architecture, you grant the event-generating service permission to invoke your function in the function's [resource-based policy \(p. 832\)](#). Then you configure that service to generate events that invoke your function.

The events are data structured in JSON format. The JSON structure varies depending on the service that generates it and the event type, but they all contain the data that the function needs to process the event.

Lambda converts the event document into an object and passes it to your [function handler \(p. 9\)](#). For compiled languages, Lambda provides definitions for event types in a library. For more information, see the topic about building functions with your language: [Building Lambda functions with C# \(p. 487\)](#), [Building Lambda functions with Go \(p. 453\)](#), [Building Lambda functions with Java \(p. 386\)](#), or [Building Lambda functions with PowerShell \(p. 528\)](#).

Depending on the service, the event-driven invocation can be synchronous or asynchronous.

- For synchronous invocation, the service that generates the event waits for the response from your function. That service defines the data that the function needs to return in the response. The service controls the error strategy, such as whether to retry on errors. For more information, see [the section called "Synchronous invocation" \(p. 120\)](#).
- For asynchronous invocation, Lambda queues the event before passing it to your function. When Lambda queues the event, it immediately sends a success response to the service that generated the event. After the function processes the event, Lambda doesn't return a response to the event-generating service. For more information, see [the section called "Asynchronous invocation" \(p. 123\)](#).

For more information about how Lambda manages error handling for synchronously and asynchronously invoked functions, see [the section called "Error handling" \(p. 161\)](#).

Lambda polling

For services that generate a queue or data stream, you set up an [event source mapping \(p. 131\)](#) in Lambda to have Lambda poll the queue or a data stream.

When you implement a Lambda polling architecture, you grant Lambda permission to access the other service in the function's [execution role \(p. 816\)](#). Lambda reads data from the other service, creates an event, and invokes your function.

Common Lambda application types and use cases

Lambda functions and triggers are the core components of building applications on AWS Lambda. A Lambda function is the code and runtime that process events, while a trigger is the AWS service or application that invokes the function. To illustrate, consider the following scenarios:

- **File processing** – Suppose you have a photo sharing application. People use your application to upload photos, and the application stores these user photos in an Amazon S3 bucket. Then, your application creates a thumbnail version of each user's photos and displays them on the user's profile page. In this scenario, you may choose to create a Lambda function that creates a thumbnail automatically. Amazon S3 is one of the supported AWS event sources that can publish *object-created events* and invoke your Lambda function. Your Lambda function code can read the photo object from the S3 bucket, create a thumbnail version, and then save it in another S3 bucket.
- **Data and analytics** – Suppose you are building an analytics application and storing raw data in a DynamoDB table. When you write, update, or delete items in a table, DynamoDB streams can publish item update events to a stream associated with the table. In this case, the event data provides the item key, event name (such as insert, update, and delete), and other relevant details. You can write a Lambda function to generate custom metrics by aggregating raw data.
- **Websites** – Suppose you are creating a website and you want to host the backend logic on Lambda. You can invoke your Lambda function over HTTP using Amazon API Gateway as the HTTP endpoint. Now, your web client can invoke the API, and then API Gateway can route the request to Lambda.
- **Mobile applications** – Suppose you have a custom mobile application that produces events. You can create a Lambda function to process events published by your custom application. For example, you can configure a Lambda function to process the clicks within your custom mobile application.

AWS Lambda supports many AWS services as event sources. For more information, see [Using AWS Lambda with other services \(p. 556\)](#). When you configure these event sources to trigger a Lambda function, the Lambda function is invoked automatically when events occur. You define *event source mapping*, which is how you identify what events to track and which Lambda function to invoke.

The following are introductory examples of event sources and how the end-to-end experience works.

Example 1: Amazon S3 pushes events and invokes a Lambda function

Amazon S3 can publish events of different types, such as PUT, POST, COPY, and DELETE object events on a bucket. Using the bucket notification feature, you can configure an event source mapping that directs Amazon S3 to invoke a Lambda function when a specific type of event occurs.

The following is a typical sequence:

1. The user creates an object in a bucket.
2. Amazon S3 detects the object created event.
3. Amazon S3 invokes your Lambda function using the permissions provided by the [execution role \(p. 816\)](#).
4. AWS Lambda runs the Lambda function, specifying the event as a parameter.

You configure Amazon S3 to invoke your function as a bucket notification action. To grant Amazon S3 permission to invoke the function, update the function's [resource-based policy \(p. 832\)](#).

Example 2: AWS Lambda pulls events from a Kinesis stream and invokes a Lambda function

For poll-based event sources, AWS Lambda polls the source and then invokes the Lambda function when records are detected on that source.

- [CreateEventSourceMapping \(p. 1153\)](#)
- [UpdateEventSourceMapping \(p. 1356\)](#)

The following steps describe how a custom application writes records to a Kinesis stream:

1. The custom application writes records to a Kinesis stream.
2. AWS Lambda continuously polls the stream, and invokes the Lambda function when the service detects new records on the stream. AWS Lambda knows which stream to poll and which Lambda function to invoke based on the event source mapping you create in Lambda.
3. The Lambda function is invoked with the incoming event.

When working with stream-based event sources, you create event source mappings in AWS Lambda. Lambda reads items from the stream and invokes the function synchronously. You don't need to grant Lambda permission to invoke the function, but it does need permission to read from the stream.

Using AWS Lambda with Alexa

You can use Lambda functions to build services that give new skills to Alexa, the Voice assistant on Amazon Echo. The Alexa Skills Kit provides the APIs, tools, and documentation to create these new skills, powered by your own services running as Lambda functions. Amazon Echo users can access these new skills by asking Alexa questions or making requests.

The Alexa Skills Kit is available on GitHub.

- [Alexa Skills Kit SDK for Java](#)
- [Alexa Skills Kit SDK for Node.js](#)
- [Alexa Skills Kit SDK for Python](#)

Example Alexa smart home event

```
{  
  "header": {  
    "payloadVersion": "1",  
    "namespace": "Control",  
    "name": "SwitchOnOffRequest"  
  },  
  "payload": {  
    "switchControlAction": "TURN_ON",  
    "appliance": {  
      "additionalApplianceDetails": {  
        "key2": "value2",  
        "key1": "value1"  
      },  
      "applianceId": "sampleId"  
    },  
    "accessToken": "sampleAccessToken"  
  }  
}
```

For more information, see [Host a custom skill as an AWS Lambda Function](#) in the *Build Skills with the Alexa Skills Kit* guide.

Using AWS Lambda with Amazon API Gateway

You can create a web API with an HTTP endpoint for your Lambda function by using Amazon API Gateway. API Gateway provides tools for creating and documenting web APIs that route HTTP requests to Lambda functions. You can secure access to your API with authentication and authorization controls. Your APIs can serve traffic over the internet or can be accessible only within your VPC.

Resources in your API define one or more methods, such as GET or POST. Methods have an integration that routes requests to a Lambda function or another integration type. You can define each resource and method individually, or use special resource and method types to match all requests that fit a pattern. A *proxy resource* catches all paths beneath a resource. The ANY method catches all HTTP methods.

This section explains general information on how to choose an API type, add an endpoint to your Lambda function, and information on events, permissions, responses, and error handling.

Sections

- [Adding an endpoint to your Lambda function \(p. 562\)](#)
- [Proxy integration \(p. 562\)](#)
- [Event format \(p. 563\)](#)
- [Response format \(p. 564\)](#)
- [Permissions \(p. 564\)](#)
- [Handling errors with an API Gateway API \(p. 566\)](#)
- [Choosing an API type \(p. 567\)](#)
- [Sample applications \(p. 568\)](#)
- [Tutorial: Using Lambda with API Gateway \(p. 568\)](#)
- [AWS SAM template for an API Gateway application \(p. 582\)](#)

Adding an endpoint to your Lambda function

To add a public endpoint to your Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Under **Function overview**, choose **Add trigger**.
4. Select **API Gateway**.
5. Choose **Create an API or Use an existing API**.
 - a. **New API:** For **API type**, choose **HTTP API**. For more information, see [API types \(p. 567\)](#).
 - b. **Existing API:** Select the API from the dropdown menu or enter the API ID (for example, r3pmxmplak).
6. For **Security**, choose **Open**.
7. Choose **Add**.

Proxy integration

API Gateway APIs are comprised of stages, resources, methods, and integrations. The stage and resource determine the path of the endpoint:

API path format

- /prod/ – The prod stage and root resource.
- /prod/user – The prod stage and user resource.
- /dev/{proxy+} – Any route in the dev stage.
- / – (HTTP APIs) The default stage and root resource.

A Lambda integration maps a path and HTTP method combination to a Lambda function. You can configure API Gateway to pass the body of the HTTP request as-is (custom integration), or to encapsulate the request body in a document that includes all of the request information including headers, resource, path, and method.

Event format

Amazon API Gateway invokes your function [synchronously \(p. 120\)](#) with an event that contains a JSON representation of the HTTP request. For a custom integration, the event is the body of the request. For a proxy integration, the event has a defined structure. The following example shows a proxy event from an API Gateway REST API.

Example [event.json](#) API Gateway proxy event (REST API)

```
{
    "resource": "/",
    "path": "/",
    "httpMethod": "GET",
    "requestContext": {
        "resourcePath": "/",
        "httpMethod": "GET",
        "path": "/Prod/",
        ...
    },
    "headers": {
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9",
        "accept-encoding": "gzip, deflate, br",
        "Host": "70ixmpl4fl.execute-api.us-east-2.amazonaws.com",
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.132 Safari/537.36",
        "X-Amzn-Trace-Id": "Root=1-5e66d96f-7491f09xmpl79d18acf3d050",
        ...
    },
    "multiValueHeaders": {
        "accept": [
            "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9"
        ],
        "accept-encoding": [
            "gzip, deflate, br"
        ],
        ...
    },
    "queryStringParameters": null,
    "multiValueQueryStringParameters": null,
    "pathParameters": null,
    "stageVariables": null,
    "body": null,
    "isBase64Encoded": false
}
```

Response format

API Gateway waits for a response from your function and relays the result to the caller. For a custom integration, you define an integration response and a method response to convert the output from the function to an HTTP response. For a proxy integration, the function must respond with a representation of the response in a specific format.

The following example shows a response object from a Node.js function. The response object represents a successful HTTP response that contains a JSON document.

Example [index.js](#) – Proxy integration response object (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}"
}
```

The Lambda runtime serializes the response object into JSON and sends it to the API. The API parses the response and uses it to create an HTTP response, which it then sends to the client that made the original request.

Example HTTP response

```
< 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
}
```

Permissions

Amazon API Gateway gets permission to invoke your function from the function's [resource-based policy \(p. 832\)](#). You can grant invoke permission to an entire API, or grant limited access to a stage, resource, or method.

When you add an API to your function by using the Lambda console, using the API Gateway console, or in an AWS SAM template, the function's resource-based policy is updated automatically. The following is an example function policy.

Example function policy

```
{
    "Version": "2012-10-17",
```

```
"Id": "default",
"Statement": [
  {
    "Sid": "nodejs-apig-functiongetEndpointPermissionProd-BWDBXMPLEX2F",
    "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/*"
      }
    }
  }
]
```

You can manage function policy permissions manually with the following API operations:

- [AddPermission](#) (p. 1141)
 - [RemovePermission](#) (p. 1343)
 - [GetPolicy](#) (p. 1252)

To grant invocation permission to an existing API, use the `add-permission` command.

```
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/"
```

You should see the following output:

```
{  
    "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

Note If your function and API are in different regions, the region identifier in the source ARN must match the region of the function, not the region of the API. When API Gateway invokes a function, it uses a resource ARN that is based on the ARN of the API, but modified to match the function's region.

The source ARN in this example grants permission to an integration on the GET method of the root resource in the default stage of an API, with ID mn1xmpli7. You can use an asterisk in the source ARN to grant permissions to multiple stages, methods, or resources.

Resource patterns

- `mnh1xmplj7/*/GET/*` – GET method on all resources in all stages.

- `mnh1xmpli7/prod/ANY/user` – ANY method on the user resource in the prod stage.
- `mnh1xmpli7/*/*/*` – Any method on all resources in all stages.

For details on viewing the policy and removing statements, see [Cleaning up resource-based policies \(p. 836\)](#).

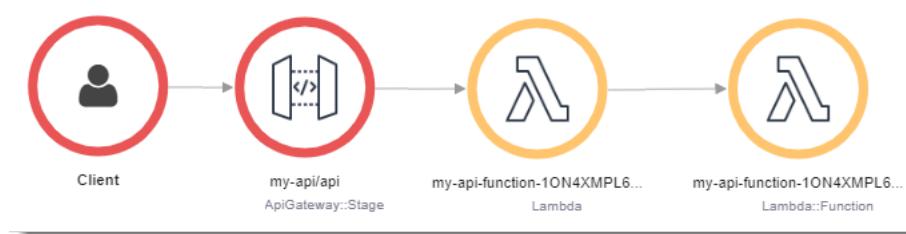
Handling errors with an API Gateway API

API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502. In both cases, the body of the response from API Gateway is `{"message": "Internal server error"}`.

Note

API Gateway does not retry any Lambda invocations. If Lambda returns an error, API Gateway returns an error response to the client.

The following example shows an X-Ray trace map for a request that resulted in a function error and a 502 from API Gateway. The client receives the generic error message.



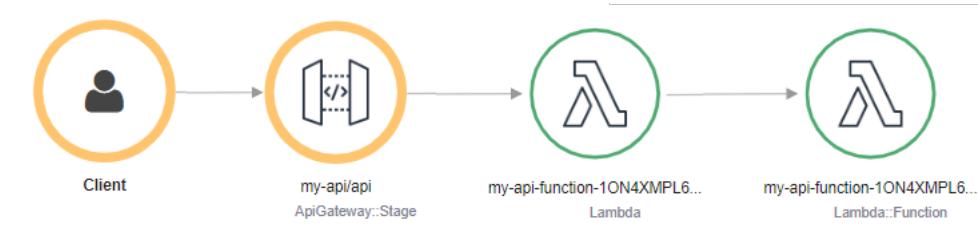
To customize the error response, you must catch errors in your code and format a response in the required format.

Example `index.js` – Error formatting

```

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 converts this response into an HTTP error with a custom status code and body. In the trace map, the function node is green because it handled the error.



Choosing an API type

API Gateway supports three types of APIs that invoke Lambda functions:

- **HTTP API** – A lightweight, low-latency RESTful API.
 - **REST API** – A customizable, feature-rich RESTful API.
 - **WebSocket API** – A web API that maintains persistent connections with clients for full-duplex communication.

HTTP APIs and REST APIs are both RESTful APIs that process HTTP requests and return responses. HTTP APIs are newer and are built with the API Gateway version 2 API. The following features are new for HTTP APIs:

HTTP API features

- **Automatic deployments** – When you modify routes or integrations, changes deploy automatically to stages that have automatic deployment enabled.
 - **Default stage** – You can create a default stage (`$default`) to serve requests at the root path of your API's URL. For named stages, you must include the stage name at the beginning of the path.
 - **CORS configuration** – You can configure your API to add CORS headers to outgoing responses, instead of adding them manually in your function code.

REST APIs are the classic RESTful APIs that API Gateway has supported since launch. REST APIs currently have more customization, integration, and management features.

REST API features

- **Integration types** – REST APIs support custom Lambda integrations. With a custom integration, you can send just the body of the request to the function, or apply a transform template to the request body before sending it to the function.
 - **Access control** – REST APIs support more options for authentication and authorization.
 - **Monitoring and tracing** – REST APIs support AWS X-Ray tracing and additional logging options.

For a detailed comparison, see [Choosing between HTTP APIs and REST APIs](#) in the *API Gateway Developer Guide*.

WebSocket APIs also use the API Gateway version 2 API and support a similar feature set. Use a WebSocket API for applications that benefit from a persistent connection between the client and API. WebSocket APIs provide full-duplex communication, which means that both the client and the API can send messages continuously without waiting for a response.

HTTP APIs support a simplified event format (version 2.0). The following example shows an event from an HTTP API.

Example [event-v2.json](#) – API Gateway proxy event (HTTP API)

```
{  
    "version": "2.0",  
    "routeKey": "ANY /nodejs-apig-function-1G3XmplZxVXYI",  
    "rawPath": "/default/nodejs-apig-function-1G3XmplZxVXYI",  
    "rawQueryString": "",  
    "cookies": [  
        "s_fid=7AABXmpl1AFD9BBF-0643Xmpl09956DE2".  
    ]  
}
```

```
        "regStatus=pre-register"
],
"headers": {
    "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9",
    "accept-encoding": "gzip, deflate, br",
    ...
},
"requestContext": {
    "accountId": "123456789012",
    "apiId": "r3pmxmplak",
    "domainName": "r3pmxmplak.execute-api.us-east-2.amazonaws.com",
    "domainPrefix": "r3pmxmplak",
    "http": {
        "method": "GET",
        "path": "/default/nodejs-apig-function-1G3XMPLZXVXYI",
        "protocol": "HTTP/1.1",
        "sourceIp": "205.255.255.176",
        "userAgent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.132 Safari/537.36"
    },
    "requestId": "JKJaXmPLvHcEsha=",
    "routeKey": "ANY /nodejs-apig-function-1G3XMPLZXVXYI",
    "stage": "default",
    "time": "10/Mar/2020:05:16:23 +0000",
    "timeEpoch": 1583817383220
},
"isBase64Encoded": true
}
```

For more information, see [AWS Lambda integrations](#) in the API Gateway Developer Guide.

Sample applications

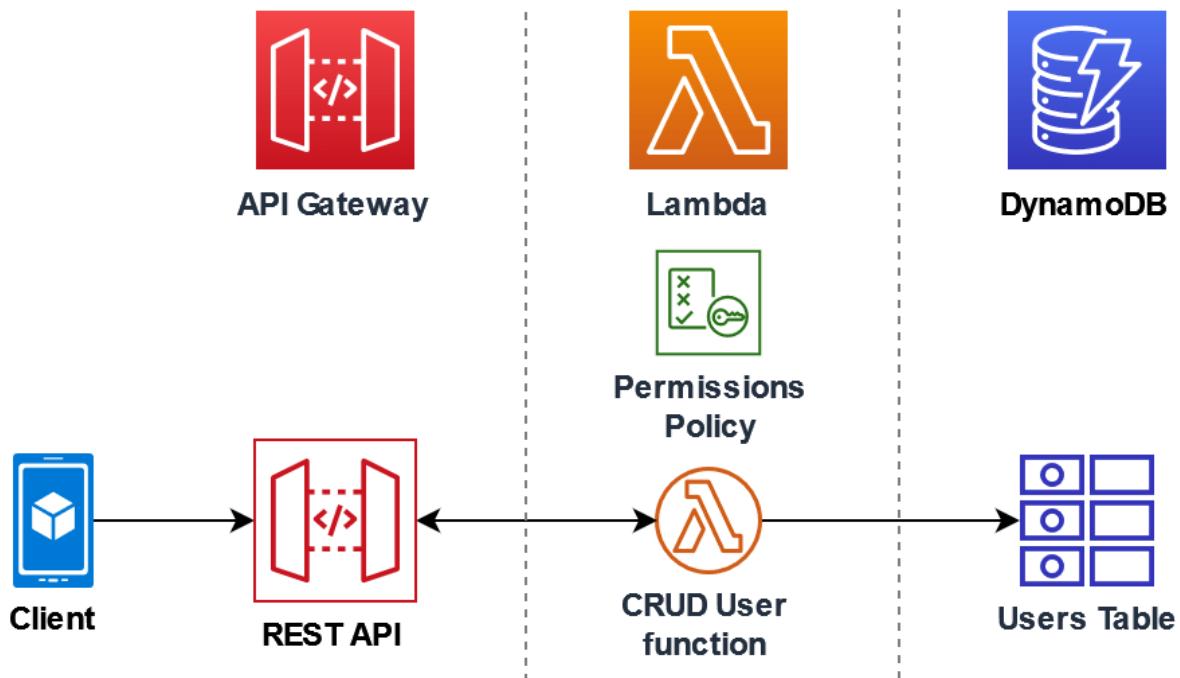
The GitHub repository for this guide provides the following sample application for API Gateway.

- [API Gateway with Node.js](#) – A function with an AWS SAM template that creates a REST API that has AWS X-Ray tracing enabled. It includes scripts for deploying, invoking the function, testing the API, and cleanup.

Lambda also provides [blueprints \(p. 35\)](#) and [templates \(p. 36\)](#) that you can use to create an API Gateway application in the Lambda console.

Tutorial: Using Lambda with API Gateway

In this tutorial, you create a REST API through which you invoke a Lambda function using an HTTP request. Your Lambda function will perform create, read, update, and delete (CRUD) operations on a DynamoDB table. This function is provided here for demonstration, but you will learn to configure an API Gateway REST API that can invoke any Lambda function.



Using API Gateway provides users with a secure HTTP endpoint to invoke your Lambda function and can help manage large volumes of calls to your function by throttling traffic and automatically validating and authorizing API calls. API Gateway also provides flexible security controls using AWS Identity and Access Management (IAM) and Amazon Cognito. This is useful for use cases where advance authorization is required for calls to your application.

To complete this tutorial, you will go through the following stages:

1. Create and configure a Lambda function in Python or Node.js to perform operations on a DynamoDB table.
2. Create a REST API in API Gateway to connect to your Lambda function.
3. Create a DynamoDB table and test it with your Lambda function in the console.
4. Deploy your API and test the full setup using curl in a terminal.

By completing these stages, you will learn how to use API Gateway to create an HTTP endpoint that can securely invoke a Lambda function at any scale. You will also learn how to deploy your API, and how to test it in the console and by sending an HTTP request using a terminal.

A sample AWS Serverless Application Model (AWS SAM) template for the Lambda application you create in this tutorial is also available. See [AWS SAM template for an API Gateway application \(p. 582\)](#).

Sections

- [Prerequisites \(p. 570\)](#)
- [Create a permissions policy \(p. 571\)](#)
- [Create an execution role \(p. 572\)](#)
- [Create the function \(p. 573\)](#)
- [Invoke the function using the AWS CLI \(p. 576\)](#)
- [Create a REST API using API Gateway \(p. 577\)](#)
- [Create a resource on your REST API \(p. 577\)](#)

- [Create an HTTP POST method \(p. 578\)](#)
- [Create a DynamoDB table \(p. 578\)](#)
- [Test the integration of API Gateway, Lambda, and DynamoDB \(p. 579\)](#)
- [Deploy the API \(p. 580\)](#)
- [Use curl to invoke your function using HTTP requests \(p. 581\)](#)
- [Clean up your resources \(optional\) \(p. 104\)](#)

Prerequisites

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, [assign administrative access to an administrative user](#), and use only the root user to perform [tasks that require root user access](#).

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to <https://aws.amazon.com/> and choosing **My Account**.

Create an administrative user

After you sign up for an AWS account, create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the [AWS Management Console](#) as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

For help signing in by using root user, see [Signing in as the root user](#) in the *AWS Sign-In User Guide*.
2. Turn on multi-factor authentication (MFA) for your root user.

For instructions, see [Enable a virtual MFA device for your AWS account root user \(console\)](#) in the *IAM User Guide*.

Create an administrative user

- For your daily administrative tasks, grant administrative access to an administrative user in AWS IAM Identity Center (successor to AWS Single Sign-On).

For instructions, see [Getting started](#) in the *AWS IAM Identity Center (successor to AWS Single Sign-On) User Guide*.

Sign in as the administrative user

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

For help signing in using an IAM Identity Center user, see [Signing in to the AWS access portal](#) in the [AWS Sign-In User Guide](#).

Install the AWS Command Line Interface

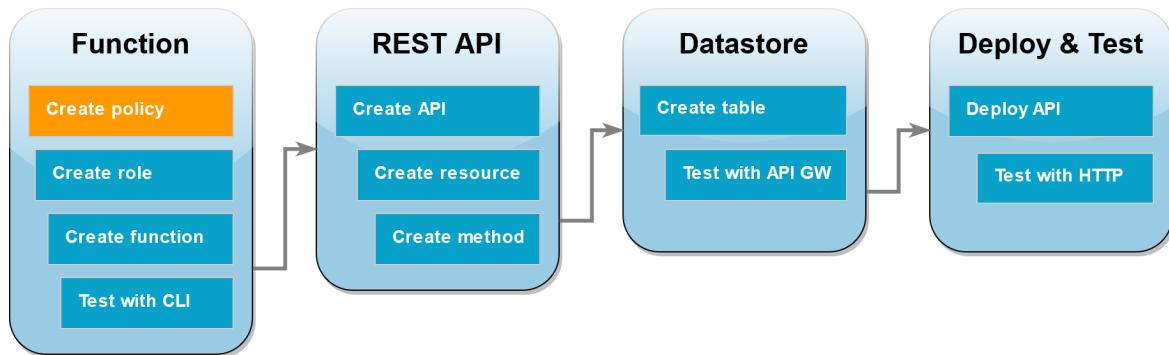
If you have not yet installed the AWS Command Line Interface, follow the steps at [Installing or updating the latest version of the AWS CLI](#) to install it.

The tutorial requires a command line terminal or shell to run commands. In Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#).

Create a permissions policy



Before you can create an [execution role \(p. 816\)](#) for your Lambda function, you first need to create a permissions policy to give your function permission to access the required AWS resources. For this tutorial, the policy allows Lambda to perform CRUD operations on a DynamoDB table and write to Amazon CloudWatch Logs.

To create the policy

- Open the [Policies page](#) of the IAM console.
- Choose **Create Policy**.
- Choose the **JSON** tab, and then paste the following custom policy into the JSON editor.

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Sid": "Stmt1428341300017",  
      "Action": [  
        "dynamodb>DeleteItem",  
        "dynamodb>GetItem",  
        "dynamodb>PutItem",  
        "dynamodb>Query",  
        "dynamodb>Scan",  
        "dynamodb>UpdateItem"  
      ],  
      "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/MyTable",  
      "Effect": "Allow"  
    }  
  ]  
}
```

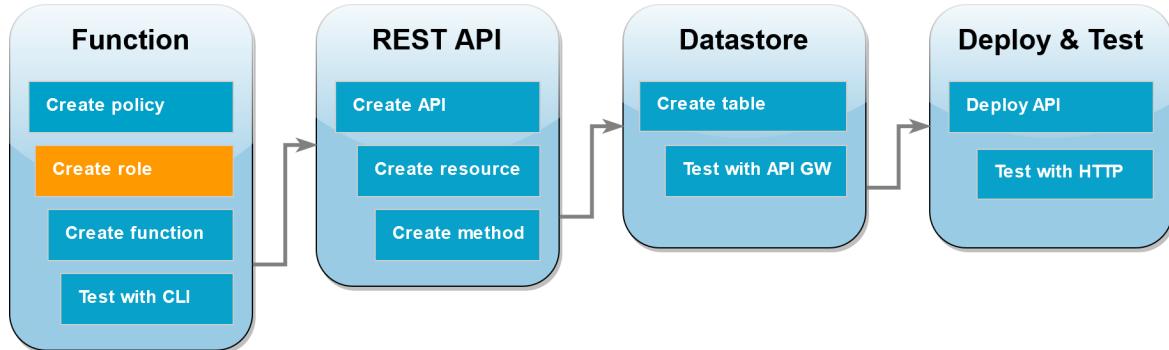
```

        "dynamodb:UpdateItem"
    ],
    "Effect": "Allow",
    "Resource": "*"
},
{
    "Sid": "",
    "Resource": "*",
    "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
    ],
    "Effect": "Allow"
}
]
}

```

4. Choose **Next: Tags**.
5. Choose **Next: Review**.
6. Under **Review policy**, for the policy **Name**, enter **lambda-apigateway-policy**.
7. Choose **Create policy**.

Create an execution role



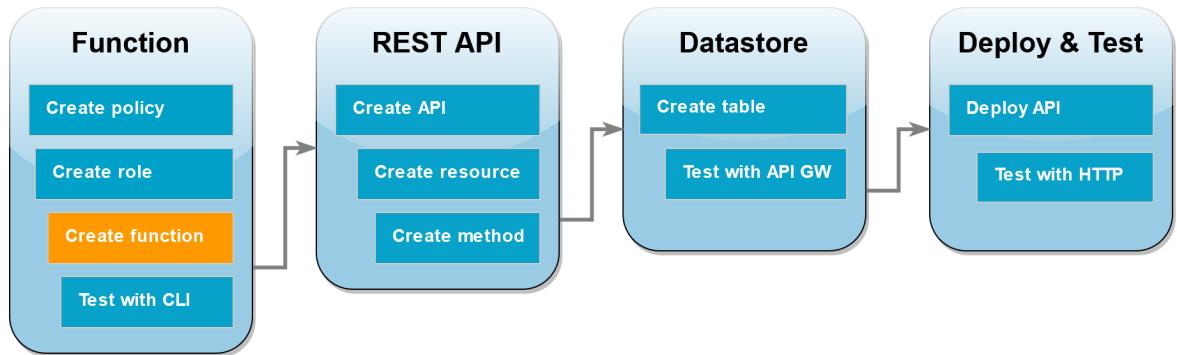
An execution role is an AWS Identity and Access Management (IAM) role that grants a Lambda function permission to access AWS services and resources. To enable your function to perform operations on a DynamoDB table, you attach the permissions policy you created in the previous step.

To create an execution role and attach your custom permissions policy

1. Open the [Roles page](#) of the IAM console.
2. Choose **Create role**.
3. For the type of trusted entity, choose **AWS service**, then for the use case, choose **Lambda**.
4. Choose **Next**.
5. In the policy search box, enter **lambda-apigateway-policy**.
6. In the search results, select the policy that you created (**lambda-apigateway-policy**), and then choose **Next**.
7. Under **Role details**, for the **Role name**, enter **lambda-apigateway-role**, then choose **Create role**.

Later in the tutorial, you need the Amazon Resource Name (ARN) of the role you just created. On the **Roles** page of the IAM console, choose the name of your role (**lambda-apigateway-role**) and copy the **Role ARN** displayed on the **Summary** page.

Create the function



The following code example receives an event input from API Gateway specifying an operation to perform on the DynamoDB table you will create and some payload data. If the parameters the function receives are valid, it performs the requested operation on the table.

Node.js

Example index.mjs

```

console.log('Loading function');

import { DynamoDBDocumentClient, PutCommand, GetCommand,
         UpdateCommand, DeleteCommand} from "@aws-sdk/lib-dynamodb";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

const ddbClient = new DynamoDBClient({ region: "us-west-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;

        switch (operation) {
            case 'create':
                await ddbDocClient.send(new PutCommand(event.payload));
                break;
            case 'read':
                var table_item = await ddbDocClient.send(new GetCommand(event.payload));
                console.log(table_item);
                break;
            case 'update':
                await ddbDocClient.send(new UpdateCommand(event.payload));
                break;
        }
    }
}

```

```
        break;
    case 'delete':
        await ddbDocClient.send(new DeleteCommand(event.payload));
        break;
    default:
        return ('Unknown operation: ${operation}');
    }
};
```

Note

In this example, the name of the DynamoDB table is defined as a variable in your function code. In a real application, best practice is to pass this parameter as an environment variable and to avoid hardcoding the table name. For more information see [Using AWS Lambda environment variables](#).

To create the function

1. Save the code example as a file named `index.mjs` and, if necessary, edit the AWS region specified in the code. The region specified in the code must be the same as the region in which you create your DynamoDB table later in the tutorial.
2. Create a deployment package using the following `zip` command.

```
zip function.zip index.mjs
```

3. Create a Lambda function using the `create-function` AWS CLI command. For the `role` parameter, enter the execution role's Amazon Resource Name (ARN) that you copied earlier.

```
aws lambda create-function --function-name LambdaFunctionOverHttps \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--role arn:aws:iam::123456789012:role/service-role/lambda-apigateway-role
```

Python 3

Example `LambdaFunctionOverHttps.py`

```
import boto3
import json

# define the DynamoDB table that Lambda will connect to
tableName = "lambda-apigateway"

# create the DynamoDB resource
dynamo = boto3.resource('dynamodb').Table(tableName)

print('Loading function')

def 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
    '''

    # define the functions used to perform the CRUD operations
    def ddb_create(x):
        dynamo.put_item(**x)
```

```
def ddb_read(x):
    dynamo.get_item(**x)

def ddb_update(x):
    dynamo.update_item(**x)

def ddb_delete(x):
    dynamo.delete_item(**x)

def echo(x):
    return x

operation = event['operation']

operations = {
    'create': ddb_create,
    'read': ddb_read,
    'update': ddb_update,
    'delete': ddb_delete,
    'echo': echo,
}

if operation in operations:
    return operations[operation](event.get('payload'))
else:
    raise ValueError('Unrecognized operation "{}".format(operation))
```

Note

In this example, the name of the DynamoDB table is defined as a variable in your function code. In a real application, best practice is to pass this parameter as an environment variable and to avoid hardcoding the table name. For more information see [Using AWS Lambda environment variables](#).

To create the function

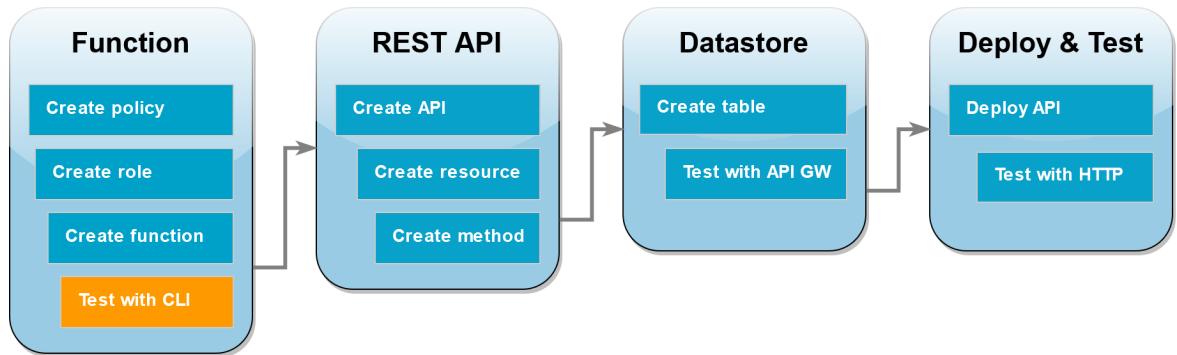
1. Save the code example as a file named `LambdaFunctionOverHttps.py`.
2. Create a deployment package using the following `zip` command.

```
zip function.zip LambdaFunctionOverHttps.py
```

3. Create a Lambda function using the `create-function` AWS CLI command. For the `role` parameter, enter the execution role's Amazon Resource Name (ARN) that you copied earlier.

```
aws lambda create-function --function-name LambdaFunctionOverHttps \
--zip-file fileb://function.zip --handler LambdaFunctionOverHttps.handler --runtime
python3.9 \
--role arn:aws:iam::123456789012:role/service-role/lambda-apigateway-role
```

Invoke the function using the AWS CLI



Before integrating your function with API Gateway, confirm that you have deployed the function successfully. Create a test event containing the parameters your API Gateway API will send to Lambda and use the AWS CLI `invoke` command to run your function.

To invoke the Lambda function with the AWS CLI

1. Save the following JSON as a file named `input.txt`.

```
{
  "operation": "echo",
  "payload": {
    "somekey1": "somevalue1",
    "somekey2": "somevalue2"
  }
}
```

2. Run the following `invoke` AWS CLI command.

```
aws lambda invoke --function-name LambdaFunctionOverHttps \
--payload file://input.txt outputfile.txt --cli-binary-format raw-in-base64-out
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

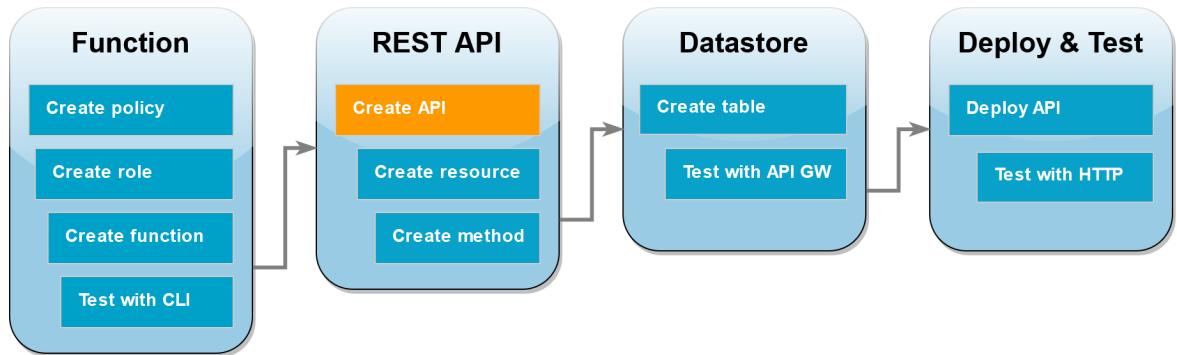
You should see the following response:

```
{
  "StatusCode": 200,
  "ExecutedVersion": "LATEST"
}
```

3. Confirm that your function performed the echo operation you specified in the JSON test event. Inspect the `outputfile.txt` file and verify it contains the following:

```
{"somekey1": "somevalue1", "somekey2": "somevalue2"}
```

Create a REST API using API Gateway

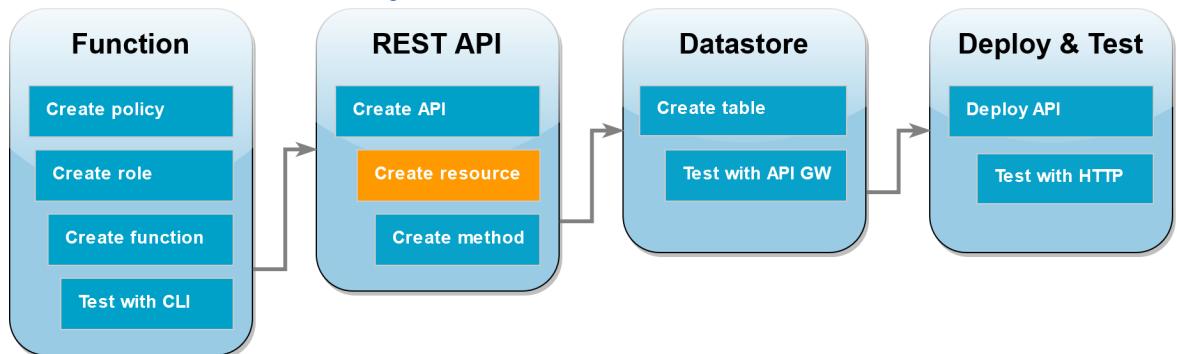


In this step, you create the API Gateway REST API you will use to invoke your Lambda function.

To create the API

1. Open the [API Gateway console](#).
2. Choose **Create API**.
3. In the **REST API** box, choose **Build**.
4. Under **Settings**, for **API Name** enter **DynamoDBOperations**.
5. Choose **Create API**.

Create a resource on your REST API

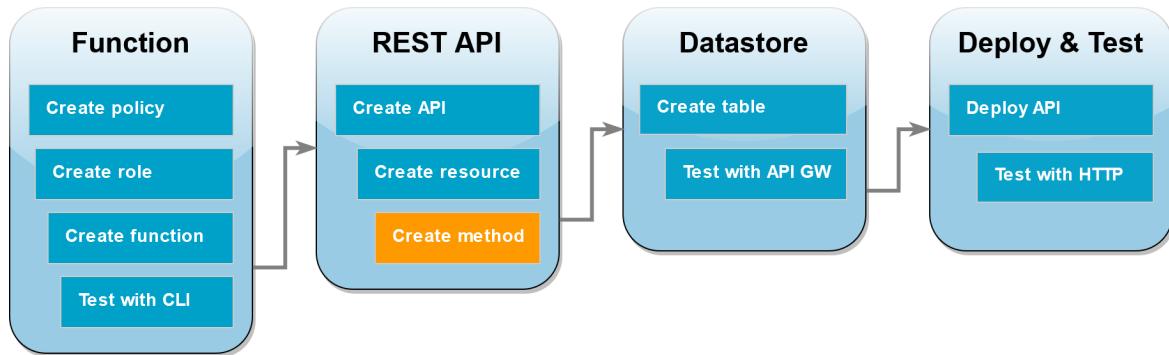


To add an HTTP method to your API, you first need to create a resource for that method to operate on. Here you create the resource to manage your DynamoDB table.

To create the resource

1. In the [API Gateway console](#), in the **Resources** tree of your API, make sure that the root (/) level is highlighted. Then, choose **Actions**, **Create Resource**.
2. Under **New child resource**, do the following:
 1. For **Resource Name**, enter **DynamoDBManager**.
 2. Keep **Resource Path** set to **/dynamodbmanager**.
 3. Choose **Create Resource**.

Create an HTTP POST method



In this step, you create a method (POST) for your `DynamoDBManager` resource. You link this POST method to your Lambda function so that when the method receives an HTTP request, API Gateway invokes your Lambda function.

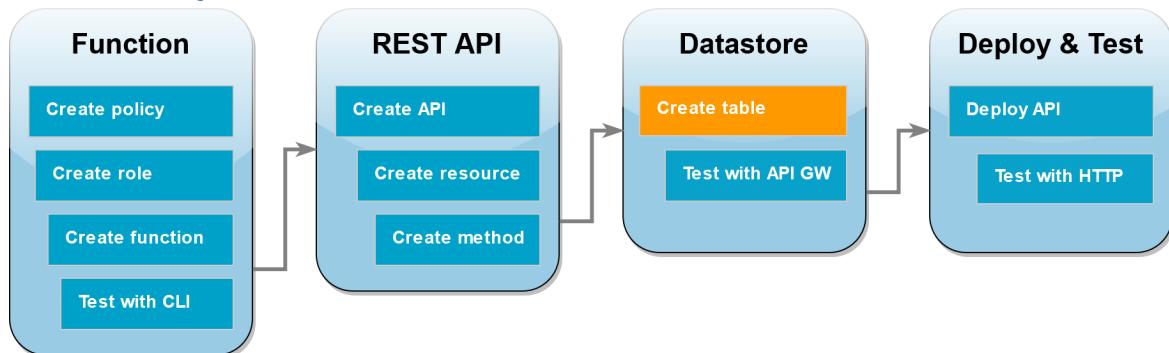
Note

For the purpose of this tutorial, one HTTP method (POST) is used to invoke a single Lambda function which carries out all of the operations on your DynamoDB table. In a real application, best practice is to use a different Lambda function and HTTP method for each operation. For more information, see [The Lambda monolith](#).

To create the POST method

1. In the [API Gateway console](#), in the **Resources** tree of your API, make sure that `/dynamodbmanager` is highlighted. Then, choose **Actions**, **Create Method**.
2. In the small dropdown menu that appears under `/dynamodbmanager`, choose **POST**, and then choose the check mark icon.
3. In the method's **Setup** pane, do the following:
 1. For **Integration type**, choose **Lambda Function**.
 2. For **Lambda Region**, choose the same AWS Region as your Lambda function.
 3. For **Lambda Function**, enter the name of your function (**LambdaFunctionOverHttps**).
 4. Select **Use Default Timeout**.
 5. Choose **Save**.
4. In the **Add Permission to Lambda Function** dialog box, choose **OK**.

Create a DynamoDB table

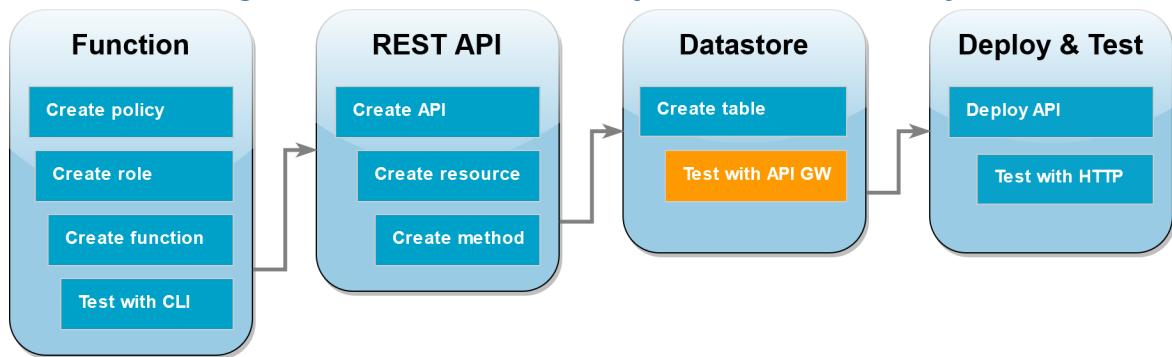


Create an empty DynamoDB table that your Lambda function will perform CRUD operations on.

To create the DynamoDB table

1. Open the [Tables page](#) of the DynamoDB console.
2. Choose **Create table**.
3. Under **Table details**, do the following:
 1. For **Table name**, enter **lambda-apigateway**.
 2. For **Partition key**, enter **id**, and keep the data type set as **String**.
4. Under **Table settings**, keep the **Default settings**.
5. Choose **Create table**.

Test the integration of API Gateway, Lambda, and DynamoDB



You're now ready to test the integration of your API Gateway API method with your Lambda function and your DynamoDB table. Using the API Gateway console, you send requests directly to your POST method using the console's test function. In this step, you first use a create operation to add a new item to your DynamoDB table, then you use an update operation to modify the item.

Test 1: To create a new item in your DynamoDB table

1. In the [API Gateway console](#), choose your API (DynamoDBOperations).
2. In the **Resources** tree, under /dynamodbmanager, choose your POST method.
3. In the **Method Execution** pane, in the **Client** box, choose **Test**.
4. In the **Method Test** pane, keep **Query Strings** and **Headers** empty. For **Request Body**, paste the following JSON:

```
{
  "operation": "create",
  "payload": {
    "Item": {
      "id": "1234ABCD",
      "number": 5
    }
  }
}
```

5. Choose **Test**.

The results that are displayed when the test completes should show status 200. This status code indicates that the create operation was successful.

To confirm, check that your DynamoDB table now contains the new item.

6. Open the [Tables page](#) of the DynamoDB console and choose the **lambda-apigateway** table.

7. Choose **Explore table items**. In the **Items returned** pane, you should see one item with the **id** 1234ABCD and the **number** 5.

Test 2: To update the item in your DynamoDB table

1. In the [API Gateway console](#), return to your POST method's **Method Test** pane.
2. In the **Method Test** pane, keep **Query Strings** and **Headers** empty. In **Request Body**, paste the following JSON:

```
{
    "operation": "update",
    "payload": {
        "Key": {
            "id": "1234ABCD"
        },
        "AttributeUpdates": {
            "number": {
                "Value": 10
            }
        }
    }
}
```

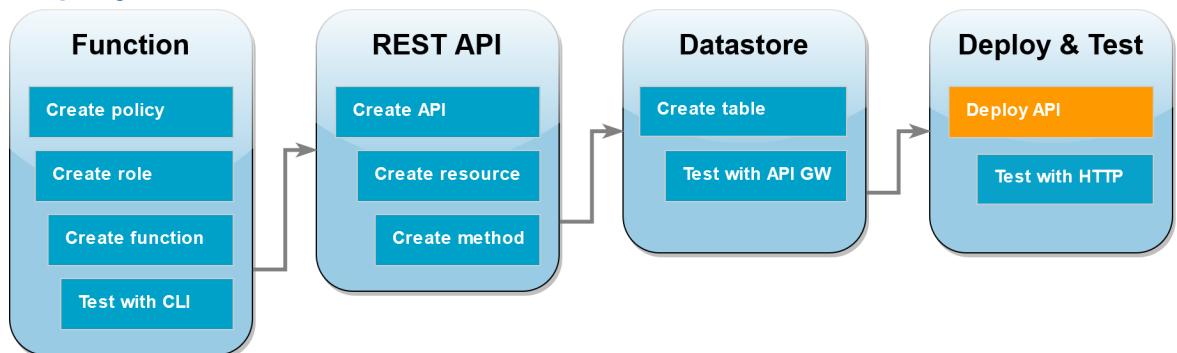
3. Choose **Test**.

The results which are displayed when the test completes should show status 200. This status code indicates that the update operation was successful.

To confirm, check that the item in your DynamoDB table has been modified.

4. Open the [Tables page](#) of the DynamoDB console and choose the lambda-apigateway table.
5. Choose **Explore table items**. In the **Items returned** pane, you should see one item with the **id** 1234ABCD and the **number** 10.

Deploy the API



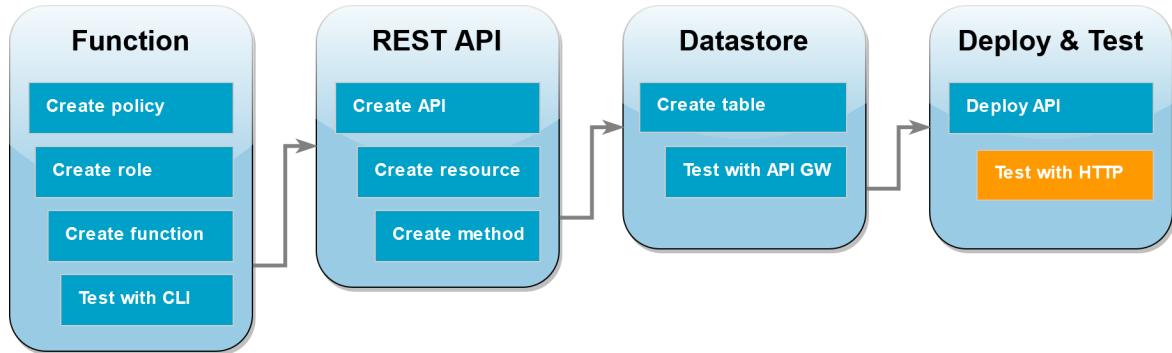
For a client to call the API, you must create a deployment and an associated stage. A stage represents a snapshot of your API including its methods and integrations.

To deploy the API

1. Open the [APIs page](#) of the [API Gateway console](#) and choose the **DynamoDBOperations API**.
2. Choose **Actions**, **Deploy API**.
3. For **Deployment stage**, choose **[New Stage]**, then for **Stage name**, enter **test**.
4. Choose **Deploy**.

- In the **test Stage Editor** pane, copy the **Invoke URL**. You will use this in the next step to invoke your function using an HTTP request.

Use curl to invoke your function using HTTP requests



You can now invoke your Lambda function by issuing an HTTP request to your API. In this step, you will create a new item in your DynamoDB table and then delete it.

To invoke the Lambda function using curl

- Run the following curl command using the invoke URL you copied in the previous step. When you use curl with the -d (data) option, it automatically uses the HTTP POST method.

```
curl https://18t0gsqxd8.execute-api.us-west-2.amazonaws.com/test/dynamodbmanager \
-d '{"operation": "create", "payload": {"Item": {"id": "5678EFGH", "number": 15}}}'
```

- To verify that the create operation was successful, do the following:

- Open the [Tables page](#) of the DynamoDB console and choose the lambda-apigateway table.
- Choose **Explore table items**. In the **Items returned** pane, you should see an item with the **id** 5678EFGH and the **number** 15.
- Run the following curl command to delete the item you just created. Use your own invoke URL.

```
curl https://18t0gsqxd8.execute-api.us-west-2.amazonaws.com/test/dynamodbmanager \
-d '{"operation": "delete", "payload": {"Key": {"id": "5678EFGH"}}}'
```

- Confirm that the delete operation was successful. In the **Items returned** pane of the DynamoDB console **Explore items** page, verify that the item with **id** 5678EFGH is no longer in the table.

Clean up your resources (optional)

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

- Open the [Functions page](#) of the Lambda console.
- Select the function that you created.
- Choose **Actions, Delete**.
- Type **delete** in the text input field and choose **Delete**.

To delete the execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the API

1. Open the [APIs page](#) of the API Gateway console.
2. Select the API you created.
3. Choose **Actions, Delete**.
4. Choose **Delete**.

To delete the DynamoDB table

1. Open the [Tables page](#) of the DynamoDB console.
2. Select the table you created.
3. Choose **Delete**.
4. Enter **delete** in the text box.
5. Choose **Delete table**.

AWS SAM template for an API Gateway application

Below is a sample AWS SAM template for the Lambda application from the [tutorial \(p. 568\)](#). Copy the text below to a file and save it next to the ZIP package you created previously. Note that the Handler and Runtime parameter values should match the ones you used when you created the function in the previous section.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  LambdaFunctionOverHttps:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs12.x
      Policies: AmazonDynamoDBFullAccess
    Events:
      HttpPost:
        Type: Api
        Properties:
          Path: '/DynamoDBOperations/DynamoDBManager'
          Method: post
```

For information on how to package and deploy your serverless application using the package and deploy commands, see [Deploying serverless applications](#) in the *AWS Serverless Application Model Developer Guide*.

Using AWS Lambda with AWS CloudTrail

AWS CloudTrail is a service that provides a record of actions taken by a user, role, or an AWS service. CloudTrail captures API calls as events. For an ongoing record of events in your AWS account, you create a trail. A trail enables CloudTrail to deliver log files of events to an Amazon S3 bucket.

You can take advantage of Amazon S3's bucket notification feature and direct Amazon S3 to publish object-created events to AWS Lambda. Whenever CloudTrail writes logs to your S3 bucket, Amazon S3 can then invoke your Lambda function by passing the Amazon S3 object-created event as a parameter. The S3 event provides information, including the bucket name and key name of the log object that CloudTrail created. Your Lambda function code can read the log object and process the access records logged by CloudTrail. For example, you might write Lambda function code to notify you if specific API call was made in your account.

In this scenario, CloudTrail writes access logs to your S3 bucket. As for AWS Lambda, Amazon S3 is the event source so Amazon S3 publishes events to AWS Lambda and invokes your Lambda function.

Example CloudTrail log

```
{  
    "Records": [  
        {  
            "eventVersion": "1.02",  
            "userIdentity": {  
                "type": "Root",  
                "principalId": "123456789012",  
                "arn": "arn:aws:iam::123456789012:root",  
                "accountId": "123456789012",  
                "accessKeyId": "access-key-id",  
                "sessionContext": {  
                    "attributes": {  
                        "mfaAuthenticated": "false",  
                        "creationDate": "2015-01-24T22:41:54Z"  
                    }  
                }  
            },  
            "eventTime": "2015-01-24T23:26:50Z",  
            "eventSource": "sns.amazonaws.com",  
            "eventName": "CreateTopic",  
            "awsRegion": "us-east-2",  
            "sourceIPAddress": "205.251.233.176",  
            "userAgent": "console.amazonaws.com",  
            "requestParameters": {  
                "name": "dropmeplease"  
            },  
            "responseElements": {  
                "topicArn": "arn:aws:sns:us-east-2:123456789012:exampletopic"  
            },  
            "requestID": "3fdb7834-9079-557e-8ef2-350abc03536b",  
            "eventID": "17b46459-dada-4278-b8e2-5a4ca9ff1a9c",  
            "eventType": "AwsApiCall",  
            "recipientAccountId": "123456789012"  
        },  
        {  
            "eventVersion": "1.02",  
            "userIdentity": {  
                "type": "Root",  
                "principalId": "123456789012",  
                "arn": "arn:aws:iam::123456789012:root",  
                "accountId": "123456789012",  
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",  
                "sessionContext": {  
                    "attributes": {  
                        "mfaAuthenticated": "false",  
                        "creationDate": "2015-01-24T22:41:54Z"  
                    }  
                }  
            },  
            "eventTime": "2015-01-24T23:26:50Z",  
            "eventSource": "sns.amazonaws.com",  
            "eventName": "DeleteTopic",  
            "awsRegion": "us-east-2",  
            "sourceIPAddress": "205.251.233.176",  
            "userAgent": "console.amazonaws.com",  
            "requestParameters": {  
                "topicArn": "arn:aws:sns:us-east-2:123456789012:exampletopic"  
            },  
            "responseElements": {  
                "topicArn": "arn:aws:sns:us-east-2:123456789012:exampletopic"  
            },  
            "requestID": "3fdb7834-9079-557e-8ef2-350abc03536b",  
            "eventID": "17b46459-dada-4278-b8e2-5a4ca9ff1a9c",  
            "eventType": "AwsApiCall",  
            "recipientAccountId": "123456789012"  
        }  
    ]  
}
```

```
        "attributes":{  
            "mfaAuthenticated":"false",  
            "creationDate":"2015-01-24T22:41:54Z"  
        }  
    },  
    "eventTime":"2015-01-24T23:27:02Z",  
    "eventSource":"sns.amazonaws.com",  
    "eventName":"GetTopicAttributes",  
    "awsRegion":"us-east-2",  
    "sourceIPAddress":"205.251.233.176",  
    "userAgent":"console.amazonaws.com",  
    "requestParameters":{  
        "topicArn":"arn:aws:sns:us-east-2:123456789012:exampletopic"  
    },  
    "responseElements":null,  
    "requestID":"4a0388f7-a0af-5df9-9587-c5c98c29cbec",  
    "eventID":"ec5bb073-8fa1-4d45-b03c-f07b9fc9ea18",  
    "eventType":"AwsApiCall",  
    "recipientAccountId":"123456789012"  
}  
]  
}
```

For detailed information about how to configure Amazon S3 as the event source, see [Using AWS Lambda with Amazon S3 \(p. 741\)](#).

Topics

- [Logging Lambda API calls with CloudTrail \(p. 585\)](#)
- [Sample function code \(p. 588\)](#)

Logging Lambda API calls with CloudTrail

Lambda is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Lambda. CloudTrail captures API calls for Lambda as events. The calls captured include calls from the Lambda console and code calls to the Lambda API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon Simple Storage Service (Amazon S3) bucket, including events for Lambda. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to Lambda, the IP address from which the request was made, who made the request, when it was made, and additional details.

For more information about CloudTrail, including how to configure and enable it, see the [AWS CloudTrail User Guide](#).

Lambda information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When supported event activity occurs in Lambda, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see [Viewing events with CloudTrail event history](#) in the *AWS CloudTrail User Guide*.

For an ongoing record of events in your AWS account, including events for Lambda, you create a *trail*. A trail enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs.

For more information, see the following topics in the *AWS CloudTrail User Guide*:

- [Overview for creating a trail](#)
- [CloudTrail supported services and integrations](#)
- [Configuring Amazon SNS notifications for CloudTrail](#)
- [Receiving CloudTrail log files from multiple regions](#) and [Receiving CloudTrail log files from multiple accounts](#)

Every log entry contains information about who generated the request. The user identity information in the log helps you determine whether the request was made with user credentials, with temporary security credentials for a role or federated user, or by another AWS service. For more information, see the [userIdentity field in the CloudTrail event reference](#).

You can store your log files in your bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted by using Amazon S3 server-side encryption (SSE).

You can choose to have CloudTrail publish Amazon Simple Notification Service (Amazon SNS) notifications when new log files are delivered if you want to take quick action upon log file delivery. For more information, see [Configuring Amazon SNS notifications for CloudTrail](#).

You can also aggregate Lambda log files from multiple Regions and multiple AWS accounts into a single S3 bucket. For more information, see [Working with CloudTrail log files](#).

List of supported Lambda API actions

Lambda supports logging the following actions as events in CloudTrail log files.

Note

In the CloudTrail log file, the eventName might include date and version information, but it is still referring to the same public API. For example the, GetFunction action might appear as

"GetFunction20150331". To see the eventName for a particular action, view a log file entry in your event history. For more information, see [Viewing events with CloudTrail event history](#) in the AWS CloudTrail User Guide.

- [AddLayerVersionPermission \(p. 1137\)](#)
- [AddPermission \(p. 1141\)](#)
- [CreateEventSourceMapping \(p. 1153\)](#)
- [CreateFunction \(p. 1165\)](#)

(The Environment and ZipFile parameters are omitted from the CloudTrail logs for CreateFunction.)

- [CreateFunctionUrlConfig](#)
- [DeleteEventSourceMapping \(p. 1186\)](#)
- [DeleteFunction \(p. 1193\)](#)
- [DeleteFunctionUrlConfig](#)
- [GetEventSourceMapping \(p. 1214\)](#)
- [GetFunction \(p. 1220\)](#)
- [GetFunctionUrlConfig](#)
- [GetFunctionConfiguration \(p. 1229\)](#)
- [GetLayerVersionPolicy \(p. 1250\)](#)
- [GetPolicy \(p. 1252\)](#)
- [ListEventSourceMappings \(p. 1279\)](#)
- [ListFunctions \(p. 1286\)](#)
- [ListFunctionUrlConfigs](#)
- [PublishLayerVersion \(p. 1310\)](#)

(The ZipFile parameter is omitted from the CloudTrail logs for PublishLayerVersion.)

- [RemovePermission \(p. 1343\)](#)
- [UpdateEventSourceMapping \(p. 1356\)](#)
- [UpdateFunctionCode \(p. 1367\)](#)

(The ZipFile parameter is omitted from the CloudTrail logs for UpdateFunctionCode.)

- [UpdateFunctionConfiguration \(p. 1377\)](#)

(The Environment parameter is omitted from the CloudTrail logs for UpdateFunctionConfiguration.)

- [UpdateFunctionUrlConfig](#)

Understanding Lambda log file entries

CloudTrail log files contain one or more log entries where each entry is made up of multiple JSON-formatted events. A log entry represents a single request from any source and includes information about the requested action, any parameters, the date and time of the action, and so on. The log entries are not guaranteed to be in any particular order. That is, they are not an ordered stack trace of the public API calls.

The following example shows CloudTrail log entries for the GetFunction and DeleteFunction actions.

Note

The eventName might include date and version information, such as "GetFunction20150331", but it is still referring to the same public API.

```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::999999999999:user/myUserName",
        "accountId": "999999999999",
        "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::999999999999:user/myUserName is not authorized to perform: lambda:GetFunction on resource: arn:aws:lambda:us-west-2:999999999999:function:other-acct-function",
      "requestParameters": null,
      "responseElements": null,
      "requestID": "7aebcd0f-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "e92a3e85-8ecd-4d23-8074-843aabfe89bf",
      "eventType": "AwsApiCall",
      "recipientAccountId": "999999999999"
    },
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::999999999999:user/myUserName",
        "accountId": "999999999999",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:04:42Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "DeleteFunction",
      "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": "999999999999"
    }
  ]
}
```

Using CloudTrail to track function invocations

CloudTrail also logs data events. You can turn on data event logging so that you log an event every time Lambda functions are invoked. This helps you understand what identities are invoking the functions and the frequency of their invocations. For more information on this option, see [Logging data events for trails](#).

Note

CloudTrail logs only authenticated and authorized requests. CloudTrail does not log requests that fail authentication (credentials are missing or the provided credentials are not valid) or requests with credentials that are not authorized to invoke the function.

Using CloudTrail to troubleshoot disabled event sources

One data event that can be encountered is a LambdaESMDisabled event. There are five general categories of error that are associated with this event:

RESOURCE_NOT_FOUND

The resource specified in the request does not exist.

FUNCTION_NOT_FOUND

The function attached to the event source does not exist.

REGION_NAME_NOT_VALID

A Region name provided to the event source or function is invalid.

AUTHORIZATION_ERROR

Permissions have not been set, or are misconfigured.

FUNCTION_IN_FAILED_STATE

The function code does not compile, has encountered an unrecoverable exception, or a bad deployment has occurred.

These errors are included in the CloudTrail event message within the `serviceEventDetails` entity.

Example `serviceEventDetails` entity

```
"serviceEventDetails":{  
    "ESMDisableReason":"Lambda Function not found"  
}
```

Sample function code

Sample code is available for the following languages.

Topics

- [Node.js \(p. 588\)](#)

Node.js

The following example processes CloudTrail logs, and sends a notification when an Amazon SNS topic was created.

Example `index.js`

```
var aws = require('aws-sdk');  
var zlib = require('zlib');  
var async = require('async');  
  
var EVENT_SOURCE_TO_TRACK = '/sns.amazonaws.com/';  
var EVENT_NAME_TO_TRACK = '/CreateTopic/';
```

```

var DEFAULT_SNS_REGION  = 'us-west-2';
var SNS_TOPIC_ARN        = 'The ARN of your SNS topic';

var s3 = new aws.S3();
var sns = new aws.SNS({
    apiVersion: '2010-03-31',
    region: DEFAULT_SNS_REGION
});

exports.handler = function(event, context, callback) {
    var srcBucket = event.Records[0].s3.bucket.name;
    var srcKey = event.Records[0].s3.object.key;

    async.waterfall([
        function fetchLogFromS3(next){
            console.log('Fetching compressed log from S3...');
            s3.getObject({
                Bucket: srcBucket,
                Key: srcKey
            },
            next);
        },
        function uncompressLog(response, next){
            console.log("Uncompressing log...");
            zlib.gunzip(response.Body, next);
        },
        function publishNotifications(jsonBuffer, next) {
            console.log('Filtering log...');
            var json = jsonBuffer.toString();
            console.log('CloudTrail JSON from S3:', json);
            var records;
            try {
                records = JSON.parse(json);
            } catch (err) {
                next('Unable to parse CloudTrail JSON: ' + err);
                return;
            }
            var matchingRecords = records
                .Records
                .filter(function(record) {
                    return record.eventSource.match(EVENT_SOURCE_TO_TRACK)
                        && record.eventName.match(EVENT_NAME_TO_TRACK);
                });

            console.log('Publishing ' + matchingRecords.length + ' notification(s) in
parallel...');

            async.each(
                matchingRecords,
                function(record, publishComplete) {
                    console.log('Publishing notification: ', record);
                    sns.publish({
                        Message:
                            'Alert... SNS topic created: \n TopicARN=' +
record.responseElements.topicArn + '\n\n' +
                            JSON.stringify(record),
                        TopicArn: SNS_TOPIC_ARN
                    }, publishComplete);
                },
                next
            );
        },
        function (err) {
            if (err) {
                console.error('Failed to publish notifications: ', err);
            } else {
                console.log('Successfully published all notifications.');
            }
        }
    ], function (err) {
        if (err) {
            console.error('Failed to publish notifications: ', err);
        } else {
            console.log('Successfully published all notifications.');
        }
    });
}

```

```
        }
        callback(null,"message");
    );
};
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Node.js Lambda functions with .zip file archives \(p. 263\)](#).

Using AWS Lambda with Amazon EventBridge (CloudWatch Events)

Note

Amazon EventBridge is the preferred way to manage your events. CloudWatch Events and EventBridge are the same underlying service and API, but EventBridge provides more features. Changes you make in either CloudWatch Events or EventBridge will appear in each console. For more information, see the [Amazon EventBridge documentation](#).

EventBridge (CloudWatch Events) helps you to respond to state changes in your AWS resources. For more information about EventBridge, see [What is Amazon EventBridge?](#) in the *Amazon EventBridge User Guide*.

When your resources change state, they automatically send events into an event stream. With EventBridge (CloudWatch Events), you can create rules that match selected events in the stream and route them to your AWS Lambda function to take action. For example, you can automatically invoke an AWS Lambda function to log the state of an [EC2 instance](#) or [AutoScaling group](#).

EventBridge (CloudWatch Events) invokes your function asynchronously with an event document that wraps the event from its source. The following example shows an event that originated from a database snapshot in Amazon Relational Database Service.

Example EventBridge (CloudWatch Events) event

```
{  
    "version": "0",  
    "id": "fe8d3c65-xmpl-c5c3-2c87-81584709a377",  
    "detail-type": "RDS DB Instance Event",  
    "source": "aws.rds",  
    "account": "123456789012",  
    "time": "2020-04-28T07:20:20Z",  
    "region": "us-east-2",  
    "resources": [  
        "arn:aws:rds:us-east-2:123456789012:db:rdz6xmpliljlb1"  
    ],  
    "detail": {  
        "EventCategories": [  
            "backup"  
        ],  
        "SourceType": "DB_INSTANCE",  
        "SourceArn": "arn:aws:rds:us-east-2:123456789012:db:rdz6xmpliljlb1",  
        "Date": "2020-04-28T07:20:20.112Z",  
        "Message": "Finished DB Instance backup",  
        "SourceIdentifier": "rdz6xmpliljlb1"  
    }  
}
```

You can also create a Lambda function and direct AWS Lambda to invoke it on a regular schedule. You can specify a fixed rate (for example, invoke a Lambda function every hour or 15 minutes), or you can specify a Cron expression.

Example EventBridge (CloudWatch Events) message event

```
{  
  "version": "0",  
  "account": "123456789012",  
  "region": "us-east-2",  
  "detail": {},  
  "detail-type": "Scheduled Event".
```

```
"source": "aws.events",
"time": "2019-03-01T01:23:45Z",
"id": "cdc73f9d-aea9-11e3-9d5a-835b769c0d9c",
"resources": [
    "arn:aws:events:us-east-2:123456789012:rule/my-schedule"
]
}
```

To configure EventBridge (CloudWatch Events) to invoke your function

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function
3. Under **Function overview**, choose **Add trigger**.
4. Set the trigger type to **EventBridge (CloudWatch Events)**.
5. For **Rule**, choose **Create a new rule**.
6. Configure the remaining options and choose **Add**.

For more information on expressions schedules, see [Schedule expressions using rate or cron \(p. 595\)](#).

Each AWS account can have up to 100 unique event sources of the **EventBridge (CloudWatch Events)-Schedule** source type. Each of these can be the event source for up to five Lambda functions. That is, you can have up to 500 Lambda functions that can be executing on a schedule in your AWS account.

Topics

- [Tutorial: Using AWS Lambda with scheduled events \(p. 592\)](#)
- [Schedule expressions using rate or cron \(p. 595\)](#)

Tutorial: Using AWS Lambda with scheduled events

In this tutorial, you do the following:

- Create a Lambda function using the **Schedule a periodic check of any URL** blueprint. You configure the Lambda function to run every minute. Note that if the function returns an error, Lambda logs error metrics to Amazon CloudWatch.
- Configure a CloudWatch alarm on the `Errors` metric of your Lambda function to post a message to your Amazon SNS topic when AWS Lambda emits error metrics to CloudWatch. You subscribe to the Amazon SNS topics to get email notification. In this tutorial, you do the following to set this up:
 - Create an Amazon SNS topic.
 - Subscribe to the topic so you can get email notifications when a new message is posted to the topic.
 - In Amazon CloudWatch, set an alarm on the `Errors` metric of your Lambda function to publish a message to your SNS topic when errors occur.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console \(p. 4\)](#) to create your first Lambda function.

Create a Lambda function

1. Sign in to the AWS Management Console and open the AWS Lambda console at <https://console.aws.amazon.com/lambda/>.

2. Choose **Create function**.
3. Choose **Use a blueprint**.
4. Open the **Select blueprint** dropdown list and choose the **Schedule a periodic check of any URL** blueprint.
5. Configure the following settings.
 - **Function name – lambda-canary**.
 - **Execution role – Create a new role from AWS policy templates**.
 - **Role name – lambda-eventbridge-role**.
 - **Policy templates – Simple microservice permissions**.
 - **Rule – Create a new rule**.
 - **Rule name – CheckWebsiteScheduledEvent**.
 - **Rule description – CheckWebsiteScheduledEvent trigger**.
 - **Rule type – Schedule expression**.
 - **Schedule expression – rate(1 minute)**.
 - **Environment variables**
 - **site – https://docs.aws.amazon.com/lambda/latest/dg/welcome.html**
 - **expected – What is AWS Lambda?**
6. Choose **Create function**.

EventBridge (CloudWatch Events) emits an event every minute, based on the schedule expression. The event triggers the Lambda function, which verifies that the expected string appears in the specified page. For more information on expressions schedules, see [Schedule expressions using rate or cron \(p. 595\)](#).

Test the Lambda function

Test the function with a sample event provided by the Lambda console.

1. Open the [Functions page](#) of the Lambda console.
2. Choose the **lambda-canary** function.
3. Choose **Test**.
4. Create a new event using the **CloudWatch** event template (**cloudwatch-scheduled-event**).
5. Choose **Create event**.
6. Choose **Invoke**.

The output from the function execution is shown at the top of the page.

Create an Amazon SNS topic and subscribe to it

Create an Amazon Simple Notification Service (Amazon SNS) topic to receive notifications when the canary function returns an error.

To create a topic

1. Open the [Amazon SNS console](#).
2. Switch to the AWS Region where you created the Lambda function.
3. Choose **Topics**, and then choose **Create topic**.
4. Create a topic with the following settings.
 - **Type – Standard**.

- Name – **lambda-canary-notifications**.
 - Display name – **Canary**.
5. Choose **Create topic**.
 6. On the **lambda-canary-notifications** topic page, choose **Create subscription**.
 7. Create a subscription with the following settings.
 - **Protocol** – **Email**.
 - **Endpoint** – Your email address.
 8. Choose **Create subscription**.

Amazon SNS sends an email from **Canary <no-reply@sns.amazonaws.com>**, reflecting the friendly name of the topic. Use the link in the email to confirm your address.

Configure an alarm

Configure an alarm in Amazon CloudWatch that monitors the Lambda function and sends a notification when it fails.

To create an alarm

1. Open the [CloudWatch console](#).
2. Switch to the AWS Region where you created the Lambda function.
3. Choose **All alarms**.
4. Choose **Create alarm**.
5. On the **Specify metric and conditions** page, choose **Select metric**.
6. In the **Metrics** search box, enter **lambda-canary Errors**.
7. Choose **Lambda > By Function 1 Name**.
8. Select the **lambda-canary Errors** metric.
9. On the **Specify metric and conditions** page, in the **Statistic** drop-down menu, choose **Sum**.
10. Set the threshold to **Greater/Equal than 1**.
11. On the **Configure actions** page, add a notification with the following settings:
 - Alarm state trigger – **In alarm**
 - Send notification to... – **lambda-canary-notifications**
12. On the **Add name and description** page, enter the following:
 - **Name** – **lambda-canary-alarm**
 - **Description** – **Lambda canary alarm**
13. Choose **Create alarm**.

Test the alarm

Update the function configuration to cause the function to return an error, which triggers the alarm.

To trigger an alarm

1. Open the [Functions page](#) of the Lambda console.
2. Choose the **lambda-canary** function.
3. Scroll down. Under **Environment variables**, choose **Edit**.

4. Set **expected** to **404**.
5. Choose **Save**.

Wait a minute, and then check your email for a message from Amazon SNS.

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

To delete the CloudWatch alarm

1. Open the [All alarms page](#) of the CloudWatch console.
2. Select the alarm you created.
3. Choose **Actions, Delete**.
4. Choose **Delete**.

To delete the Amazon SNS subscription

1. Open the [Subscriptions page](#) of the Amazon SNS console.
2. Select the subscription you created.
3. Choose **Delete, Delete**.

To delete the Amazon SNS topic

1. Open the [Topics page](#) of the Amazon SNS console.
2. Select the topic you created.
3. Choose **Delete**.
4. Enter **delete me** in the text input field.
5. Choose **Delete**.

Schedule expressions using rate or cron

AWS Lambda supports standard rate and cron expressions for frequencies of up to once per minute. Rate expressions are simpler to define but do not offer the fine-grained schedule control that cron triggers support.

EventBridge (CloudWatch Events) rate expressions have the following format.

```
rate(Value Unit)
```

Where *Value* is a positive integer and *Unit* can be minute(s), hour(s), or day(s). A rate expression starts when you create the scheduled event rule. For a singular value the unit must be singular (for example, `rate(1 day)`), otherwise plural (for example, `rate(5 days)`).

Rate expression examples

Frequency	Expression
Every 5 minutes	<code>rate(5 minutes)</code>
Every hour	<code>rate(1 hour)</code>
Every seven days	<code>rate(7 days)</code>

Cron expressions have the following format.

```
cron(Minutes Hours Day-of-month Month Day-of-week Year)
```

Cron expression examples

Frequency	Expression
10:15 AM (UTC) every day	<code>cron(15 10 * * ? *)</code>
6:00 PM Monday through Friday	<code>cron(0 18 ? * MON-FRI *)</code>
8:00 AM on the first day of the month	<code>cron(0 8 1 * ? *)</code>
Every 10 min on weekdays	<code>cron(0/10 * ? * MON-FRI *)</code>
Every 5 minutes between 8:00 AM and 5:55 PM weekdays	<code>cron(0/5 8-17 ? * MON-FRI *)</code>
9:00 AM on the first Monday of each month	<code>cron(0 9 ? * 2#1 *)</code>

Note the following:

- If you are using the Lambda console, do not include the `cron` prefix in your expression.
- One of the day-of-month or day-of-week values must be a question mark (?).

For more information, see [Schedule expressions for rules](#) in the *EventBridge User Guide*.

Using Lambda with CloudWatch Logs

You can use a Lambda function to monitor and analyze logs from an Amazon CloudWatch Logs log stream. Create [subscriptions](#) for one or more log streams to invoke a function when logs are created or match an optional pattern. Use the function to send a notification or persist the log to a database or storage.

CloudWatch Logs invokes your function asynchronously with an event that contains log data. The value of the data field is a Base64-encoded .gzip file archive.

Example CloudWatch Logs message event

```
{  
  "awslogs": {  
    "data":  
      "ewogICAgIm1lc3NhZ2VUeXB1IjogIkRBVEFTUVTU0FHSRIsCiAgICAib3duZXII0iAiMTIzNDU2Nzg5MDEyIiwKICAgICJsb2dH  
  }  
}
```

When decoded and decompressed, the log data is a JSON document with the following structure:

Example CloudWatch Logs message data (decoded)

```
{  
  "messageType": "DATA_MESSAGE",  
  "owner": "123456789012",  
  "logGroup": "/aws/lambda/echo-nodejs",  
  "logStream": "2019/03/13/[$LATEST]94fa867e5374431291a7fc14e2f56ae7",  
  "subscriptionFilters": [  
    "LambdaStream_cloudwatchlogs-node"  
  ],  
  "logEvents": [  
    {  
      "id": "34622316099697884706540976068822859012661220141643892546",  
      "timestamp": 1552518348220,  
      "message": "REPORT RequestId: 6234bffe-149a-b642-81ff-2e8e376d8aff\\tDuration:  
46.84 ms\\tBilled Duration: 47 ms \\tMemory Size: 192 MB\\tMax Memory Used: 72 MB\\t\\n"  
    }  
  ]  
}
```

For a sample application that uses CloudWatch Logs as a trigger, see [Error processor sample application for AWS Lambda \(p. 1015\)](#).

Using AWS Lambda with AWS CloudFormation

In an AWS CloudFormation template, you can specify a Lambda function as the target of a custom resource. Use custom resources to process parameters, retrieve configuration values, or call other AWS services during stack lifecycle events.

The following example invokes a function that's defined elsewhere in the template.

Example – Custom resource definition

```
Resources:
  primerinvoke:
    Type: AWS::CloudFormation::CustomResource
    Version: "1.0"
    Properties:
      ServiceToken: !GetAtt primer.Arn
      FunctionName: !Ref randomerror
```

The service token is the Amazon Resource Name (ARN) of the function that AWS CloudFormation invokes when you create, update, or delete the stack. You can also include additional properties like `FunctionName`, which AWS CloudFormation passes to your function as is.

AWS CloudFormation invokes your Lambda function [asynchronously \(p. 123\)](#) with an event that includes a callback URL.

Example – AWS CloudFormation message event

```
{
  "RequestType": "Create",
  "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-processor-primer-14R0R2T3JKU66",
  "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=AKIAIOSF0DNN7EXAMPLE&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-14R0R2T3JKU66",
    "FunctionName": "lambda-error-processor-randomerror-ZWUC391MQAJK"
  }
}
```

The function is responsible for returning a response to the callback URL that indicates success or failure. For the full response syntax, see [Custom resource response objects](#).

Example – AWS CloudFormation custom resource response

```
{
  "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"
```

}

AWS CloudFormation provides a library called `cfn-response` that handles sending the response. If you define your function within a template, you can require the library by name. AWS CloudFormation then adds the library to the deployment package that it creates for the function.

If your function that a Custom Resource uses has an [Elastic Network Interface \(p. 90\)](#) attached to it, add the following resources to the VPC policy where `region` is the Region the function is in without the dashes. For example, `us-east-1` is `useast1`. This will allow the Custom Resource to respond to the callback URL that sends a signal back to the AWS CloudFormation stack.

```
arn:aws:s3:::cloudformation-custom-resource-response-region",  
"arn:aws:s3:::cloudformation-custom-resource-response-region/*",
```

The following example function invokes a second function. If the call succeeds, the function sends a success response to AWS CloudFormation, and the stack update continues. The template uses the [AWS::Serverless::Function](#) resource type provided by AWS Serverless Application Model.

Example [error-processor/template.yml](#) – Custom resource function

```
Transform: 'AWS::Serverless-2016-10-31'  
Resources:  
  primer:  
    Type: AWS::Serverless::Function  
    Properties:  
      Handler: index.handler  
      Runtime: nodejs12.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
```

If the function that the custom resource invokes isn't defined in a template, you can get the source code for `cfn-response` from [cfn-response module](#) in the AWS CloudFormation User Guide.

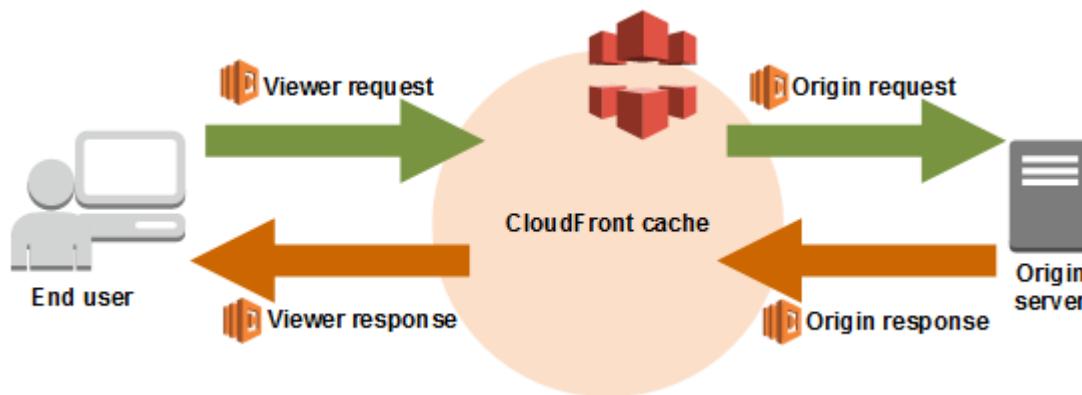
For a sample application that uses a custom resource to ensure that a function's log group is created before another resource that depends on it, see [Error processor sample application for AWS Lambda \(p. 1015\)](#).

For more information about custom resources, see [Custom resources](#) in the *AWS CloudFormation User Guide*.

Using AWS Lambda with CloudFront Lambda@Edge

Lambda@Edge lets you run Node.js and Python Lambda functions to customize content that CloudFront delivers, executing the functions in AWS locations closer to the viewer. The functions run in response to CloudFront events, without provisioning or managing servers. You can use Lambda functions to change CloudFront requests and responses at the following points:

- After CloudFront receives a request from a viewer (viewer request)
- Before CloudFront forwards the request to the origin (origin request)
- After CloudFront receives the response from the origin (origin response)
- Before CloudFront forwards the response to the viewer (viewer response)



Note

Lambda@Edge supports a limited set of runtimes and features. For details, see [Requirements and restrictions on Lambda functions](#) in the Amazon CloudFront developer guide.

You can also generate responses to viewers without ever sending the request to the origin.

Example CloudFront message event

```
{  
  "Records": [  
    {  
      "cf": {  
        "config": {  
          "distributionId": "EDFDVBD6EXAMPLE"  
        },  
        "request": {  
          "clientIp": "2001:0db8:85a3:0:0:8a2e:0370:7334",  
          "method": "GET",  
          "uri": "/picture.jpg",  
          "headers": {  
            "host": [  
              {  
                "key": "Host",  
                "value": "d111111abcdef8.cloudfront.net"  
              }  
            ],  
            "user-agent": [  
              {  
                "key": "User-Agent",  
                "value": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/89.0.4369.90 Safari/537.36"  
              }  
            ]  
          }  
        }  
      }  
    }  
  ]  
}
```

```
        "value": "curl/7.51.0"
    }
}
}
]
}
```

With Lambda@Edge, you can build a variety of solutions, for example:

- Inspect cookies to rewrite URLs to different versions of a site for A/B testing.
- Send different objects to your users based on the User-Agent header, which contains information about the device that submitted the request. For example, you can send images in different resolutions to users based on their devices.
- Inspect headers or authorized tokens, inserting a corresponding header and allowing access control before forwarding a request to the origin.
- Add, delete, and modify headers, and rewrite the URL path to direct users to different objects in the cache.
- Generate new HTTP responses to do things like redirect unauthenticated users to login pages, or create and deliver static webpages right from the edge. For more information, see [Using Lambda functions to generate HTTP responses to viewer and origin requests](#) in the *Amazon CloudFront Developer Guide*.

For more information about using Lambda@Edge, see [Using CloudFront with Lambda@Edge](#).

Using AWS Lambda with AWS CodeCommit

You can create a trigger for an AWS CodeCommit repository so that events in the repository will invoke a Lambda function. For example, you can invoke a Lambda function when a branch or tag is created or when a push is made to an existing branch.

Example AWS CodeCommit message event

```
{  
    "Records": [  
        {  
            "awsRegion": "us-east-2",  
            "codecommit": {  
                "references": [  
                    {  
                        "commit": "5e493c6f3067653f3d04eca608b4901eb227078",  
                        "ref": "refs/heads/master"  
                    }  
                ]  
            },  
            "eventId": "31ade2c7-f889-47c5-a937-1cf99e2790e9",  
            "eventName": "ReferenceChanges",  
            "eventPartNumber": 1,  
            "eventSource": "aws:codecommit",  
            "eventSourceARN": "arn:aws:codecommit:us-east-2:123456789012:lambda-pipeline-  
repo",  
            "eventTime": "2019-03-12T20:58:25.400+0000",  
            "eventTotalParts": 1,  
            "eventTriggerConfigId": "0d17d6a4-efeb-46f3-b3ab-a63741badeb8",  
            "eventTriggerName": "index.handler",  
            "eventVersion": "1.0",  
            "userIdentityARN": "arn:aws:iam::123456789012:user/intern"  
        }  
    ]  
}
```

For more information, see [Manage triggers for an AWS CodeCommit repository](#).

Using AWS Lambda with AWS CodePipeline

AWS CodePipeline is a service that enables you to create continuous delivery pipelines for applications that run on AWS. You can create a pipeline to deploy your Lambda application. You can also configure a pipeline to invoke a Lambda function to perform a task when the pipeline runs. When you [create a Lambda application \(p. 960\)](#) in the Lambda console, Lambda creates a pipeline that includes source, build, and deploy stages.

CodePipeline invokes your function asynchronously with an event that contains details about the job. The following example shows an event from a pipeline that invoked a function named my-function.

Example CodePipeline event

```
{
    "CodePipeline.job": {
        "id": "c0d76431-b0e7-xmpl-97e3-e8ee786eb6f6",
        "accountId": "123456789012",
        "data": {
            "actionConfiguration": {
                "configuration": {
                    "FunctionName": "my-function",
                    "UserParameters": "{\"KEY\": \"VALUE\"}"
                }
            },
            "inputArtifacts": [
                {
                    "name": "my-pipeline-SourceArtifact",
                    "revision": "e0c7xmpl2308ca3071aa7bab414de234ab52eea",
                    "location": {
                        "type": "S3",
                        "s3Location": {
                            "bucketName": "us-west-2-123456789012-my-pipeline",
                            "objectKey": "my-pipeline/test-api-2/Td0SFRV"
                        }
                    }
                }
            ],
            "outputArtifacts": [
                {
                    "name": "invokeOutput",
                    "revision": null,
                    "location": {
                        "type": "S3",
                        "s3Location": {
                            "bucketName": "us-west-2-123456789012-my-pipeline",
                            "objectKey": "my-pipeline/invokeOutp/D0YHsJn"
                        }
                    }
                }
            ],
            "artifactCredentials": {
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "secretAccessKey": "6CGtmAa3lzWtV7a...",
                "sessionToken": "IQoJb3JpZ2luX2VjEA...",
                "expirationTime": 1575493418000
            }
        }
    }
}
```

To complete the job, the function must call the CodePipeline API to signal success or failure. The following example Node.js function uses the PutJobSuccessResult operation to signal success. It gets the job ID for the API call from the event object.

Example index.js

```
var AWS = require('aws-sdk')
var codepipeline = new AWS.CodePipeline()

exports.handler = async (event) => {
    console.log(JSON.stringify(event, null, 2))
    var jobId = event["CodePipeline.job"].id
    var params = {
        jobId: jobId
    }
    return codepipeline.putJobSuccessResult(params).promise()
}
```

For asynchronous invocation, Lambda queues the message and [retries \(p. 161\)](#) if your function returns an error. Configure your function with a [destination \(p. 125\)](#) to retain events that your function could not process.

For a tutorial on how to configure a pipeline to invoke a Lambda function, see [Invoke an AWS Lambda function in a pipeline](#) in the *AWS CodePipeline User Guide*.

You can use AWS CodePipeline to create a continuous delivery pipeline for your Lambda application. CodePipeline combines source control, build, and deployment resources to create a pipeline that runs whenever you make a change to your application's source code.

For an alternate method of creating a pipeline with AWS Serverless Application Model and AWS CloudFormation, watch [Automate your serverless application deployments](#) on the Amazon Web Services YouTube channel.

Permissions

To invoke a function, a CodePipeline pipeline needs permission to use the following API operations:

- [ListFunctions \(p. 1286\)](#)
- [InvokeFunction \(p. 1260\)](#)

The default [pipeline service role](#) includes these permissions.

To complete a job, the function needs the following permissions in its [execution role \(p. 816\)](#).

- codepipeline:PutJobSuccessResult
- codepipeline:PutJobFailureResult

These permissions are included in the [AWSCodePipelineCustomActionAccess](#) managed policy.

Working with Amazon CodeWhisperer in the Lambda console

Amazon CodeWhisperer is a general purpose, machine learning-powered code generator that provides you with code recommendations in real time. When activated in the Lambda console, CodeWhisperer automatically generates suggestions based on your existing code and comments. Your personalized recommendations can vary in size and scope, ranging from a single one-liner to fully formed functions.

For more information, see the [Amazon CodeWhisperer User Guide](#).

Using AWS Lambda with Amazon Cognito

The Amazon Cognito Events feature enables you to run Lambda functions in response to events in Amazon Cognito. Amazon Cognito provides authentication, authorization, and user management for your web and mobile apps. You can invoke a Lambda function in response to important events in Amazon Cognito. For example, using the Sync Trigger events, you can invoke a Lambda function that is published each time a dataset is synchronized. To learn more and walk through an example, see [Introducing Amazon Cognito Events: Sync Triggers](#) in the Mobile Development blog.

Example Amazon Cognito message event

```
{  
    "datasetName": "datasetName",  
    "eventType": "SyncTrigger",  
    "region": "us-east-1",  
    "identityId": "identityId",  
    "datasetRecords": [  
        {"SampleKey2": {  
            "newValue": "newValue2",  
            "oldValue": "oldValue2",  
            "op": "replace"  
        }  
        {"SampleKey1": {  
            "newValue": "newValue1",  
            "oldValue": "oldValue1",  
            "op": "replace"  
        }  
    ],  
    "identityPoolId": "identityPoolId",  
    "version": 2  
}
```

You configure event source mapping using Amazon Cognito event subscription configuration. For information about event source mapping and a sample event, see [Amazon Cognito events](#) in the *Amazon Cognito Developer Guide*.

Using AWS Lambda with AWS Config

You can use AWS Lambda functions to evaluate whether your AWS resource configurations comply with your custom Config rules. As resources are created, deleted, or changed, AWS Config records these changes and sends the information to your Lambda functions. Your Lambda functions then evaluate the changes and report results to AWS Config. You can then use AWS Config to assess overall resource compliance: you can learn which resources are noncompliant and which configuration attributes are the cause of noncompliance.

Example AWS Config message event

```
{  
    "invokingEvent": "{\"configurationItem\":{\"configurationItemCaptureTime\": \"2016-02-17T01:36:34.043Z\", \"awsAccountId\": \"000000000000\", \"configurationItemStatus\": \"OK\", \"resourceId\": \"i-0000000000\", \"ARN\": \"arn:aws:ec2:us-east-1:000000000000:instance/i-0000000000\", \"awsRegion\": \"us-east-1\", \"availabilityZone\": \"us-east-1a\", \"resourceType\": \"AWS::EC2::Instance\", \"tags\": {\"Foo\": \"Bar\"}, \"relationships\": [{\"resourceId\": \"eipalloc-00000000\", \"resourceType\": \"AWS::EC2::EIP\", \"name\": \"Is attached to ElasticIp\"}], \"configuration\": {\"foo\": \"bar\"}}, \"messageType\": \"ConfigurationItemChangeNotification\"},  
    "ruleParameters": {\"myParameterKey\": \"myParameterValue\"},  
    "resultToken": \"myResultToken\",  
    "eventLeftScope": false,  
    "executionRoleArn": \"arn:aws:iam::111122223333:role/config-role\",  
    "configRuleArn": \"arn:aws:config:us-east-1:111122223333:config-rule/config-rule-0123456\",  
    \"configRuleName\": \"change-triggered-config-rule\",  
    \"configRuleId\": \"config-rule-0123456\",  
    \"accountId\": \"111122223333\",  
    \"version\": \"1.0\"  
}
```

For more information, see [Evaluating resources with AWS Config rules](#).

Using Lambda with Amazon Connect

You can use a Lambda function to process requests from Amazon Connect. You can use Amazon Connect to create a cloud contact center.

Amazon Connect invokes your Lambda function synchronously with an event that contains the request body and metadata.

Example Amazon Connect request event

```
{  
    "Details": {  
        "ContactData": {  
            "Attributes": {},  
            "Channel": "VOICE",  
            "ContactId": "4a573372-1f28-4e26-b97b-XXXXXXXXXX",  
            "CustomerEndpoint": {  
                "Address": "+1234567890",  
                "Type": "TELEPHONE_NUMBER"  
            },  
            "InitialContactId": "4a573372-1f28-4e26-b97b-XXXXXXXXXX",  
            "InitiationMethod": "INBOUND | OUTBOUND | TRANSFER | CALLBACK",  
            "InstanceARN": "arn:aws:connect:aws-region:1234567890:instance/  
c8c0e68d-2200-4265-82c0-XXXXXXXXXX",  
            "PreviousContactId": "4a573372-1f28-4e26-b97b-XXXXXXXXXX",  
            "Queue": {  
                "ARN": "arn:aws:connect:eu-west-2:111111111111:instance/cccccccc-bbbb-dddd-  
eeee-ffffffffffff/queue/aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeee",  
                "Name": "PasswordReset"  
            },  
            "SystemEndpoint": {  
                "Address": "+1234567890",  
                "Type": "TELEPHONE_NUMBER"  
            }  
        },  
        "Parameters": {  
            "sentAttributeKey": "sentAttributeValue"  
        }  
    },  
    "Name": "ContactFlowEvent"  
}
```

For information about how to use Amazon Connect with Lambda, see [Invoke Lambda functions](#) in the *Amazon Connect administrator guide*.

Using AWS Lambda with Amazon DocumentDB

You can use an AWS Lambda function to process events in a [Amazon DocumentDB \(with MongoDB compatibility\) change stream](#) by configuring an Amazon DocumentDB cluster as an event source. Then, you can automate event-driven workloads by invoking your Lambda function each time data changes with your Amazon DocumentDB cluster.

Note

Lambda supports version 4.0 of Amazon DocumentDB only. Lambda doesn't support versions 3.6 and 5.0.

Also, Lambda supports instance-based clusters and regional clusters only. Lambda doesn't support [elastic clusters](#) or [global clusters](#).

Lambda processes events from Amazon DocumentDB change streams sequentially in the order in which they arrive. Because of this, your function can handle only one concurrent invocation from DocumentDB at a time. To monitor your function, you can track its [concurrency metrics](#).

Topics

- [Example Amazon DocumentDB event \(p. 610\)](#)
- [Prerequisites and permissions \(p. 611\)](#)
- [Network configuration \(p. 612\)](#)
- [Configuring a DocumentDB change stream as an event source \(p. 612\)](#)
- [Event source mapping API \(p. 613\)](#)
- [Monitoring your DocumentDB event source \(p. 615\)](#)
- [Tutorial: Using AWS Lambda with Amazon DocumentDB Streams \(p. 615\)](#)

Example Amazon DocumentDB event

```
{  
    "eventSourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:canaryclusterb2a659a2-q05tcmqkcl03",  
    "events": [  
        {  
            "event": {  
                "_id": {  
                    "_data": "0163eeb6e70000009010000009000041e1"  
                },  
                "clusterTime": {  
                    "$timestamp": {  
                        "t": 1676588775,  
                        "i": 9  
                    }  
                },  
                "documentKey": {  
                    "_id": {  
                        "$oid": "63eeb6e7d418cd98afb1c1d7"  
                    }  
                },  
                "fullDocument": {  
                    "_id": {  
                        "$oid": "63eeb6e7d418cd98afb1c1d7"  
                    },  
                    "anyField": "sampleValue"  
                },  
                "ns": {  
                    "db": "test_database",  
                    "coll": "test_collection"  
                }  
            }  
        }  
    ]  
}
```

```

        "coll": "test_collection"
    },
    "operationType": "insert"
}
],
"eventSource": "aws:docdb"
}

```

For more information on the types of events and their shapes, see the [MongoDB documentation](#).

Prerequisites and permissions

Before you can use Amazon DocumentDB as an event source for your Lambda function, note the following prerequisites:

- **You must have an existing DocumentDB cluster in the same AWS account and AWS Region as your function.** If you don't have an existing cluster, you can create one by following the steps in [Get Started with Amazon DocumentDB](#). Alternatively, the first set of steps in [Tutorial: Using AWS Lambda with Amazon DocumentDB Streams \(p. 615\)](#) guide you through creating a DocumentDB cluster with all the necessary prerequisites.
- **Lambda must have access to the Amazon Virtual Private Cloud (Amazon VPC) resources associated with your DocumentDB cluster.** For more information, see [Network configuration \(p. 612\)](#).
- **You must enable TLS on your DocumentDB cluster (this is the default setting).** If you disable TLS, Lambda cannot communicate with your cluster. Re-enable TLS to restore connectivity with Lambda.
- **You must activate change streams on your DocumentDB cluster.** For more information, see [Using Change Streams with Amazon DocumentDB](#).
- **During setup, you must provide credentials to access your DocumentDB cluster to Lambda.** You do this by providing the [AWS Secrets Manager](#) key that contains the authentication details (username and password) required to access your cluster.
 - If you're setting up the event source using the console, provide this key in the **Secrets manager key** field.
 - If you're setting up the event source using the AWS Command Line Interface (AWS CLI), provide this key in the **SourceAccessConfigurations** field in the [CreateEventSourceMapping \(p. 1153\)](#) or [UpdateEventSourceMapping \(p. 1356\)](#) APIs. For example:

```

aws lambda create-event-source-mapping \
...
--source-access-configurations
'[{"Type": "BASIC_AUTH", "URI": "arn:aws:secretsmanager:us-
west-2:123456789012:secret:DocDBSecret-AbC4E6"}]' \
...

```

- **You must grant permissions to Lambda to manage resources related to your DocumentDB stream.** Manually add the following permissions to your function's [execution role \(p. 816\)](#), or see [the section called "Create the execution role" \(p. 626\)](#) for a sample IAM policy containing these permissions:

- [rds:DescribeDBClusters](#)
- [rds:DescribeDBClusterParameters](#)
- [rds:DescribeDBSubnetGroups](#)
- [ec2:CreateNetworkInterface](#)
- [ec2:DescribeNetworkInterfaces](#)
- [ec2:DescribeVpcs](#)
- [ec2:DeleteNetworkInterface](#)
- [ec2:DescribeSubnets](#)

- [ec2:DescribeSecurityGroups](#)
 - [kms:Decrypt](#)
 - [secretsmanager:GetSecretValue](#)
- **Each DocumentDB change stream event that you send to Lambda cannot exceed 6MB in size.**
Lambda only supports payloads of up to 6MB. If your change stream tries to send Lambda an event larger than 6MB, Lambda drops the message and emits the `OversizedRecordCount` metric. Lambda emits all metrics on a best-effort basis.

Network configuration

Lambda must have access to the Amazon VPC resources associated with your DocumentDB cluster. To do this, you must deploy two [interface VPC endpoints](#): one for Lambda, and one for Secrets Manager. For detailed steps on how to do this, see [the section called "Create interface VPC endpoints" \(p. 625\)](#).

Configure your Amazon VPC security groups with the following rules (at minimum):

- Inbound rules – Allow all traffic on the DocumentDB cluster port (27017) for the security groups specified for your event source.
- Outbound rules – Allow all traffic on port 443 for all destinations. Allow all traffic on the DocumentDB cluster port (27017) for the security groups specified for your event source.
- If you are using VPC endpoints instead of a NAT gateway, the security groups associated with the VPC endpoints must allow all inbound traffic on port 443 from the event source's security groups.

See [the section called "Create the EC2 security group" \(p. 619\)](#) for a sample procedure on how to create the security group.

Configuring a DocumentDB change stream as an event source

To configure a function to read from a DocumentDB cluster's change stream, create a DocumentDB [event source mapping \(p. 131\)](#). This section describes how to do this from the console. For information on configuring a DocumentDB change stream using the AWS CLI, see [event source mapping API](#).

To create a DocumentDB event source mapping (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of a function.
3. Under **Function overview**, choose **Add trigger**.
4. In the dropdown menu, choose **DocumentDB**.
5. Configure the required options, and then choose **Add**.

Lambda supports the following options for DocumentDB event sources:

- **Activate trigger immediately** – Choose whether you want to activate the trigger now. If you check this box, your function immediately starts receiving traffic from the specified DocumentDB change stream upon creation of the event source mapping. We recommend unchecking the box to create the event source mapping in a deactivated state for testing. After creation, you can activate the event source mapping at any time.
- **DocumentDB cluster** – Select a DocumentDB cluster.
- **Database name** – Enter the name of a database within the cluster to consume.

- **Collection name** – Enter the name of a collection within the database to consume. If you don't specify a collection, Lambda listens to all events from each collection in the database.
- **Authentication** – Choose the authentication method for accessing the brokers in your cluster.
 - **BASIC_AUTH** – With basic authentication, you must provide the Secrets Manager key that contains the credentials to access your cluster.
- **Secrets Manager key** – Choose the Secrets Manager key that contains the authentication details (username and password) required to access your DocumentDB cluster.
- **Batch size** – Set the maximum number of messages to retrieve in a single batch, up to 10,000. The default batch size is 100.
- **Starting position** – Choose the position in the stream to start reading records from.
 - **Latest** – Process only new records that are added to the stream. Your function starts processing records only after Lambda finishes creating your event source. This means some records may be dropped until your event source is created successfully.
 - **Trim horizon** – Process all records in the stream. Lambda uses the log retention duration of your cluster to determine where to start reading events from. Specifically, Lambda starts reading from `current_time - log_retention_duration`. Your change stream must already be active before this timestamp for Lambda to properly read all events.
 - **At timestamp** – Process records starting from a specific time. Your change stream must already be active before the specified timestamp for Lambda to properly read all events.
- **Full document configuration** – For document update operations, choose what you want to send to the stream. The default value is `Default`, which means that for each change stream event, DocumentDB sends only a delta describing the changes made. For more information about this field, see [FullDocument](#) in the MongoDB Javadocs.
 - **Default** – Lambda sends only a partial document describing the changes made.
 - **UpdateLookup** – Lambda sends a delta describing the changes, along with a copy of the entire document.

Event source mapping API

To manage your DocumentDB event source with the [AWS CLI](#) or an [AWS SDK](#), you can use the following API operations:

- [CreateEventSourceMapping \(p. 1153\)](#)
- [ListEventSourceMappings \(p. 1279\)](#)
- [GetEventSourceMapping \(p. 1214\)](#)
- [UpdateEventSourceMapping \(p. 1356\)](#)
- [DeleteEventSourceMapping \(p. 1186\)](#)

Note

When you update, disable, or delete an event source mapping for Amazon DocumentDB, it can take up to 15 minutes for your changes to take effect. Before this period has elapsed, your event source mapping may continue to process events and invoke your function using your previous settings. This is true even when the status of the event source mapping displayed in the console indicates that your changes have been applied.

To create the event source mapping with the AWS CLI, use the `create-event-source-mapping` command. The following example uses the AWS CLI to map a function named `my-function` to a DocumentDB change stream. The event source is specified by an Amazon Resource Name (ARN), with a batch size of 500, starting from the timestamp in Unix time. The command also specifies the Secrets Manager key that Lambda uses to connect to DocumentDB, as well as `document-db-event-source-config` parameters to specify the database and collection to read from:

```
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"}' \
```

You should see output that looks like this:

```
{
    "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"
}
```

After creation, you can use the update-event-source-mapping command to change update settings related to your DocumentDB event source. For example, the following command updates the batch size to 1,000 and the batch window to 10 seconds. For this command, you need the UUID of your event source mapping, which you can retrieve from a list-event-source-mapping call or the console:

```
aws lambda update-event-source-mapping --function-name my-function \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--batch-size 1000 \
--batch-window 10
```

You should see this output:

```
{
    "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 updates settings asynchronously and you may not see these changes in the output until the process completes. Use the `get-event-source-mapping` command to view the current status.

```
aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b
```

You should see this output:

```
{  
    "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"  
}
```

To delete your DocumentDB event source mapping, use the `delete-event-source-mapping` command:

```
aws lambda delete-event-source-mapping \  
--uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284
```

Monitoring your DocumentDB event source

To help you monitor your DocumentDB event source, Lambda emits the `IteratorAge` metric when your function finishes processing a batch of records. `IteratorAge` is the difference between the timestamp of the most recent event and the current timestamp. Essentially, this metric indicates how old the last processed record in the batch is. If your function is currently processing new events, you can use the iterator age to estimate the latency between when a record is added and when your function processes it.

An increasing trend in `IteratorAge` can indicate issues with your function. For more information, see [Working with Lambda function metrics \(p. 870\)](#).

Lambda only supports payloads of up to 6MB. However, DocumentDB change stream events can be up to 16MB in size. If your change stream tries to send Lambda a change stream event larger than 6MB, Lambda drops the message and emits the `OversizedRecordCount` metric. Lambda emits all metrics on a best-effort basis.

Tutorial: Using AWS Lambda with Amazon DocumentDB Streams

In this tutorial, you create a basic Lambda function that consumes events from an Amazon DocumentDB (with MongoDB compatibility) change stream. To complete this tutorial, you will go through the following stages:

- Set up your Amazon DocumentDB cluster, connect to it, and activate change streams on it.

- Create your Lambda function, and configure your Amazon DocumentDB cluster as an event source for your function.
- Test the end-to-end setup by inserting items into your Amazon DocumentDB database.

Topics

- [Prerequisites \(p. 616\)](#)
- [Create the AWS Cloud9 environment \(p. 618\)](#)
- [Create the EC2 security group \(p. 619\)](#)
- [Create the secret in Secrets Manager \(p. 620\)](#)
- [Create the DocumentDB cluster \(p. 621\)](#)
- [Install the mongo shell \(p. 622\)](#)
- [Connect to the DocumentDB cluster \(p. 623\)](#)
- [Activate change streams \(p. 624\)](#)
- [Create interface VPC endpoints \(p. 625\)](#)
- [Create the execution role \(p. 626\)](#)
- [Create the Lambda function \(p. 628\)](#)
- [Create the Lambda event source mapping \(p. 629\)](#)
- [Test your function - manual invoke \(p. 630\)](#)
- [Test your function - insert a record \(p. 631\)](#)
- [Test your function - update a record \(p. 632\)](#)
- [Test your function - delete a record \(p. 633\)](#)
- [Clean up your resources \(p. 633\)](#)

Prerequisites

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, [assign administrative access to an administrative user](#), and use only the root user to perform [tasks that require root user access](#).

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to <https://aws.amazon.com/> and choosing **My Account**.

Create an administrative user

After you sign up for an AWS account, create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the [AWS Management Console](#) as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

For help signing in by using root user, see [Signing in as the root user](#) in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

For instructions, see [Enable a virtual MFA device for your AWS account root user \(console\)](#) in the *IAM User Guide*.

Create an administrative user

- For your daily administrative tasks, grant administrative access to an administrative user in AWS IAM Identity Center (successor to AWS Single Sign-On).

For instructions, see [Getting started](#) in the *AWS IAM Identity Center (successor to AWS Single Sign-On) User Guide*.

Sign in as the administrative user

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

For help signing in using an IAM Identity Center user, see [Signing in to the AWS access portal](#) in the *AWS Sign-In User Guide*.

Install the AWS Command Line Interface

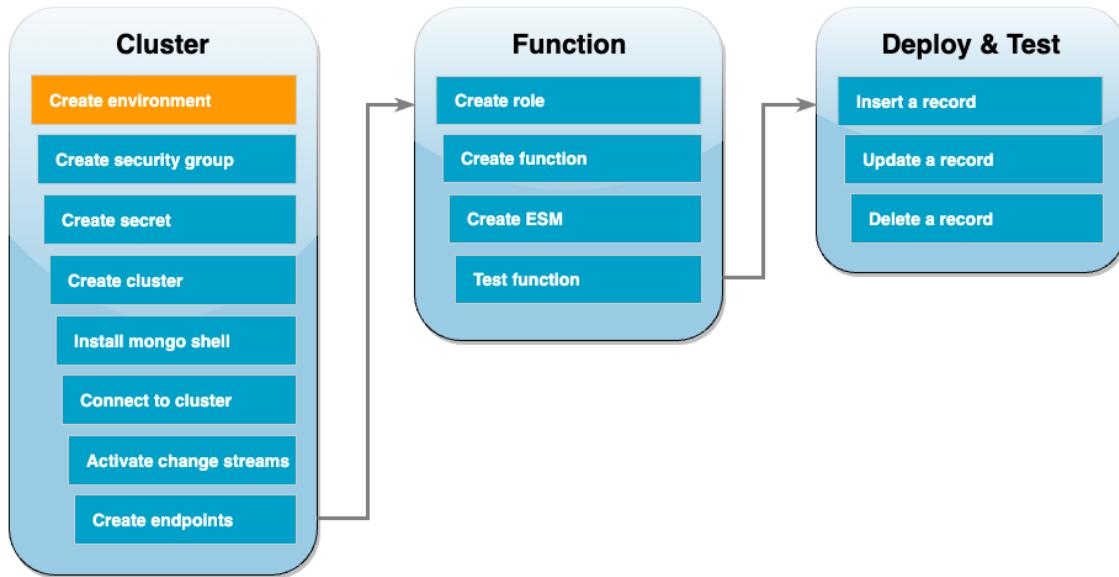
If you have not yet installed the AWS Command Line Interface, follow the steps at [Installing or updating the latest version of the AWS CLI](#) to install it.

The tutorial requires a command line terminal or shell to run commands. In Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#).

Create the AWS Cloud9 environment



Before creating the Lambda function, you need to create and configure your Amazon DocumentDB cluster. The steps to set up your cluster in this tutorial is based on the procedure in [Get Started with Amazon DocumentDB](#).

Note

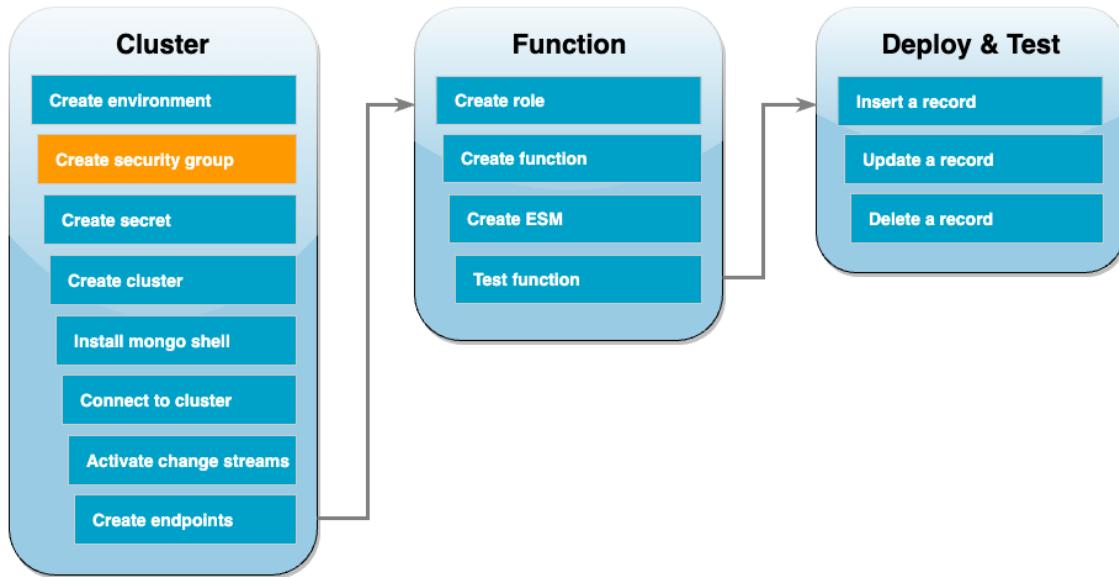
If you already have a Amazon DocumentDB cluster set up, ensure that you activate change streams and create the necessary interface VPC endpoints. Then, you can skip directly to the function creation steps.

First, create an AWS Cloud9 environment. You'll use this environment throughout this tutorial to connect to and query your DocumentDB cluster.

To create an AWS Cloud9 environment

1. Open the [Cloud9 console](#) and choose **Create environment**.
2. Create an environment with the following configuration:
 - Under **Details**:
 - **Name** – DocumentDBCloud9Environment
 - **Environment type** – New EC2 instance
 - Under **New EC2 instance**:
 - **Instance type** – t2.micro (1 GiB RAM + 1 vCPU)
 - **Platform** – Amazon Linux 2
 - **Timeout** – 30 minutes
 - Under **Network settings**:
 - **Connection** – AWS Systems Manager (SSM)
 - Expand the **VPC settings** dropdown.
 - **Amazon Virtual Private Cloud (VPC)** – Choose your [default VPC](#).
 - **Subnet** – No preference
 - Keep all other default settings.
3. Choose **Create**. Provisioning your new AWS Cloud9 environment can take several minutes.

Create the EC2 security group

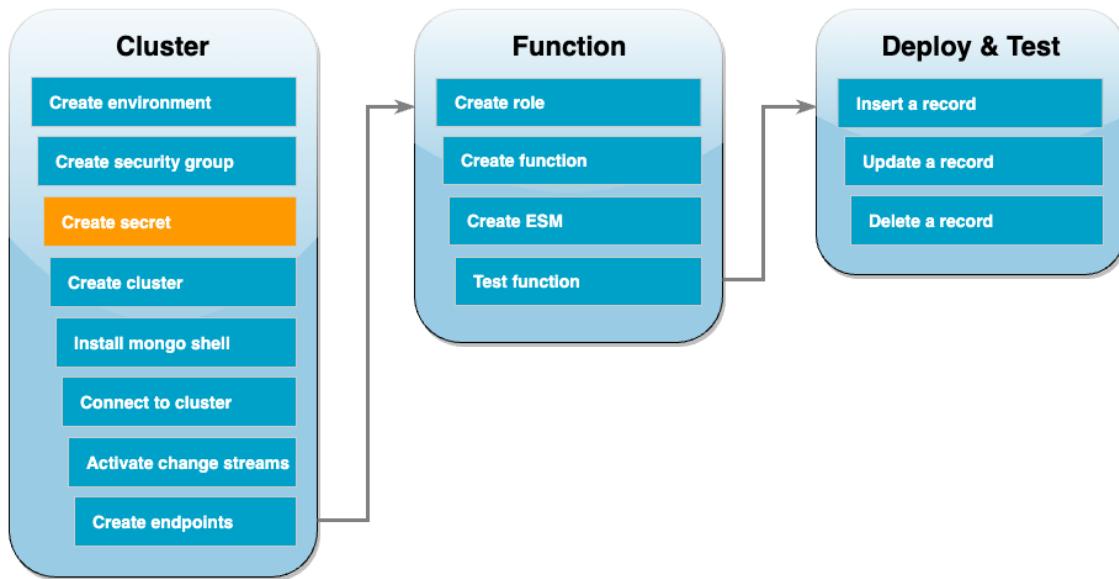


Next, create a [EC2 security group](#) with rules that allow traffic between your DocumentDB cluster and your Cloud9 environment.

To create an EC2 security group

1. Open the [EC2 console](#). Under **Network and Security**, choose **Security groups**.
2. Choose **Create security group**.
3. Create a security group with the following configuration:
 - Under **Basic details**:
 - **Security group name** – DocDBTutorial
 - **Description** – Security group for traffic between Cloud9 and DocumentDB.
 - **VPC** – Choose your [default VPC](#).
 - Under **Inbound rules**, choose **Add rule**. Create a rule with the following configuration:
 - **Type** – Custom TCP
 - **Port range** – 27017
 - **Source** – Custom
 - In the search box next to **Source**, choose the security group for the AWS Cloud9 environment you created in the previous step. To see a list of available security groups, enter cloud9 in the search box. Choose the security group with the name aws-cloud9-<environment_name>.
 - Keep all other default settings.
4. Choose **Create security group**.

Create the secret in Secrets Manager



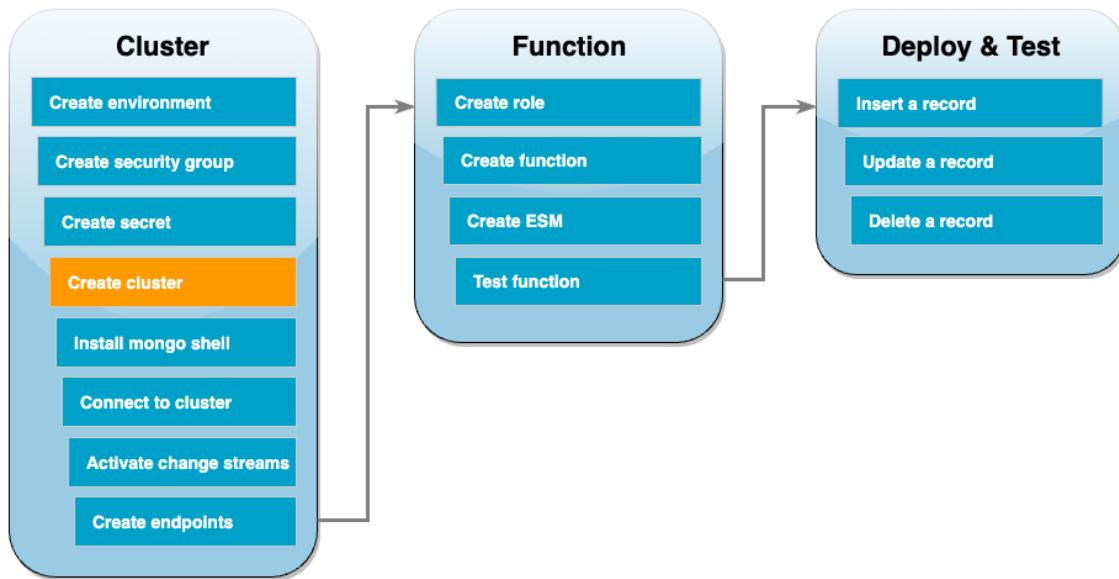
To access your DocumentDB cluster manually, you must provide username and password credentials. For Lambda to access your cluster, you must provide a Secrets Manager secret that contains these same access credentials when setting up your event source mapping. In this step, you'll create this secret.

To create the secret in Secrets Manager

1. Open the [Secrets Manager](#) console and choose **Store a new secret**.
2. For **Choose secret type**, choose the following options:
 - Under **Basic details**:
 - **Secret type** – Credentials for Amazon DocumentDB database
 - Under **Credentials**, enter the username and password you'll use to access your DocumentDB cluster.
 - **Database** – Choose your DocumentDB cluster.
 - Choose **Next**.
3. For **Configure secret**, choose the following options:
 - **Secret name** – DocumentDBSecret
 - Choose **Next**.
4. Choose **Next**.
5. Choose **Store**.
6. Refresh the console to verify that you successfully stored the DocumentDBSecret secret.

Note down the **Secret ARN** of your secret. You'll need it in a later step.

Create the DocumentDB cluster

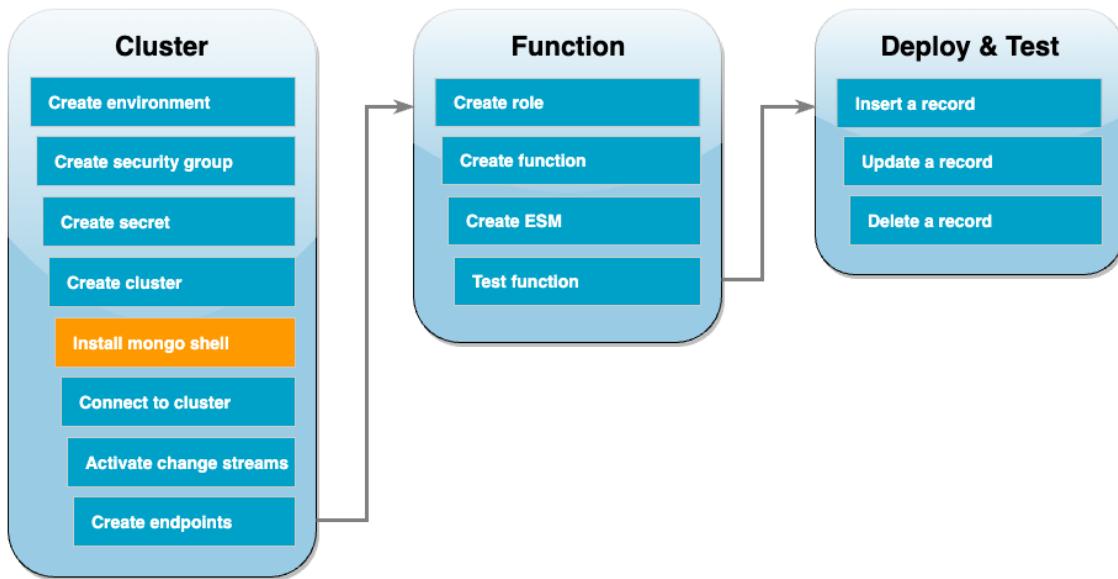


In this step, you'll create a DocumentDB cluster using the security group from the previous step.

To create a DocumentDB cluster

1. Open the [DocumentDB console](#). Under **Clusters**, choose **Create**.
2. Create a cluster with the following configuration:
 - For **Cluster type**, choose Instance Based Cluster.
 - Under **Configuration**:
 - **Engine version** – 4.0.0
 - **Instance class** – db.t3.medium
 - **Number of instances** – 1.
 - Under **Authentication**:
 - Enter the **Username** and **Password** needed to connect to your cluster (same credentials as you used to create the secret in the previous step). In **Confirm password**, confirm your password.
 - Toggle on **Show advanced settings**.
 - Under **Network settings**:
 - **Virtual Private Cloud (VPC)** – Choose your [default VPC](#).
 - **Subnet group** – default
 - **VPC security groups** – In addition to default (VPC), choose the DocDBTutorial (VPC) security group you created in the previous step.
 - Keep all other default settings.
3. Choose **Create cluster**. Provisioning your DocumentDB cluster can take several minutes.

Install the mongo shell



In this step, you'll install the mongo shell in your Cloud9 environment. The mongo shell is a command-line utility that you use to connect to and query your DocumentDB cluster.

To install the mongo shell on your Cloud9 environment

1. Open the [Cloud9 console](#). Next to the DocumentDBCloud9Environment environment you created earlier, click on the **Open** link under the **Cloud9 IDE** column.
2. In the terminal window, create the MongoDB repository file with the following command:

```
echo -e "[mongodb-org-4.0]\nname=MongoDB Repository\nbaseurl=https://repo.mongodb.org/yum/amazon/2013.03/mongodb-org/4.0/x86_64/\npgpcheck=1\nenabled=1\npgpkey=https://www.mongodb.org/static/pgp/server-4.0.asc" | sudo tee /etc/yum.repos.d/mongodb-org-4.0.repo
```

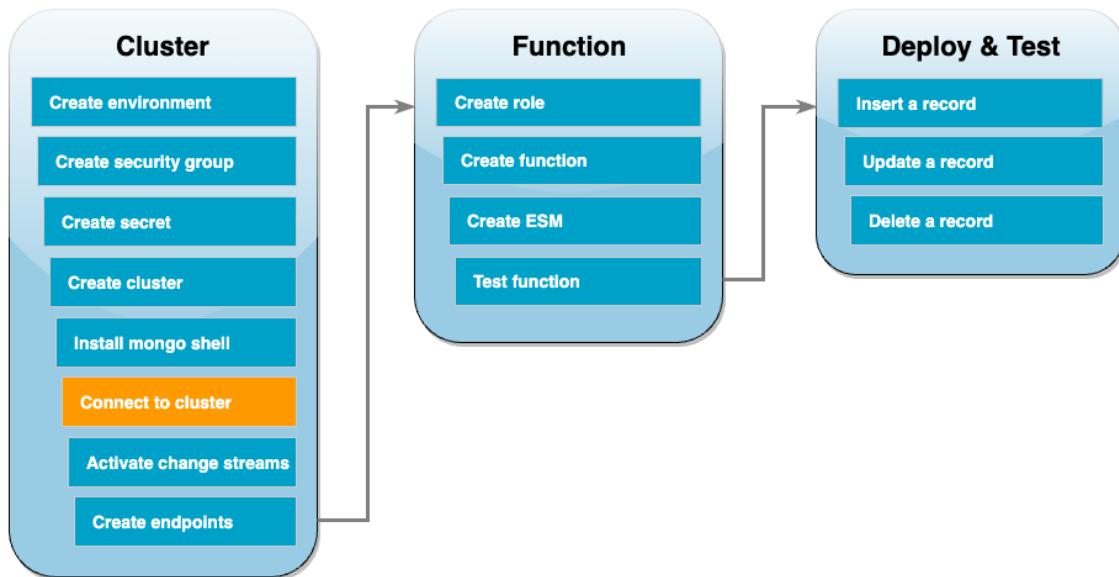
3. Then, install the mongo shell with the following command:

```
sudo yum install -y mongodb-org-shell
```

4. To encrypt data in transit, download the [public key for Amazon DocumentDB](#). The following command downloads a file named global-bundle.pem:

```
wget https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem
```

Connect to the DocumentDB cluster



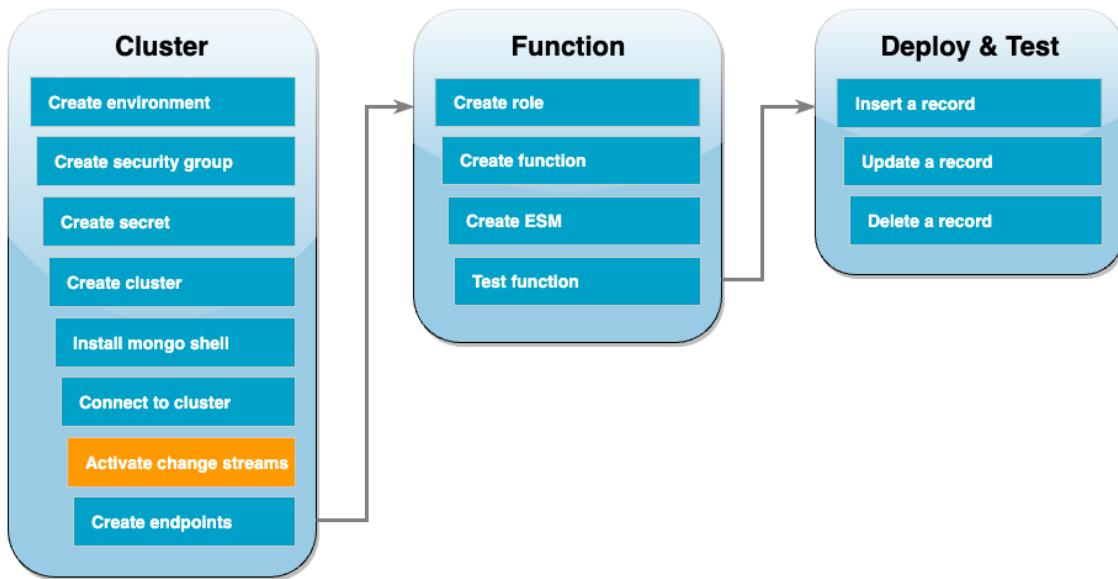
You're now ready to connect to your DocumentDB cluster using the mongo shell.

To connect to your DocumentDB cluster

1. Open the [DocumentDB console](#). Under **Clusters**, choose your cluster by choosing its cluster identifier.
2. In the **Connectivity & security** tab, under **Connect to this cluster with the mongo shell**, choose **Copy**.
3. In your Cloud9 environment, paste this command into the terminal. Replace <insertYourPassword> with the correct password.

After entering this command, if the command prompt becomes `rs0:PRIMARY>`, then you're connected to your Amazon DocumentDB cluster.

Activate change streams



For this tutorial, you'll track changes to the products collection of the docdbdemo database in your DocumentDB cluster. You do this by activating [change streams](#). First, create the docdbdemo database and test it by inserting a record.

To create a new database within your cluster

1. In your Cloud9 environment, ensure that you're still [connected to your DocumentDB cluster \(p. 623\)](#).
2. In the terminal window, use the following command to create a new database called docdbdemo:

```
use docdbdemo
```

3. Then, use the following command to insert a record into docdbdemo:

```
db.products.insert({ "hello": "world" })
```

You should see output that looks like this:

```
WriteResult({ "nInserted" : 1 })
```

4. Use the following command to list all databases:

```
show dbs
```

Ensure that your output contains the docdbdemo database:

```
docdbdemo 0.000GB
```

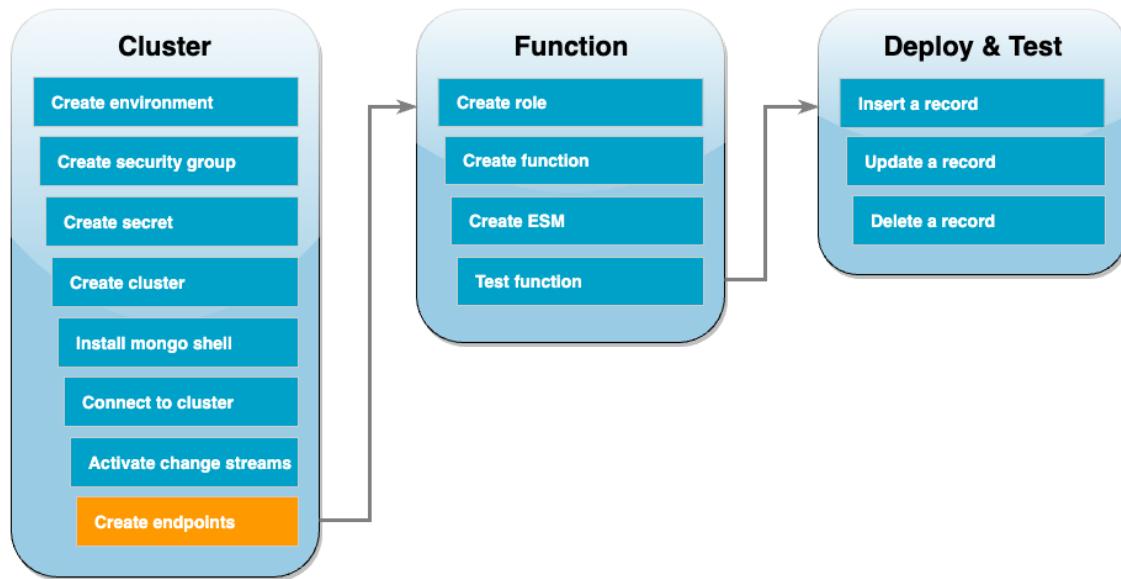
Next, activate change streams on the products collection of the docdbdemo database using the following command:

```
db.adminCommand({modifyChangeStreams: 1,  
  database: "docdbdemo",  
  collection: "products",  
  enable: true});
```

You should see output that looks like this:

```
{ "ok" : 1, "operationTime" : Timestamp(1680126165, 1) }
```

Create interface VPC endpoints



Next, create [interface VPC endpoints](#) to ensure that Lambda and Secrets Manager (used later to store our cluster access credentials) can connect to your default VPC.

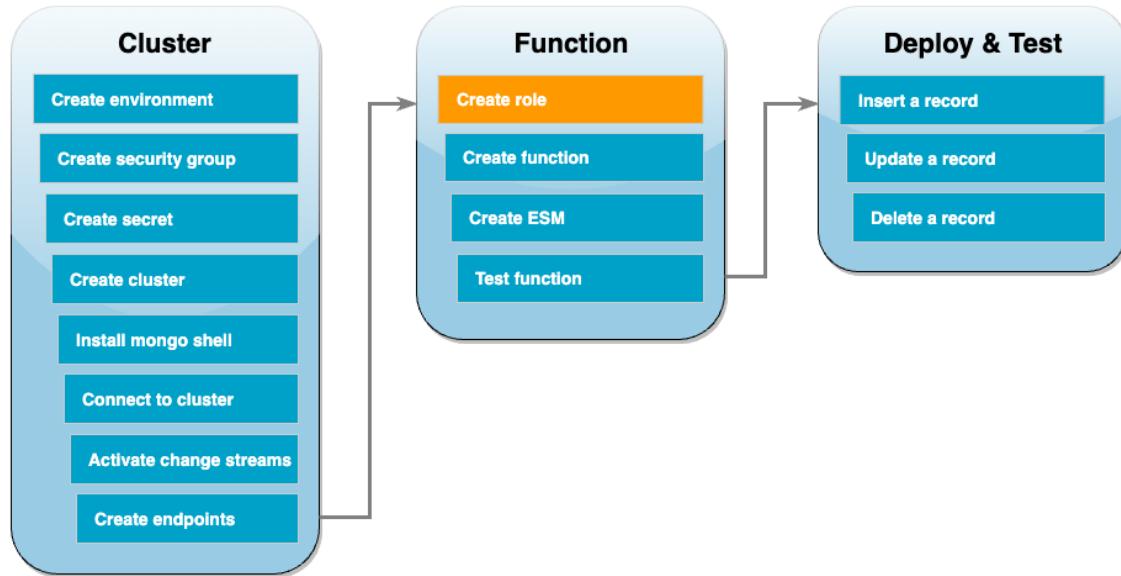
To create interface VPC endpoints

1. Open the [VPC console](#). In the left menu, under **Virtual private cloud**, choose **Endpoints**.
2. Choose **Create endpoint**. Create an endpoint with the following configuration:
 - For **Name tag**, enter lambda-default-vpc.
 - For **Service category**, choose AWS services.
 - For **Services**, enter lambda in the search box. Choose the service with format com.amazonaws.<region>.lambda.
 - For **VPC**, choose your [default VPC](#).
 - For **Subnets**, check the boxes next to each availability zone. Choose the correct subnet ID for each availability zone.
 - For **IP address type**, select IPv4.
 - For **Security groups**, choose the default VPC security group (Group name of default), and the security group you created earlier (Group name of DocDBTutorial).
 - Keep all other default settings.
 - Choose **Create endpoint**.
3. Again, choose **Create endpoint**. Create an endpoint with the following configuration:

- For **Name tag**, enter secretsmanager-default-vpc.
- For **Service category**, choose AWS services.
- For **Services**, enter secretsmanager in the search box. Choose the service with format com.amazonaws.<region>.secretsmanager.
- For **VPC**, choose your [default VPC](#).
- For **Subnets**, check the boxes next to each availability zone. Choose the correct subnet ID for each availability zone.
- For **IP address type**, select IPv4.
- For **Security groups**, choose the default VPC security group (Group name of default), and the security group you created earlier (Group name of DocDBTutorial).
- Keep all other default settings.
- Choose **Create endpoint**.

This completes the cluster setup portion of this tutorial.

Create the execution role



In the next set of steps, you'll create your Lambda function. First, you need to create the execution role that gives your function permission to access your cluster. You do this by creating an IAM policy first, then attaching this policy to an IAM role.

To create IAM policy

1. Open the [Policies page](#) in the IAM console and choose **Create policy**.
2. Choose the **JSON tab**. In the following policy, replace the Secrets Manager resource ARN in the final line of the statement with your secret ARN from earlier, and copy the policy into the editor.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "LambdaESMNetworkingAccess",
            "Effect": "Allow",
            "Action": "secretsmanager:GetSecretValue",
            "Resource": "arn:aws:secretsmanager:<region>:123456789012:secret:MySecret"
        }
    ]
}
```

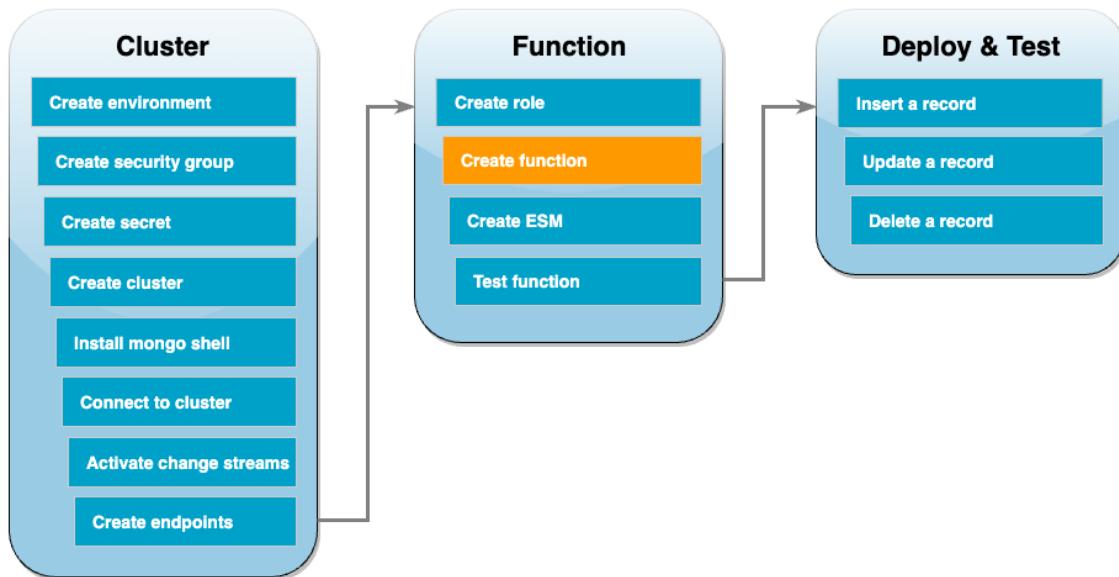
```
"Effect": "Allow",
"Action": [
    "ec2:CreateNetworkInterface",
    "ec2:DescribeNetworkInterfaces",
    "ec2:DescribeVpcs",
    "ec2:DeleteNetworkInterface",
    "ec2:DescribeSubnets",
    "ec2:DescribeSecurityGroups",
    "kms:Decrypt"
],
"Resource": "*"
},
{
    "Sid": "LambdaDocDBESMAccess",
    "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. Choose **Next: Tags**, then choose **Next: Review**.
4. For **Name**, enter AWSDocumentDBLambdaPolicy.
5. Choose **Create policy**.

To create the IAM role

1. Open the [Roles page](#) in the IAM console and choose **Create role**.
2. For **Select trusted entity**, choose the following options:
 - **Trusted entity type** – AWS service
 - **Use case** – Lambda
 - Choose **Next**.
3. For **Add permissions**, choose the AWSDocumentDBLambdaPolicy policy you just created, as well as the AWSLambdaBasicExecutionRole to give your function permissions to write to Amazon CloudWatch Logs.
4. Choose **Next**.
5. For **Role name**, enter AWSDocumentDBLambdaExecutionRole.
6. Choose **Create role**.

Create the Lambda function



The following example code receives a DocumentDB event input and processes the message that it contains.

```
console.log('Loading function');
exports.handler = async (event, context) =>
{
    console.log('Received event:', JSON.stringify(event, null, 2));
    return 'OK';
};
```

To create the Lambda function

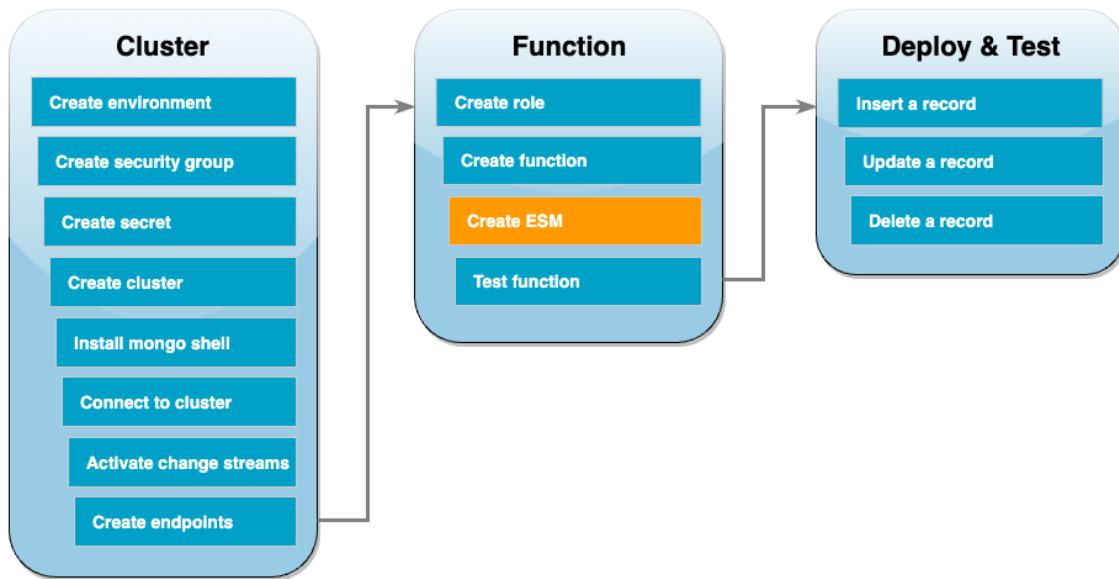
1. Copy the sample code into a file named `index.js`.
2. Create a deployment package with the following command.

```
zip function.zip index.js
```

3. Use the following CLI command to create the function. Replace `us-east-1` with the region, and `123456789012` with your account ID.

```
aws lambda create-function --function-name ProcessDocumentDBRecords \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs16.x \
--region us-east-1 \
--role arn:aws:iam::123456789012:role/AWSDocumentDBLambdaExecutionRole
```

Create the Lambda event source mapping

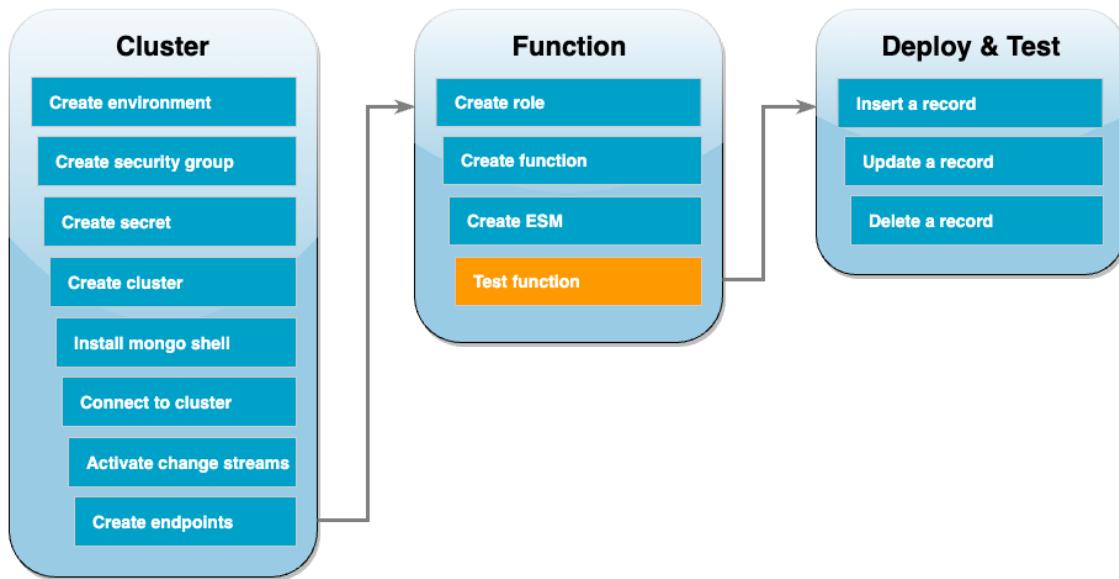


Create the event source mapping that associates your DocumentDB change stream with your Lambda function. After you create this event source mapping, AWS Lambda immediately starts polling the stream.

To create the event source mapping

1. Open the [Functions page](#) in the Lambda console.
2. Choose the `ProcessDocumentDBRecords` function you created earlier.
3. Choose the **Configuration** tab, then choose **Triggers** in the left menu.
4. Choose **Add trigger**.
5. Under **Trigger configuration**, for the source, select **DocumentDB**.
6. Create the event source mapping with the following configuration:
 - **DocumentDB cluster** – Choose the cluster you created earlier.
 - **Database name** – `docdbdemo`
 - **Collection name** – `products`
 - **Batch size** – 1
 - **Starting position** – Latest
 - **Authentication** – `BASIC_AUTH`
 - **Secrets Manager key** – Choose the `DocumentDBSecret` you just created.
 - **Batch window** – 1
 - **Full document configuration** – `UpdateLookup`
7. Choose **Add**. Creating your event source mapping can take a few minutes.

Test your function - manual invoke



To test that you created your function and event source mapping correctly, invoke your function using the `invoke` command. To do this, first copy the following event JSON into a file called `input.txt`:

```
{
  "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": "docdbdemo",
          "coll": "products"
        },
        "operationType": "insert"
      }
    }
  ],
}
```

```
    "eventSource": "aws:docdb"
}
```

Then, use the following command to invoke your function with this event:

```
aws lambda invoke --function-name ProcessDocumentDBRecords \
--cli-binary-format raw-in-base64-out \
--region us-east-1 \
--payload file://input.txt out.txt
```

You should see a response that looks like the following:

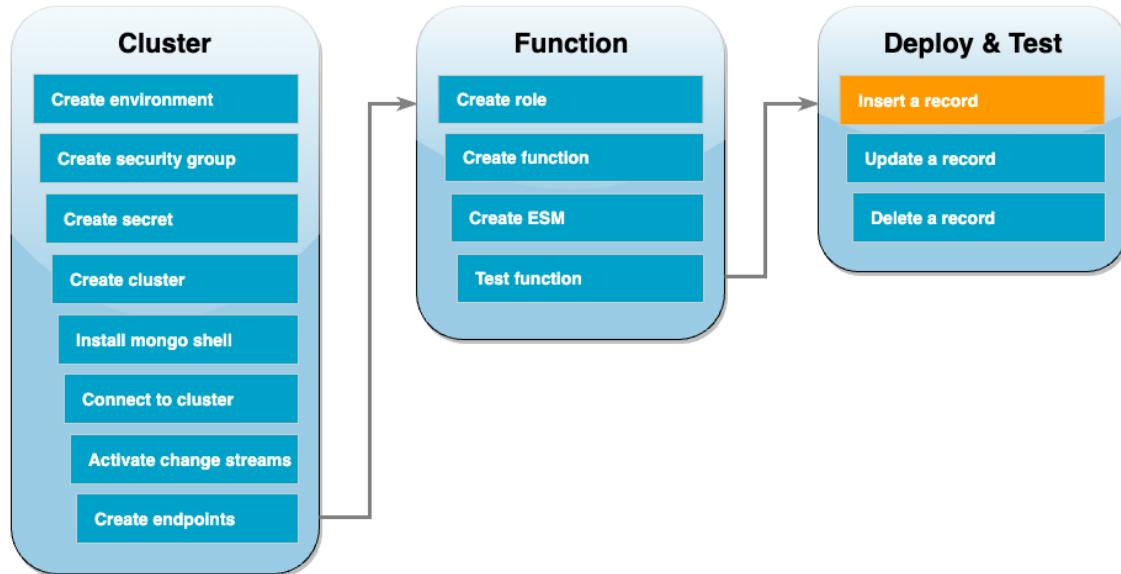
```
{
  "StatusCode": 200,
  "ExecutedVersion": "$LATEST"
}
```

You can verify that your function successfully processed the event by checking CloudWatch Logs.

To verify manual invocation via CloudWatch Logs

1. Open the [Functions page](#) in the Lambda console.
2. Choose the **Monitor** tab, then choose **View CloudWatch logs**. This takes you to the specific log group associated with your function in the CloudWatch console.
3. Choose the most recent log stream. Within the log messages, you should see the event JSON.

Test your function - insert a record



Test your end-to-end setup by interacting directly with your DocumentDB database. In the next set of steps, you'll insert a record, update it, then delete it.

To insert a record

1. [Reconnect to your DocumentDB cluster \(p. 623\)](#) in your Cloud9 environment.

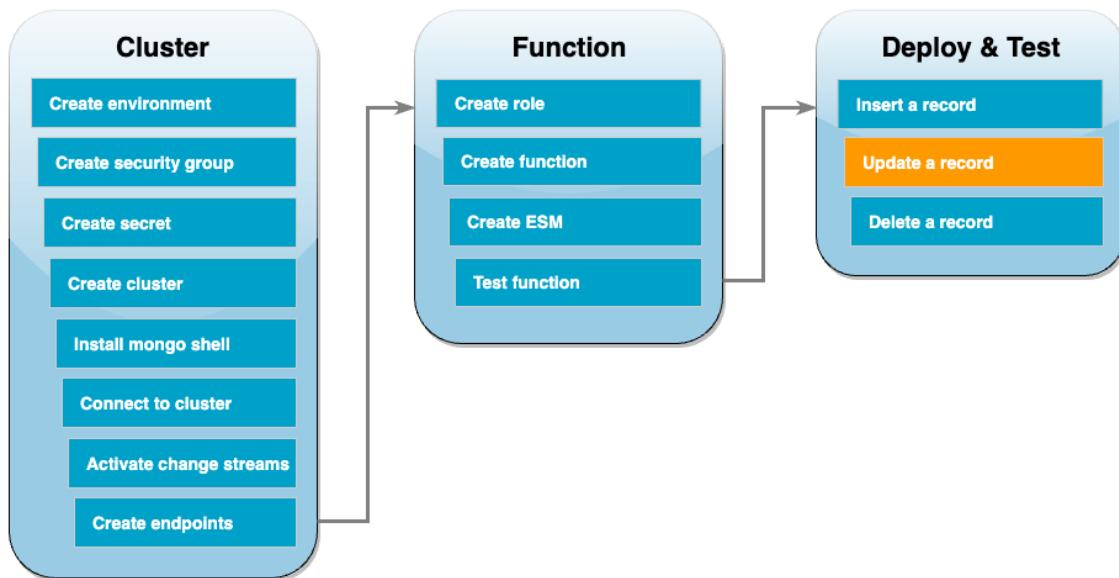
2. Use this command to ensure that you're currently using the docdbdemo database:

```
use docdbdemo
```

3. Insert a record into the products collection of the docdbdemo database:

```
db.products.insert({ "name": "Pencil", "price": 1.00 })
```

Test your function - update a record

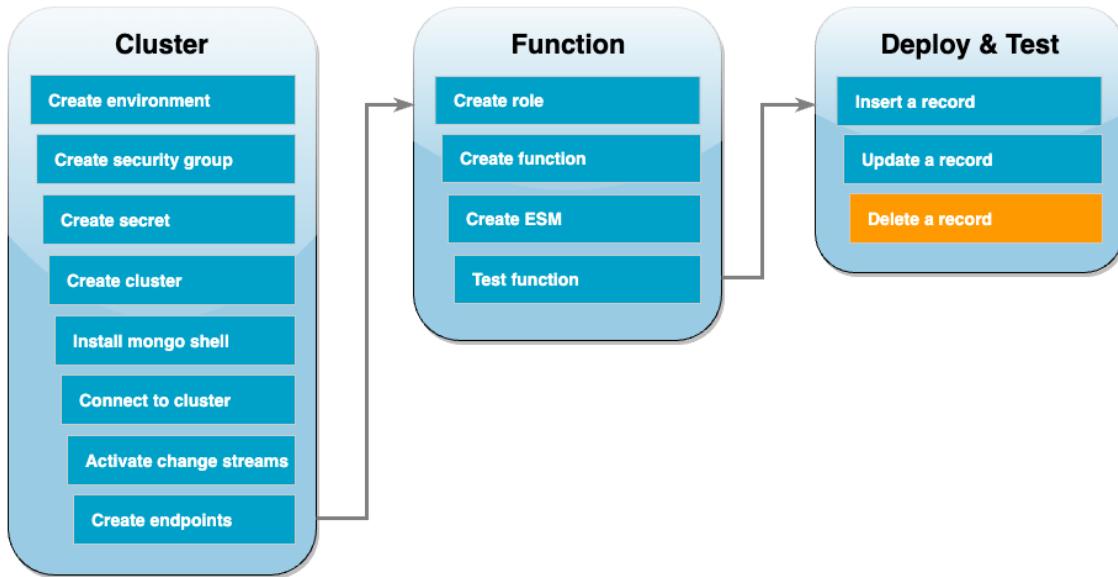


Next, update the record you just inserted with the following command:

```
db.products.update(  
  { "name": "Pencil" },  
  { $set: { "price": 0.50 } }  
)
```

Verify that your function successfully processed this event by checking CloudWatch Logs.

Test your function - delete a record



Finally, delete the record you just updated with the following command:

```
db.products.remove( { "name": "Pencil" } )
```

Verify that your function successfully processed this event by checking CloudWatch Logs.

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

To delete the execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the VPC endpoints

1. Open the [VPC console](#). In the left menu, under **Virtual private cloud**, choose **Endpoints**.
2. Select the endpoints you created.

3. Choose **Actions, Delete VPC endpoints**.
4. Enter **delete** in the text input field.
5. Choose **Delete**.

To delete the Amazon DocumentDB cluster

1. Open the [DocumentDB console](#).
2. Choose the DocumentDB cluster you created for this tutorial, and disable deletion protection.
3. In the main **Clusters** page, choose your DocumentDB cluster again.
4. Choose **Actions, Delete**.
5. For **Create final cluster snapshot**, select **No**.
6. Enter **delete** in the text input field.
7. Choose **Delete**.

To delete the secret in Secrets Manager

1. Open the [Secrets Manager](#) console.
2. Choose the secret you created for this tutorial.
3. Choose **Actions, Delete secret**.
4. Choose **Schedule deletion**.

To delete the Amazon EC2 security group

1. Open the [EC2 console](#). Under **Network and Security**, choose **Security groups**.
2. Select the security group you created for this tutorial.
3. Choose **Actions, Delete security groups**.
4. Choose **Delete**.

To delete the Cloud9 environment

1. Open the [Cloud9 console](#).
2. Select the environment you created for this tutorial.
3. Choose **Delete**.
4. Enter **delete** in the text input field.
5. Choose **Delete**.

Using AWS Lambda with Amazon DynamoDB

Note

If you want to send data to a target other than a Lambda function or enrich the data before sending it, see [Amazon EventBridge Pipes](#).

You can use an AWS Lambda function to process records in an [Amazon DynamoDB stream](#). With DynamoDB Streams, you can trigger a Lambda function to perform additional work each time a DynamoDB table is updated.

Lambda reads records from the stream and invokes your function [synchronously \(p. 120\)](#) with an event that contains stream records. Lambda reads records in batches and invokes your function to process records from the batch.

Sections

- [Example event \(p. 635\)](#)
 - [Polling and batching streams \(p. 636\)](#)
 - [Simultaneous readers of a shard in DynamoDB Streams \(p. 637\)](#)
 - [Execution role permissions \(p. 637\)](#)
 - [Configuring a stream as an event source \(p. 637\)](#)
 - [Event source mapping APIs \(p. 638\)](#)
 - [Error handling \(p. 640\)](#)
 - [Amazon CloudWatch metrics \(p. 641\)](#)
 - [Time windows \(p. 641\)](#)
 - [Reporting batch item failures \(p. 645\)](#)
 - [Amazon DynamoDB Streams configuration parameters \(p. 647\)](#)
 - [Tutorial: Using AWS Lambda with Amazon DynamoDB streams \(p. 648\)](#)
 - [Sample function code \(p. 653\)](#)
 - [AWS SAM template for a DynamoDB application \(p. 656\)](#)

Example event

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-west-2:111122223333:table/TestTable/
stream/2015-05-11T21:21:33.291",
    "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-west-2:111122223333:table/TestTable/
stream/2015-05-11T21:21:33.291",
    "eventSource": "aws:dynamodb"
}
]
}

```

Polling and batching streams

Lambda polls shards in your DynamoDB stream for records at a base rate of 4 times per second. When records are available, Lambda invokes your function and waits for the result. If processing succeeds, Lambda resumes polling until it receives more records.

By default, Lambda invokes your function as soon as records are available. If the batch that Lambda reads from the event source has only one record in it, Lambda sends only one record to the function. To avoid invoking the function with a small number of records, you can tell the event source to buffer records for up to 5 minutes by configuring a *batching window*. Before invoking the function, Lambda continues to read records from the event source until it has gathered a full batch, the batching window expires, or the batch reaches the payload limit of 6 MB. For more information, see [Batching behavior \(p. 132\)](#).

If your function returns an error, Lambda retries the batch until processing succeeds or the data expires. To avoid stalled shards, you can configure the event source mapping to retry with a smaller batch size,

limit the number of retries, or discard records that are too old. To retain discarded events, you can configure the event source mapping to send details about failed batches to a standard SQS queue or standard SNS topic.

You can also increase concurrency by processing multiple batches from each shard in parallel. Lambda can process up to 10 batches in each shard simultaneously. If you increase the number of concurrent batches per shard, Lambda still ensures in-order processing at the partition key level.

Configure the `ParallelizationFactor` setting to process one shard of a Kinesis or DynamoDB data stream with more than one Lambda invocation simultaneously. You can specify the number of concurrent batches that Lambda polls from a shard via a parallelization factor from 1 (default) to 10. For example, when you set `ParallelizationFactor` to 2, you can have 200 concurrent Lambda invocations at maximum to process 100 Kinesis data shards. This helps scale up the processing throughput when the data volume is volatile and the `IteratorAge` is high. Note that parallelization factor will not work if you are using Kinesis aggregation. For more information, see [New AWS Lambda scaling controls for Kinesis and DynamoDB event sources](#). Also, see the [Serverless Data Processing on AWS](#) workshop for complete tutorials.

Simultaneous readers of a shard in DynamoDB Streams

For single-Region tables that are not global tables, you can design for up to two Lambda functions to read from the same DynamoDB Streams shard at the same time. Exceeding this limit can result in request throttling. For global tables, we recommend you limit the number of simultaneous functions to one to avoid request throttling.

Execution role permissions

Lambda needs the following permissions to manage resources related to your DynamoDB stream. Add them to your function's execution role.

- [dynamodb:DescribeStream](#)
- [dynamodb:GetRecords](#)
- [dynamodb:GetShardIterator](#)
- [dynamodb>ListStreams](#)

The `AWSLambdaDynamoDBExecutionRole` managed policy includes these permissions. For more information, see [Lambda execution role \(p. 816\)](#).

To send records of failed batches to a standard SQS queue or standard SNS topic, your function needs additional permissions. Each destination service requires a different permission, as follows:

- **Amazon SQS** – [sq:SendMessage](#)
- **Amazon SNS** – [sns:Publish](#)

Configuring a stream as an event source

Create an event source mapping to tell Lambda to send records from your stream to a Lambda function. You can create multiple event source mappings to process the same data with multiple Lambda functions, or to process items from multiple streams with a single function.

To configure your function to read from DynamoDB Streams in the Lambda console, create a **DynamoDB trigger**.

To create a trigger

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of a function.
3. Under **Function overview**, choose **Add trigger**.
4. Choose a trigger type.
5. Configure the required options, and then choose **Add**.

Lambda supports the following options for DynamoDB event sources.

Event source options

- **DynamoDB table** – The DynamoDB table to read records from.
- **Batch size** – The number of records to send to the function in each batch, up to 10,000. Lambda passes all of the records in the batch to the function in a single call, as long as the total size of the events doesn't exceed the [payload limit \(p. 1131\)](#) for synchronous invocation (6 MB).
- **Batch window** – Specify the maximum amount of time to gather records before invoking the function, in seconds.
- **Starting position** – Process only new records, or all existing records.
 - **Latest** – Process new records that are added to the stream.
 - **Trim horizon** – Process all records in the stream.

After processing any existing records, the function is caught up and continues to process new records.

- **On-failure destination** – A standard SQS queue or standard SNS topic for records that can't be processed. When Lambda discards a batch of records that's too old or has exhausted all retries, Lambda sends details about the batch to the queue or topic.
- **Retry attempts** – The maximum number of times that Lambda retries when the function returns an error. This doesn't apply to service errors or throttles where the batch didn't reach the function.
- **Maximum age of record** – The maximum age of a record that Lambda sends to your function.
- **Split batch on error** – When the function returns an error, split the batch into two before retrying. Your original batch size setting remains unchanged.
- **Concurrent batches per shard** – Concurrently process multiple batches from the same shard.
- **Enabled** – Set to true to enable the event source mapping. Set to false to stop processing records. Lambda keeps track of the last record processed and resumes processing from that point when the mapping is reenabled.

Note

You are not charged for GetRecords API calls invoked by Lambda as part of DynamoDB triggers.

To manage the event source configuration later, choose the trigger in the designer.

Event source mapping APIs

To manage an event source with the [AWS Command Line Interface \(AWS CLI\)](#) or an [AWS SDK](#), you can use the following API operations:

- [CreateEventSourceMapping \(p. 1153\)](#)
- [ListEventSourceMappings \(p. 1279\)](#)
- [GetEventSourceMapping \(p. 1214\)](#)
- [UpdateEventSourceMapping \(p. 1356\)](#)
- [DeleteEventSourceMapping \(p. 1186\)](#)

The following example uses the AWS CLI to map a function named `my-function` to a DynamoDB stream that its Amazon Resource Name (ARN) specifies, with a batch size of 500.

```
aws lambda create-event-source-mapping --function-name my-function --batch-size 500 --  
maximum-batching-window-in-seconds 5 --starting-position LATEST \  
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/  
stream/2019-06-10T19:26:16.525
```

You should see the following output:

```
{  
    "UUID": "14e0db71-5d35-4eb5-b481-8945cf9d10c2",  
    "BatchSize": 500,  
    "MaximumBatchingWindowInSeconds": 5,  
    "ParallelizationFactor": 1,  
    "EventSourceArn": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/  
stream/2019-06-10T19:26:16.525",  
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",  
    "LastModified": 1560209851.963,  
    "LastProcessingResult": "No records processed",  
    "State": "Creating",  
    "StateTransitionReason": "User action",  
    "DestinationConfig": {},  
    "MaximumRecordAgeInSeconds": 604800,  
    "BisectBatchOnFunctionError": false,  
    "MaximumRetryAttempts": 10000  
}
```

Configure additional options to customize how batches are processed and to specify when to discard records that can't be processed. The following example updates an event source mapping to send a failure record to a standard SQS queue after two retry attempts, or if the records are more than an hour old.

```
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:sqs:us-  
east-2:123456789012:dlq"}}'
```

You should see this output:

```
{  
    "UUID": "f89f8514-cdd9-4602-9e1f-01a5b77d449b",  
    "BatchSize": 100,  
    "MaximumBatchingWindowInSeconds": 0,  
    "ParallelizationFactor": 1,  
    "EventSourceArn": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/  
stream/2019-06-10T19:26:16.525",  
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",  
    "LastModified": 1573243620.0,  
    "LastProcessingResult": "PROBLEM: Function call failed",  
    "State": "Updating",  
    "StateTransitionReason": "User action",  
    "DestinationConfig": {},  
    "MaximumRecordAgeInSeconds": 604800,  
    "BisectBatchOnFunctionError": false,  
    "MaximumRetryAttempts": 10000  
}
```

Updated settings are applied asynchronously and aren't reflected in the output until the process completes. Use the `get-event-source-mapping` command to view the current status.

```
aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b
```

You should see this output:

```
{  
    "UUID": "f89f8514-cdd9-4602-9e1f-01a5b77d449b",  
    "BatchSize": 100,  
    "MaximumBatchingWindowInSeconds": 0,  
    "ParallelizationFactor": 1,  
    "EventSourceArn": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/stream/2019-06-10T19:26:16.525",  
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",  
    "LastModified": 1573244760.0,  
    "LastProcessingResult": "PROBLEM: Function call failed",  
    "State": "Enabled",  
    "StateTransitionReason": "User action",  
    "DestinationConfig": {  
        "OnFailure": {  
            "Destination": "arn:aws:sqs:us-east-2:123456789012:dlq"  
        }  
    },  
    "MaximumRecordAgeInSeconds": 3600,  
    "BisectBatchOnFunctionError": false,  
    "MaximumRetryAttempts": 2  
}
```

To process multiple batches concurrently, use the `--parallelization-factor` option.

```
aws lambda update-event-source-mapping --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284 \  
--parallelization-factor 5
```

Error handling

The event source mapping that reads records from your DynamoDB stream, invokes your function synchronously, and retries on errors. If Lambda throttles the function or returns an error without invoking the function, Lambda retries until the records expire or exceed the maximum age that you configure on the event source mapping.

If the function receives the records but returns an error, Lambda retries until the records in the batch expire, exceed the maximum age, or reach the configured retry quota. For function errors, you can also configure the event source mapping to split a failed batch into two batches. Retrying with smaller batches isolates bad records and works around timeout issues. Splitting a batch does not count towards the retry quota.

If the error handling measures fail, Lambda discards the records and continues processing batches from the stream. With the default settings, this means that a bad record can block processing on the affected shard for up to one day. To avoid this, configure your function's event source mapping with a reasonable number of retries and a maximum record age that fits your use case.

To retain a record of discarded batches, configure a failed-event destination. Lambda sends a document to the destination queue or topic with details about the batch.

To configure a destination for failed-event records

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Under **Function overview**, choose **Add destination**.
4. For **Source**, choose **Stream invocation**.

5. For **Stream**, choose a stream that is mapped to the function.
6. For **Destination type**, choose the type of resource that receives the invocation record.
7. For **Destination**, choose a resource.
8. Choose **Save**.

The following example shows an invocation record for a DynamoDB stream.

Example Invocation Record

```
{  
    "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"  
    }  
}
```

You can use this information to retrieve the affected records from the stream for troubleshooting. The actual records aren't included, so you must process this record and retrieve them from the stream before they expire and are lost.

Amazon CloudWatch metrics

Lambda emits the `IteratorAge` metric when your function finishes processing a batch of records. The metric indicates how old the last record in the batch was when processing finished. If your function is processing new events, you can use the iterator age to estimate the latency between when a record is added and when the function processes it.

An increasing trend in iterator age can indicate issues with your function. For more information, see [Working with Lambda function metrics \(p. 870\)](#).

Time windows

Lambda functions can run continuous stream processing applications. A stream represents unbounded data that flows continuously through your application. To analyze information from this continuously updating input, you can bound the included records using a window defined in terms of time.

Tumbling windows are distinct time windows that open and close at regular intervals. By default, Lambda invocations are stateless—you cannot use them for processing data across multiple continuous invocations without an external database. However, with tumbling windows, you can maintain your state

across invocations. This state contains the aggregate result of the messages previously processed for the current window. Your state can be a maximum of 1 MB per shard. If it exceeds that size, Lambda terminates the window early.

Each record in a stream belongs to a specific window. Lambda will process each record at least once, but doesn't guarantee that each record will be processed only once. In rare cases, such as error handling, some records might be processed more than once. Records are always processed in order the first time. If records are processed more than once, they might be processed out of order.

Aggregation and processing

Your user managed function is invoked both for aggregation and for processing the final results of that aggregation. Lambda aggregates all records received in the window. You can receive these records in multiple batches, each as a separate invocation. Each invocation receives a state. Thus, when using tumbling windows, your Lambda function response must contain a `state` property. If the response does not contain a `state` property, Lambda considers this a failed invocation. To satisfy this condition, your function can return a `TimeWindowEventResponse` object, which has the following JSON shape:

Example `TimeWindowEventResponse` values

```
{  
  "state": {  
    "1": 282,  
    "2": 715  
  },  
  "batchItemFailures": []  
}
```

Note

For Java functions, we recommend using a `Map<String, String>` to represent the state.

At the end of the window, the flag `isFinalInvokeForWindow` is set to `true` to indicate that this is the final state and that it's ready for processing. After processing, the window completes and your final invocation completes, and then the state is dropped.

At the end of your window, Lambda uses final processing for actions on the aggregation results. Your final processing is synchronously invoked. After successful invocation, your function checkpoints the sequence number and stream processing continues. If invocation is unsuccessful, your Lambda function suspends further processing until a successful invocation.

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!"  
          }  
        },  
        "OldImage": {}  
      }  
    }  
  ]  
}
```

```

        "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
}

```

Configuration

You can configure tumbling windows when you create or update an [event source mapping \(p. 131\)](#). To configure a tumbling window, specify the window in seconds. The following example AWS Command Line Interface (AWS CLI) command creates a streaming event source mapping that has a tumbling window of 120 seconds. The Lambda function defined for aggregation and processing is named `tumbling-window-example-function`.

```
aws lambda create-event-source-mapping --event-source-arn arn:aws:dynamodb:us-east-1:123456789012:stream/lambda-stream --function-name "arn:aws:lambda:us-east-1:123456789018:function:tumbling-window-example-function" --region us-east-1 --starting-position TRIM_HORIZON --tumbling-window-in-seconds 120
```

Lambda determines tumbling window boundaries based on the time when records were inserted into the stream. All records have an approximate timestamp available that Lambda uses in boundary determinations.

Tumbling window aggregations do not support resharding. When the shard ends, Lambda considers the window closed, and the child shards start their own window in a fresh state.

Tumbling windows fully support the existing retry policies `maxRetryAttempts` and `maxRecordAge`.

Example Handler.py – Aggregation and processing

The following Python function demonstrates how to aggregate and then process your final state:

```

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}
```

Reporting batch item failures

When consuming and processing streaming data from an event source, by default Lambda checkpoints to the highest sequence number of a batch only when the batch is a complete success. Lambda treats all other results as a complete failure and retries processing the batch up to the retry limit. To allow for partial successes while processing batches from a stream, turn on `ReportBatchItemFailures`. Allowing partial successes can help to reduce the number of retries on a record, though it doesn't entirely prevent the possibility of retries in a successful record.

To turn on `ReportBatchItemFailures`, include the enum value `ReportBatchItemFailures` in the `FunctionResponseTypes` list. This list indicates which response types are enabled for your function. You can configure this list when you create or update an [event source mapping \(p. 131\)](#).

Report syntax

When configuring reporting on batch item failures, the `StreamsEventResponse` class is returned with a list of batch item failures. You can use a `StreamsEventResponse` object to return the sequence number of the first failed record in the batch. You can also create your own custom class using the correct response syntax. The following JSON structure shows the required response syntax:

```
{
  "batchItemFailures": [
    {
      "itemIdentifier": "<id>"
    }
  ]
}
```

Note

If the `batchItemFailures` array contains multiple items, Lambda uses the record with the lowest sequence number as the checkpoint. Lambda then retries all records starting from that checkpoint.

Success and failure conditions

Lambda treats a batch as a complete success if you return any of the following:

- An empty `batchItemFailure` list
- A null `batchItemFailure` list
- An empty `EventResponse`
- A null `EventResponse`

Lambda treats a batch as a complete failure if you return any of the following:

- An empty string `itemIdentifier`
- A null `itemIdentifier`
- An `itemIdentifier` with a bad key name

Lambda retries failures based on your retry strategy.

Bisecting a batch

If your invocation fails and `BisectBatchOnFunctionError` is turned on, the batch is bisected regardless of your `ReportBatchItemFailures` setting.

When a partial batch success response is received and both `BisectBatchOnFunctionError` and `ReportBatchItemFailures` are turned on, the batch is bisected at the returned sequence number and Lambda retries only the remaining records.

Java

Example Handler.java – return new StreamsEventResponse()

```
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.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
    Serializable> {

    @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();
    }
}
```

Python

Example Handler.py – return batchItemFailures[]

```
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": []}

```

Amazon DynamoDB Streams configuration parameters

All Lambda event source types share the same [CreateEventSourceMapping \(p. 1153\)](#) and [UpdateEventSourceMapping \(p. 1356\)](#) API operations. However, only some of the parameters apply to DynamoDB Streams.

Event source parameters that apply to DynamoDB Streams

Parameter	Required	Default	Notes
BatchSize	N	100	Maximum: 10,000
BisectBatchOnFunctionError	N	false	
DestinationConfig	N		Standard Amazon SQS queue or standard Amazon SNS topic destination for discarded records
Enabled	N	true	
EventSourceArn	Y		ARN of the data stream or a stream consumer
FilterCriteria	N		
FunctionName	Y		
MaximumBatchingWindowInMilliseconds	N	0	
MaximumRecordAgeInSeconds	N	-1	-1 means infinite: failed records are retried until the record expires Minimum: -1 Maximum: 604800
MaximumRetryAttempts	N	-1	-1 means infinite: failed records are retried until the record expires Minimum: -1 Maximum: 604800

Parameter	Required	Default	Notes
ParallelizationFactor	N	1	Maximum: 10
StartingPosition	Y		TRIM_HORIZON or LATEST
TumblingWindowInSeconds			Minimum: 0 Maximum: 900

Tutorial: Using AWS Lambda with Amazon DynamoDB streams

In this tutorial, you create a Lambda function to consume events from an Amazon DynamoDB stream.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console \(p. 4\)](#) to create your first Lambda function.

To complete the following steps, you need the [AWS Command Line Interface \(AWS CLI\) version 2](#). Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.0.57 Python/3.7.4 Darwin/19.6.0 exe/x86_64
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#). Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

Create the execution role

Create the [execution role \(p. 816\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity** – Lambda.
 - **Permissions** – **AWSLambdaDynamoDBExecutionRole**.

- **Role name – lambda-dynamodb-role.**

The **AWSLambdaDynamoDBExecutionRole** has the permissions that the function needs to read items from DynamoDB and write logs to CloudWatch Logs.

Create the function

The following example code receives a DynamoDB event input and processes the messages that it contains. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Note

For sample code in other languages, see [Sample function code \(p. 653\)](#).

Example index.js

```
console.log('Loading function');

exports.handler = function(event, context, callback) {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        console.log(record.eventID);
        console.log(record.eventName);
        console.log('DynamoDB Record: %j', record.dynamodb);
    });
    callback(null, "message");
};
```

To create the function

1. Copy the sample code into a file named `index.js`.
2. Create a deployment package.

```
zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command.

```
aws lambda create-function --function-name ProcessDynamoDBRecords \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs12.x \
--role arn:aws:iam::123456789012:role/lambda-dynamodb-role
```

Test the Lambda function

In this step, you invoke your Lambda function manually using the `invoke` AWS Lambda CLI command and the following sample DynamoDB event.

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"  
}  
]  
}
```

Run the following invoke command.

```
aws lambda invoke --function-name ProcessDynamoDBRecords --payload file://input.txt  
outputfile.txt
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

The function returns the string message in the response body.

Verify the output in the `outputfile.txt` file.

Create a DynamoDB table with a stream enabled

Create an Amazon DynamoDB table with a stream enabled.

To create a DynamoDB table

1. Open the [DynamoDB console](#).
2. Choose **Create table**.
3. Create a table with the following settings.
 - **Table name** – `lambda-dynamodb-stream`
 - **Primary key** – **id** (string)
4. Choose **Create**.

To enable streams

1. Open the [DynamoDB console](#).
2. Choose **Tables**.
3. Choose the `lambda-dynamodb-stream` table.
4. Under **Exports and streams**, choose **DynamoDB stream details**.
5. Choose **Enable**.
6. Choose **Enable stream**.

Write down the stream ARN. You need this in the next step when you associate the stream with your Lambda function. For more information on enabling streams, see [Capturing table activity with DynamoDB Streams](#).

Add an event source in AWS Lambda

Create an event source mapping in AWS Lambda. This event source mapping associates the DynamoDB stream with your Lambda function. After you create this event source mapping, AWS Lambda starts polling the stream.

Run the following AWS CLI `create-event-source-mapping` command. After the command runs, note down the UUID. You'll need this UUID to refer to the event source mapping in any commands, for example, when deleting the event source mapping.

```
aws lambda create-event-source-mapping --function-name ProcessDynamoDBRecords \
--batch-size 100 --starting-position LATEST --event-source DynamoDB-stream-arn
```

This creates a mapping between the specified DynamoDB stream and the Lambda function. You can associate a DynamoDB stream with multiple Lambda functions, and associate the same Lambda function with multiple streams. However, the Lambda functions will share the read throughput for the stream they share.

You can get the list of event source mappings by running the following command.

```
aws lambda list-event-source-mappings
```

The list returns all of the event source mappings you created, and for each mapping it shows the `LastProcessingResult`, among other things. This field is used to provide an informative message if there are any problems. Values such as `No records processed` (indicates that AWS Lambda has not started polling or that there are no records in the stream) and `OK` (indicates AWS Lambda successfully read records from the stream and invoked your Lambda function) indicate that there are no issues. If there are issues, you receive an error message.

If you have a lot of event source mappings, use the function name parameter to narrow down the results.

```
aws lambda list-event-source-mappings --function-name ProcessDynamoDBRecords
```

Test the setup

Test the end-to-end experience. As you perform table updates, DynamoDB writes event records to the stream. As AWS Lambda polls the stream, it detects new records in the stream and invokes your Lambda function on your behalf by passing events to the function.

1. In the DynamoDB console, add, update, and delete items to the table. DynamoDB writes records of these actions to the stream.
2. AWS Lambda polls the stream and when it detects updates to the stream, it invokes your Lambda function by passing in the event data it finds in the stream.
3. Your function runs and creates logs in Amazon CloudWatch. You can verify the logs reported in the Amazon CloudWatch console.

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.

2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

To delete the execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the DynamoDB table

1. Open the [Tables page](#) of the DynamoDB console.
2. Select the table you created.
3. Choose **Delete**.
4. Enter **delete** in the text box.
5. Choose **Delete table**.

Sample function code

Sample code is available for the following languages.

Topics

- [Node.js \(p. 653\)](#)
- [Java 11 \(p. 654\)](#)
- [C# \(p. 654\)](#)
- [Python 3 \(p. 655\)](#)
- [Go \(p. 655\)](#)

Node.js

The following example processes messages from DynamoDB, and logs their contents.

Example ProcessDynamoDBStream.js

```
console.log('Loading function');

exports.lambda_handler = function(event, context, callback) {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        console.log(record.eventID);
        console.log(record.eventName);
        console.log('DynamoDB Record: %j', record.dynamodb);
    });
    callback(null, "message");
};
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Node.js Lambda functions with .zip file archives \(p. 263\)](#).

Java 11

The following example processes messages from DynamoDB, and logs their contents. `handleRequest` is the handler that AWS Lambda invokes and provides event data. The handler uses the predefined `DynamodbEvent` class, which is defined in the `aws-lambda-java-events` library.

Example DDBEventProcessor.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 com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.DynamodbStreamRecord;

public class DDBEventProcessor implements
    RequestHandler<DynamodbEvent, String> {

    public String handleRequest(DynamodbEvent ddbEvent, Context context) {
        for (DynamodbStreamRecord record : ddbEvent.getRecords()){
            System.out.println(record.eventID());
            System.out.println(record.eventName());
            System.out.println(record.dynamodb().toString());
        }
        return "Successfully processed " + ddbEvent.getRecords().size() + " records.";
    }
}
```

If the handler returns normally without exceptions, Lambda considers the input batch of records as processed successfully and begins reading new records in the stream. If the handler throws an exception, Lambda considers the input batch of records as not processed and invokes the function with the same batch of records again.

Dependencies

- `aws-lambda-java-core`
- `aws-lambda-java-events`

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [Deploy Java Lambda functions with .zip or JAR file archives \(p. 393\)](#).

C#

The following example processes messages from DynamoDB, and logs their contents. `ProcessDynamoEvent` is the handler that AWS Lambda invokes and provides event data. The handler uses the predefined `DynamoDbEvent` class, which is defined in the `Amazon.Lambda.DynamoDBEvents` library.

Example ProcessingDynamoDBStreams.cs

```
using System;
using System.IO;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;
```

```
using Amazon.Lambda.Serialization.Json;

namespace DynamoDBStreams
{
    public class DdbSample
    {
        private static readonly JsonSerializer _jsonSerializer = new JsonSerializer();

        public void ProcessDynamoEvent(DynamoDBEvent dynamoEvent)
        {
            Console.WriteLine($"Beginning to process {dynamoEvent.Records.Count} records...");

            foreach (var record in dynamoEvent.Records)
            {
                Console.WriteLine($"Event ID: {record.EventID}");
                Console.WriteLine($"Event Name: {record.EventName}");

                string streamRecordJson = SerializeObject(record.Dynamodb);
                Console.WriteLine($"DynamoDB Record:");
                Console.WriteLine(streamRecordJson);
            }

            Console.WriteLine("Stream processing complete.");
        }

        private string SerializeObject(object streamRecord)
        {
            using (var ms = new MemoryStream())
            {
                _jsonSerializer.Serialize(streamRecord, ms);
                return Encoding.UTF8.GetString(ms.ToArray());
            }
        }
    }
}
```

Replace the `Program.cs` in a .NET Core project with the above sample. For instructions, see [Deploy C# Lambda functions with .zip file archives \(p. 497\)](#).

Python 3

The following example processes messages from DynamoDB, and logs their contents.

Example `ProcessDynamoDBStream.py`

```
from __future__ import print_function

def lambda_handler(event, context):
    for record in event['Records']:
        print(record['eventID'])
        print(record['eventName'])
    print('Successfully processed %s records.' % str(len(event['Records'])))
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Python Lambda functions with .zip file archives \(p. 324\)](#).

Go

The following example processes messages from DynamoDB, and logs their contents.

Example

```
import (
    "strings"

    "github.com/aws/aws-lambda-go/events"
)

func handleRequest(ctx context.Context, e events.DynamoDBEvent) {

    for _, record := range e.Records {
        fmt.Printf("Processing request data for event ID %s, type %s.\n", record.EventID,
record.EventName)

        // Print new values for attributes of type String
        for name, value := range record.Change.NewImage {
            if value.DataType() == events.DataTypeString {
                fmt.Printf("Attribute name: %s, value: %s\n", name, value.String())
            }
        }
    }
}
```

Build the executable with `go build` and create a deployment package. For instructions, see [Deploy Go Lambda functions with .zip file archives \(p. 460\)](#).

AWS SAM template for a DynamoDB application

You can build this application using [AWS SAM](#). To learn more about creating AWS SAM templates, see [AWS SAM template basics](#) in the [AWS Serverless Application Model Developer Guide](#).

Below is a sample AWS SAM template for the [tutorial application \(p. 648\)](#). Copy the text below to a `.yaml` file and save it next to the ZIP package you created previously. Note that the Handler and Runtime parameter values should match the ones you used when you created the function in the previous section.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  ProcessDynamoDBStream:
    Type: AWS::Serverless::Function
    Properties:
      Handler: handler
      Runtime: runtime
      Policies: AWSLambdaDynamoDBExecutionRole
      Events:
        Stream:
          Type: DynamoDB
          Properties:
            Stream: !GetAtt DynamoDBTable.StreamArn
            BatchSize: 100
            StartingPosition: TRIM_HORIZON

  DynamoDBTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        -AttributeName: id
        AttributeType: S
```

```
KeySchema:  
  - AttributeName: id  
    KeyType: HASH  
ProvisionedThroughput:  
  ReadCapacityUnits: 5  
  WriteCapacityUnits: 5  
StreamSpecification:  
  StreamViewType: NEW_IMAGE
```

For information on how to package and deploy your serverless application using the package and deploy commands, see [Deploying serverless applications](#) in the *AWS Serverless Application Model Developer Guide*.

Using AWS Lambda with Amazon EC2

You can use AWS Lambda to process lifecycle events from Amazon Elastic Compute Cloud and manage Amazon EC2 resources. Amazon EC2 sends events to Amazon EventBridge (CloudWatch Events) for lifecycle events such as when an instance changes state, when an Amazon Elastic Block Store volume snapshot completes, or when a spot instance is scheduled to be terminated. You configure EventBridge (CloudWatch Events) to forward those events to a Lambda function for processing.

EventBridge (CloudWatch Events) invokes your Lambda function asynchronously with the event document from Amazon EC2.

Example instance lifecycle event

```
{  
    "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"  
    }  
}
```

For details on configuring events in EventBridge (CloudWatch Events), see [Using AWS Lambda with Amazon EventBridge \(CloudWatch Events\) \(p. 591\)](#). For an example function that processes Amazon EBS snapshot notifications, see [Amazon EventBridge \(CloudWatch Events\) for Amazon EBS](#) in the Amazon EC2 User Guide for Linux Instances.

You can also use the AWS SDK to manage instances and other resources with the Amazon EC2 API.

Permissions

To process lifecycle events from Amazon EC2, EventBridge (CloudWatch Events) needs permission to invoke your function. This permission comes from the function's [resource-based policy \(p. 832\)](#). If you use the EventBridge (CloudWatch Events) console to configure an event trigger, the console updates the resource-based policy on your behalf. Otherwise, add a statement like the following:

Example resource-based policy statement for Amazon EC2 lifecycle notifications

```
{  
    "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/*"  
        }  
    }  
}
```

}

To add a statement, use the `add-permission` AWS CLI command.

```
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/*'
```

If your function uses the AWS SDK to manage Amazon EC2 resources, add Amazon EC2 permissions to the function's [execution role \(p. 816\)](#).

Tutorial: Configuring a Lambda function to access Amazon ElastiCache in an Amazon VPC

In this tutorial, you do the following:

- Create an Amazon ElastiCache cluster in your default Amazon Virtual Private Cloud. For more information about Amazon ElastiCache, see [Amazon ElastiCache](#).
- Create a Lambda function to access the ElastiCache cluster. When you create the Lambda function, you provide subnet IDs in your Amazon VPC and a VPC security group to allow the Lambda function to access resources in your VPC. For illustration in this tutorial, the Lambda function generates a UUID, writes it to the cache, and retrieves it from the cache.
- Invoke the Lambda function and verify that it accessed the ElastiCache cluster in your VPC.

For details on using Lambda with Amazon VPC, see [Configuring a Lambda function to access resources in a VPC \(p. 222\)](#).

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console \(p. 4\)](#) to create your first Lambda function.

To complete the following steps, you need the [AWS Command Line Interface \(AWS CLI\) version 2](#). Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.0.57 Python/3.7.4 Darwin/19.6.0 exe/x86_64
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#). Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

Create the execution role

Create the [execution role \(p. 816\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity** – Lambda.

- **Permissions – AWSLambdaVPCAccessExecutionRole.**
- **Role name – lambda-vpc-role.**

The **AWSLambdaVPCAccessExecutionRole** has the permissions that the function needs to manage network connections to a VPC.

Create an ElastiCache cluster

Create an ElastiCache cluster in your default VPC.

1. Run the following AWS CLI command to create a Memcached cluster.

```
aws elasticache create-cache-cluster --cache-cluster-id ClusterForLambdaTest --cache-node-type cache.t3.medium --engine memcached --num-cache-nodes 1 --security-group-ids sg-0123a1b123456c1de
```

You can look up the default VPC security group in the VPC console under **Security Groups**. Your example Lambda function will add and retrieve an item from this cluster.

2. Write down the configuration endpoint for the cache cluster that you launched. You can get this from the Amazon ElastiCache console. You will specify this value in your Lambda function code in the next section.

Create a deployment package

The following example Python code reads and writes an item to your ElastiCache cluster.

Example app.py

```
from __future__ import print_function
import time
import uuid
import sys
import socket
import elasticache_auto_discovery
from pymemcache.client.hash import HashClient

#elasticache settings
elasticache_config_endpoint = "your-elasticsearch-cluster-endpoint:port"
nodes = elasticache_auto_discovery.discover(elasticache_config_endpoint)
nodes = map(lambda x: (x[1], int(x[2])), nodes)
memcache_client = HashClient(nodes)

def handler(event, context):
    """
    This function puts into memcache and get from it.
    Memcache is hosted using elasticache
    """

    #Create a random UUID... this will be the sample element we add to the cache.
    uuid_inserted = str(uuid.uuid4())
    #Put the UUID to the cache.
    memcache_client.set('uuid', uuid_inserted)
    #Get item (UUID) from the cache.
    uuid_obtained = memcache_client.get('uuid')
    if str(uuid_obtained) == str(uuid_inserted):
        # this print should go to the CloudWatch Logs and Lambda console.
        print ("Success: Fetched value %s from memcache" %(uuid_inserted))
```

```
    else:
        raise Exception("Value is not the same as we put :(. Expected %s got %s"
%(uuid_inserted, uuid_obtained))

    return "Fetched value from memcache: " + uuid_obtained.decode("utf-8")
```

Dependencies

- [pymemcache](#) – The Lambda function code uses this library to create a HashClient object to set and get items from memcache.
- [elasticache-auto-discovery](#) – The Lambda function uses this library to get the nodes in your Amazon ElastiCache cluster.

Install dependencies with Pip and create a deployment package. For instructions, see [Deploy Python Lambda functions with .zip file archives \(p. 324\)](#).

Create the Lambda function

Create the Lambda function with the `create-function` command.

```
aws lambda create-function --function-name AccessMemCache --timeout 30 --memory-size 1024 \
--zip-file fileb://function.zip --handler app.handler --runtime python3.8 \
--role arn:aws:iam::123456789012:role/lambda-vpc-role \
--vpc-config SubnetIds=subnet-0532bb6758ce7c71f,subnet-
d6b7fda068036e11f,SecurityGroupIds=sg-0897d5f549934c2fb
```

You can find the subnet IDs and the default security group ID of your VPC from the VPC console.

Test the Lambda function

In this step, you invoke the Lambda function manually using the `invoke` command. When the Lambda function runs, it generates a UUID and writes it to the ElastiCache cluster that you specified in your Lambda code. The Lambda function then retrieves the item from the cache.

1. Invoke the Lambda function with the `invoke` command.

```
aws lambda invoke --function-name AccessMemCache output.txt
```

2. Verify that the Lambda function executed successfully as follows:

- Review the `output.txt` file.
- Review the results in the AWS Lambda console.
- Verify the results in CloudWatch Logs.

Now that you have created a Lambda function that accesses an ElastiCache cluster in your VPC, you can have the function invoked in response to events. For information about configuring event sources and examples, see [Using AWS Lambda with other services \(p. 556\)](#).

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

To delete the execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the ElastiCache cluster

1. Open the [Memcached page](#) of the ElastiCache console.
2. Select the cluster you created.
3. Choose **Actions, Delete**.
4. Choose **Delete**.

Using AWS Lambda with an Application Load Balancer

You can use a Lambda function to process requests from an Application Load Balancer. Elastic Load Balancing supports Lambda functions as a target for an Application Load Balancer. Use load balancer rules to route HTTP requests to a function, based on path or header values. Process the request and return an HTTP response from your Lambda function.

Elastic Load Balancing invokes your Lambda function synchronously with an event that contains the request body and metadata.

Example Application Load Balancer request event

```
{
  "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
}
```

Your function processes the event and returns a response document to the load balancer in JSON. Elastic Load Balancing converts the document to an HTTP success or error response and returns it to the user.

Example response document format

```
{
  "statusCode": 200,
  "statusDescription": "200 OK",
  "isBase64Encoded": False,
  "headers": [
    {"Content-Type": "text/html"}
  ],
  "body": "<h1>Hello from Lambda!</h1>"
}
```

To configure an Application Load Balancer as a function trigger, grant Elastic Load Balancing permission to run the function, create a target group that routes requests to the function, and add a rule to the load balancer that sends requests to the target group.

Use the `add-permission` command to add a permission statement to your function's resource-based policy.

```
aws lambda add-permission --function-name alb-function \
--statement-id load-balancer --action "lambda:InvokeFunction" \
--principal elasticloadbalancing.amazonaws.com
```

You should see the following output:

```
{  
    "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\\\"}\"\n}
```

For instructions on configuring the Application Load Balancer listener and target group, see [Lambda functions as a target](#) in the *User Guide for Application Load Balancers*.

Using Amazon EFS with Lambda

Lambda integrates with Amazon Elastic File System (Amazon EFS) to support secure, shared file system access for Lambda applications. You can configure functions to mount a file system during initialization with the NFS protocol over the local network within a VPC. Lambda manages the connection and encrypts all traffic to and from the file system.

The file system and the Lambda function must be in the same region. A Lambda function in one account can mount a file system in a different account. For this scenario, you configure VPC peering between the function VPC and the file system VPC.

Note

To configure a function to connect to a file system, see [Configuring file system access for Lambda functions \(p. 236\)](#).

Amazon EFS supports [file locking](#) to prevent corruption if multiple functions try to write to the same file system at the same time. Locking in Amazon EFS follows the NFS v4.1 protocol for advisory locking, and enables your applications to use both whole file and byte range locks.

Amazon EFS provides options to customize your file system based on your application's need to maintain high performance at scale. There are three primary factors to consider: the number of connections, throughput (in MiB per second), and IOPS.

Quotas

For detail on file system quotas and limits, see [Quotas for Amazon EFS file systems](#) in the [Amazon Elastic File System User Guide](#).

To avoid issues with scaling, throughput, and IOPS, monitor the [metrics](#) that Amazon EFS sends to Amazon CloudWatch. For an overview of monitoring in Amazon EFS, see [Monitoring Amazon EFS](#) in the [Amazon Elastic File System User Guide](#).

Sections

- [Connections \(p. 666\)](#)
- [Throughput \(p. 667\)](#)
- [IOPS \(p. 667\)](#)

Connections

Amazon EFS supports up to 25,000 connections per file system. During initialization, each instance of a function creates a single connection to its file system that persists across invocations. This means that you can reach 25,000 concurrency across one or more functions connected to a file system. To limit the number of connections a function creates, use [reserved concurrency \(p. 210\)](#).

However, when you make changes to your function's code or configuration at scale, there is a temporary increase in the number of function instances beyond the current concurrency. Lambda provisions new instances to handle new requests and there is some delay before old instances close their connections to the file system. To avoid hitting the maximum connections limit during a deployment, use [rolling deployments \(p. 973\)](#). With rolling deployments, you gradually shift traffic to the new version each time you make a change.

If you connect to the same file system from other services such as Amazon EC2, you should also be aware of the scaling behavior of connections in Amazon EFS. A file system supports the creation of up to 3,000 connections in a burst, after which it supports 500 new connections per minute. This matches [burst scaling \(p. 220\)](#) behavior in Lambda, which applies across all functions in a Region. But if you are creating connections outside of Lambda, your functions may not be able to scale at full speed.

To monitor and trigger an alarm on connections, use the ClientConnections metric.

Throughput

At scale, it is also possible to exceed the maximum *throughput* for a file system. In *bursting mode* (the default), a file system has a low baseline throughput that scales linearly with its size. To allow for bursts of activity, the file system is granted burst credits that allow it to use 100 MiB/s or more of throughput. Credits accumulate continually and are expended with every read and write operation. If the file system runs out of credits, it throttles read and write operations beyond the baseline throughput, which can cause invocations to time out.

Note

If you use [provisioned concurrency \(p. 213\)](#), your function can consume burst credits even when idle. With provisioned concurrency, Lambda initializes instances of your function before it is invoked, and recycles instances every few hours. If you use files on an attached file system during initialization, this activity can use all of your burst credits.

To monitor and trigger an alarm on throughput, use the `BurstCreditBalance` metric. It should increase when your function's concurrency is low and decrease when it is high. If it always decreases or does not accumulate enough during low activity to cover peak traffic, you may need to limit your function's concurrency or enable [provisioned throughput](#).

IOPS

Input/output operations per second (IOPS) is a measurement of the number of read and write operations processed by the file system. In general purpose mode, IOPS is limited in favor of lower latency, which is beneficial for most applications.

To monitor and alarm on IOPS in general purpose mode, use the `PercentIOLimit` metric. If this metric reaches 100%, your function can time out waiting for read and write operations to complete.

Using AWS Lambda with AWS IoT

AWS IoT provides secure communication between internet-connected devices (such as sensors) and the AWS Cloud. This makes it possible for you to collect, store, and analyze telemetry data from multiple devices.

You can create AWS IoT rules for your devices to interact with AWS services. The AWS IoT [Rules Engine](#) provides a SQL-based language to select data from message payloads and send the data to other services, such as Amazon S3, Amazon DynamoDB, and AWS Lambda. You define a rule to invoke a Lambda function when you want to invoke another AWS service or a third-party service.

When an incoming IoT message triggers the rule, AWS IoT invokes your Lambda function [asynchronously \(p. 123\)](#) and passes data from the IoT message to the function.

The following example shows a moisture reading from a greenhouse sensor. The **row** and **pos** values identify the location of the sensor. This example event is based on the `greenhouse` type in the [AWS IoT Rules tutorials](#).

Example AWS IoT message event

```
{  
    "row" : "10",  
    "pos" : "23",  
    "moisture" : "75"  
}
```

For asynchronous invocation, Lambda queues the message and [retries \(p. 161\)](#) if your function returns an error. Configure your function with a [destination \(p. 125\)](#) to retain events that your function could not process.

You need to grant permission for the AWS IoT service to invoke your Lambda function. Use the `add-permission` command to add a permission statement to your function's resource-based policy.

```
aws lambda add-permission --function-name my-function \  
--statement-id iot-events --action "lambda:InvokeFunction" --principal iot.amazonaws.com
```

You should see the following output:

```
{  
    "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\""}  
}
```

For more information about how to use Lambda with AWS IoT, see [Creating an AWS Lambda rule](#).

Using AWS Lambda with AWS IoT Events

AWS IoT Events monitors the inputs from multiple IoT sensors and applications to recognize event patterns. Then it takes appropriate actions when events occur. AWS IoT Events receives its inputs as JSON payloads from many sources. AWS IoT Events supports simple events (where each input triggers an event) and complex events (where multiple inputs must occur to trigger the event).

To use AWS IoT Events, you define a detector model, which is a state-machine model of your equipment or process. In addition to states, you define inputs and events for the model. You also define the actions to take when an event occurs. Use a Lambda function for an action when you want to invoke another AWS service (such as Amazon Connect), or take actions in an external application (such as your enterprise resource planning (ERP) application).

When the event occurs, AWS IoT Events invokes your Lambda function asynchronously. It provides information about the detector model and the event that triggered the action. The following example message event is based on the definitions in the AWS IoT Events [simple step-by-step example](#).

Example AWS IoT Events message event

```
{  
  "event":{  
    "eventName": "myChargedEvent",  
    "eventTime": 1567797571647,  
    "payload":{  
      "detector":{  
        "detectorModelName": "AWS_IoTEvents_Hello_World1567793458261",  
        "detectorModelVersion": "4",  
        "keyValue": "100009"  
      },  
      "eventTriggerDetails":{  
        "triggerType": "Message",  
        "inputName": "AWS_IoTEvents_HelloWorld_VoltageInput",  
        "messageId": "64c75a34-068b-4a1d-ae58-c16215dc4efd"  
      },  
      "actionExecutionId": "49f0f32f-1209-38a7-8a76-d6ca49dd0bc4",  
      "state":{  
        "variables": {},  
        "stateName": "Charged",  
        "timers": {}  
      }  
    }  
  }  
}
```

The event that is passed into the Lambda function includes the following fields:

- **eventName** – The name for this event in the detector model.
- **eventTime** – The time that the event occurred.
- **detector** – The name and version of the detector model.
- **eventTriggerDetails** – A description of the input that triggered the event.
- **actionExecutionId** – The unique execution identifier of the action.
- **state** – The state of the detector model when the event occurred.
 - **stateName** – The name of the state in the detector model.
 - **timers** – Any timers that are set in this state.
 - **variables** – Any variable values that are set in this state.

You need to grant permission for the AWS IoT Events service to invoke your Lambda function. Use the add-permission command to add a permission statement to your function's resource-based policy.

```
aws lambda add-permission --function-name my-function \
--statement-id iot-events --action "lambda:InvokeFunction" --principal
iotevents.amazonaws.com
```

You should see the following output:

```
{  
    "Statement": "{\"Sid\":\"iot-events\",\"Effect\":\"Allow\",\"Principal\":{\"Service\":\"iotevents.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",\"Resource\":\"arn:aws:lambda:us-east-1:123456789012:function:my-function\""}  
}
```

For more information about using Lambda with AWS IoT Events, see [Using AWS IoT Events with other services](#).

Using Lambda with self-managed Apache Kafka

Note

If you want to send data to a target other than a Lambda function or enrich the data before sending it, see [Amazon EventBridge Pipes](#).

Lambda supports [Apache Kafka](#) as an [event source \(p. 131\)](#). Apache Kafka is an open-source event streaming platform that supports workloads such as data pipelines and streaming analytics.

You can use the AWS managed Kafka service Amazon Managed Streaming for Apache Kafka (Amazon MSK), or a self-managed Kafka cluster. For details about using Lambda with Amazon MSK, see [Using Lambda with Amazon MSK \(p. 716\)](#).

This topic describes how to use Lambda with a self-managed Kafka cluster. In AWS terminology, a self-managed cluster includes non-AWS hosted Kafka clusters. For example, you can host your Kafka cluster with a cloud provider such as [CloudKarafka](#). You can also use other AWS hosting options for your cluster. For more information, see [Best Practices for Running Apache Kafka on AWS](#) on the AWS Big Data Blog.

Apache Kafka as an event source operates similarly to using Amazon Simple Queue Service (Amazon SQS) or Amazon Kinesis. Lambda internally polls for new messages from the event source and then synchronously invokes the target Lambda function. Lambda reads the messages in batches and provides these to your function as an event payload. The maximum batch size is configurable. (The default is 100 messages.)

For Kafka-based event sources, Lambda supports processing control parameters, such as batching windows and batch size. For more information, see [Batching behavior \(p. 132\)](#).

For an example of how to use self-managed Kafka as an event source, see [Using self-hosted Apache Kafka as an event source for AWS Lambda](#) on the AWS Compute Blog.

Topics

- [Example event \(p. 671\)](#)
- [Kafka cluster authentication \(p. 672\)](#)
- [Managing API access and permissions \(p. 674\)](#)
- [Authentication and authorization errors \(p. 676\)](#)
- [Network configuration \(p. 677\)](#)
- [Adding a Kafka cluster as an event source \(p. 677\)](#)
- [Using a Kafka cluster as an event source \(p. 680\)](#)
- [Auto scaling of the Kafka event source \(p. 680\)](#)
- [Event source API operations \(p. 681\)](#)
- [Event source mapping errors \(p. 681\)](#)
- [Amazon CloudWatch metrics \(p. 681\)](#)
- [Self-managed Apache Kafka configuration parameters \(p. 682\)](#)

Example event

Lambda sends the batch of messages in the event parameter when it invokes your Lambda function. The event payload contains an array of messages. Each array item contains details of the Kafka topic and Kafka partition identifier, together with a timestamp and a base64-encoded message.

```
{  
  "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": [
```

```

"records": [
    "mytopic-0": [
        {
            "topic": "mytopic",
            "partition": 0,
            "offset": 15,
            "timestamp": 1545084650987,
            "timestampType": "CREATE_TIME",
            "key": "abcDEFghiJKLMNOPRstuVWXYZ1234==",
            "value": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers": [
                {
                    "headerKey": [
                        104,
                        101,
                        97,
                        100,
                        101,
                        114,
                        86,
                        97,
                        108,
                        117,
                        101
                    ]
                }
            ]
        }
    ]
}

```

Kafka cluster authentication

Lambda supports several methods to authenticate with your self-managed Apache Kafka cluster. Make sure that you configure the Kafka cluster to use one of these supported authentication methods. For more information about Kafka security, see the [Security](#) section of the Kafka documentation.

VPC access

If only Kafka users within your VPC access your Kafka brokers, you must configure the Kafka event source for Amazon Virtual Private Cloud (Amazon VPC) access.

SASL/SCRAM authentication

Lambda supports Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) authentication with Transport Layer Security (TLS) encryption (SASL_SSL). Lambda sends the encrypted credentials to authenticate with the cluster. Lambda doesn't support SASL/SCRAM with plaintext (SASL_PLAINTEXT). For more information about SASL/SCRAM authentication, see [RFC 5802](#).

Lambda also supports SASL/PLAIN authentication. Because this mechanism uses clear text credentials, the connection to the server must use TLS encryption to ensure that the credentials are protected.

For SASL authentication, you store the sign-in credentials as a secret in AWS Secrets Manager. For more information about using Secrets Manager, see [Tutorial: Create and retrieve a secret](#) in the *AWS Secrets Manager User Guide*.

Important

To use Secrets Manager for authentication, secrets must be stored in the same AWS region as your Lambda function.

Mutual TLS authentication

Mutual TLS (mTLS) provides two-way authentication between the client and server. The client sends a certificate to the server for the server to verify the client, and the server sends a certificate to the client for the client to verify the server.

In self-managed Apache Kafka, Lambda acts as the client. You configure a client certificate (as a secret in Secrets Manager) to authenticate Lambda with your Kafka brokers. The client certificate must be signed by a CA in the server's trust store.

The Kafka cluster sends a server certificate to Lambda to authenticate the Kafka brokers with Lambda. The server certificate can be a public CA certificate or a private CA/self-signed certificate. The public CA certificate must be signed by a certificate authority (CA) that's in the Lambda trust store. For a private CA/self-signed certificate, you configure the server root CA certificate (as a secret in Secrets Manager). Lambda uses the root certificate to verify the Kafka brokers.

For more information about mTLS, see [Introducing mutual TLS authentication for Amazon MSK as an event source](#).

Configuring the client certificate secret

The CLIENT_CERTIFICATE_TLS_AUTH secret requires a certificate field and a private key field. For an encrypted private key, the secret requires a private key password. Both the certificate and private key must be in PEM format.

Note

Lambda supports the [PBES1](#) (but not PBES2) private key encryption algorithms.

The certificate field must contain a list of certificates, beginning with the client certificate, followed by any intermediate certificates, and ending with the root certificate. Each certificate must start on a new line with the following structure:

```
-----BEGIN CERTIFICATE-----  
    <certificate contents>  
-----END CERTIFICATE-----
```

Secrets Manager supports secrets up to 65,536 bytes, which is enough space for long certificate chains.

The private key must be in [PKCS #8](#) format, with the following structure:

```
-----BEGIN PRIVATE KEY-----  
    <private key contents>  
-----END PRIVATE KEY-----
```

For an encrypted private key, use the following structure:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----  
    <private key contents>  
-----END ENCRYPTED PRIVATE KEY-----
```

The following example shows the contents of a secret for mTLS authentication using an encrypted private key. For an encrypted private key, include the private key password in the secret.

```
{  
    "privateKeyPassword": "testpassword",  
    "certificate": "-----BEGIN CERTIFICATE-----
```

```
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvNAQELBQAw
...
j0Lh4/+1HfgxE2KlmII36dg4IMzNjAFEBZiCRoPim040s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHzANBgkqhkiG9w0BAQsFADbb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
"privateKey": "-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzg09IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUxzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

Configuring the server root CA certificate secret

You create this secret if your Kafka brokers use TLS encryption with certificates signed by a private CA. You can use TLS encryption for VPC, SASL/SCRAM, SASL/PLAIN, or mTLS authentication.

The server root CA certificate secret requires a field that contains the Kafka broker's root CA certificate in PEM format. The following example shows the structure of the secret.

```
{
  "certificate": "-----BEGIN CERTIFICATE-----
MIID7zCCAtEgAwIBAgIBADANBgkqhkiG9w0BAQsFADCbMDELMAkGA1UEBhMCVVMx
EDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTC1Njb3R0c2RhGUxJTAjBgNVBAoT
HFN0YXJmaWVsZCBUZWNobm9sb2dpZXMsIEluYy4x0zA5BgNVBAMTM1N0YXJmaWVs
ZCBTZXJ2aNlcycBSb290IEN1cnRpZmljYXR1IEF1dG...
-----END CERTIFICATE-----"
```

Managing API access and permissions

In addition to accessing your self-managed Kafka cluster, your Lambda function needs permissions to perform various API actions. You add these permissions to the function's [execution role \(p. 816\)](#). If your users need access to any API actions, add the required permissions to the identity policy for the AWS Identity and Access Management (IAM) user or role.

Required Lambda function permissions

To create and store logs in a log group in Amazon CloudWatch Logs, your Lambda function must have the following permissions in its execution role:

- [logs:CreateLogGroup](#)
- [logs:CreateLogStream](#)
- [logs:PutLogEvents](#)

Optional Lambda function permissions

Your Lambda function might also need permissions to:

- Describe your Secrets Manager secret.

- Access your AWS Key Management Service (AWS KMS) customer managed key.
- Access your Amazon VPC.

Secrets Manager and AWS KMS permissions

Depending on the type of access control that you're configuring for your Kafka brokers, your Lambda function might need permission to access your Secrets Manager secret or to decrypt your AWS KMS customer managed key. To access these resources, your function's execution role must have the following permissions:

- [secretsmanager:GetSecretValue](#)
- [kms:Decrypt](#)

VPC permissions

If only users within a VPC can access your self-managed Apache Kafka cluster, your Lambda function must have permission to access your Amazon VPC resources. These resources include your VPC, subnets, security groups, and network interfaces. To access these resources, your function's execution role must have the following permissions:

- [ec2>CreateNetworkInterface](#)
- [ec2:DescribeNetworkInterfaces](#)
- [ec2:DescribeVpcs](#)
- [ec2>DeleteNetworkInterface](#)
- [ec2:DescribeSubnets](#)
- [ec2:DescribeSecurityGroups](#)

Adding permissions to your execution role

To access other AWS services that your self-managed Apache Kafka cluster uses, Lambda uses the permissions policies that you define in your Lambda function's [execution role \(p. 816\)](#).

By default, Lambda is not permitted to perform the required or optional actions for a self-managed Apache Kafka cluster. You must create and define these actions in an [IAM trust policy](#), and then attach the policy to your execution role. This example shows how you might create a policy that allows Lambda to access your Amazon VPC resources.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "ec2:CreateNetworkInterface",  
                "ec2:DescribeNetworkInterfaces",  
                "ec2:DescribeVpcs",  
                "ec2>DeleteNetworkInterface",  
                "ec2:DescribeSubnets",  
                "ec2:DescribeSecurityGroups"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

For information about creating a JSON policy document in the IAM console, see [Creating policies on the JSON tab](#) in the *IAM User Guide*.

Granting users access with an IAM policy

By default, users and roles don't have permission to perform [event source API operations \(p. 681\)](#). To grant access to users in your organization or account, you create or update an identity-based policy. For more information, see [Controlling access to AWS resources using policies](#) in the *IAM User Guide*.

Authentication and authorization errors

If any of the permissions required to consume data from the Kafka cluster are missing, Lambda displays one of the following error messages in the event source mapping under **LastProcessingResult**.

Error messages

- [Cluster failed to authorize Lambda \(p. 676\)](#)
- [SASL authentication failed \(p. 676\)](#)
- [Server failed to authenticate Lambda \(p. 676\)](#)
- [Lambda failed to authenticate server \(p. 677\)](#)
- [Provided certificate or private key is invalid \(p. 677\)](#)

Cluster failed to authorize Lambda

For SASL/SCRAM or mTLS, this error indicates that the provided user doesn't have all of the following required Kafka access control list (ACL) permissions:

- DescribeConfigs Cluster
- Describe Group
- Read Group
- Describe Topic
- Read Topic

When you create Kafka ACLs with the required `kafka-cluster` permissions, specify the topic and group as resources. The topic name must match the topic in the event source mapping. The group name must match the event source mapping's UUID.

After you add the required permissions to the execution role, it might take several minutes for the changes to take effect.

SASL authentication failed

For SASL/SCRAM or SASL/PLAIN, this error indicates that the provided sign-in credentials aren't valid.

Server failed to authenticate Lambda

This error indicates that the Kafka broker failed to authenticate Lambda. This can occur for any of the following reasons:

- You didn't provide a client certificate for mTLS authentication.
- You provided a client certificate, but the Kafka brokers aren't configured to use mTLS authentication.
- A client certificate isn't trusted by the Kafka brokers.

Lambda failed to authenticate server

This error indicates that Lambda failed to authenticate the Kafka broker. This can occur for any of the following reasons:

- The Kafka brokers use self-signed certificates or a private CA, but didn't provide the server root CA certificate.
- The server root CA certificate doesn't match the root CA that signed the broker's certificate.
- Hostname validation failed because the broker's certificate doesn't contain the broker's DNS name or IP address as a subject alternative name.

Provided certificate or private key is invalid

This error indicates that the Kafka consumer couldn't use the provided certificate or private key. Make sure that the certificate and key use PEM format, and that the private key encryption uses a PBES1 algorithm.

Network configuration

If you configure Amazon VPC access to your Kafka brokers, Lambda must have access to the Amazon VPC resources associated with your Kafka cluster. We recommend that you deploy AWS PrivateLink [VPC endpoints](#) for Lambda and AWS Security Token Service (AWS STS). If the broker uses authentication, also deploy a VPC endpoint for Secrets Manager.

Alternatively, ensure that the VPC associated with your Kafka cluster includes one NAT gateway per public subnet. For more information, see [Internet and service access for VPC-connected functions \(p. 228\)](#).

Configure your Amazon VPC security groups with the following rules (at minimum):

- Inbound rules – Allow all traffic on the Kafka broker port for the security groups specified for your event source. Kafka uses port 9092 by default.
- Outbound rules – Allow all traffic on port 443 for all destinations. Allow all traffic on the Kafka broker port for the security groups specified for your event source. Kafka uses port 9092 by default.
- If you are using VPC endpoints instead of a NAT gateway, the security groups associated with the VPC endpoints must allow all inbound traffic on port 443 from the event source's security groups.

For more information about configuring the network, see [Setting up AWS Lambda with an Apache Kafka cluster within a VPC](#) on the AWS Compute Blog.

Adding a Kafka cluster as an event source

To create an [event source mapping \(p. 131\)](#), add your Kafka cluster as a Lambda function [trigger \(p. 9\)](#) using the Lambda console, an [AWS SDK](#), or the [AWS Command Line Interface \(AWS CLI\)](#).

This section describes how to create an event source mapping using the Lambda console and the AWS CLI.

Note

When you update, disable, or delete an event source mapping for self-managed Apache Kafka, it can take up to 15 minutes for your changes to take effect. Before this period has elapsed, your event source mapping may continue to process events and invoke your function using your previous settings. This is true even when the status of the event source mapping displayed in the console indicates that your changes have been applied.

Prerequisites

- A self-managed Apache Kafka cluster. Lambda supports Apache Kafka version 0.10.0.0 and later.
- An [execution role \(p. 816\)](#) with permission to access the AWS resources that your self-managed Kafka cluster uses.

Customizable consumer group ID

When setting up Kafka as an event source, you can specify a consumer group ID. This consumer group ID is an existing identifier for the Kafka consumer group that you want your Lambda function to join. You can use this feature to seamlessly migrate any ongoing Kafka record processing setups from other consumers to Lambda.

If you specify a consumer group ID and there are other active pollers within that consumer group, Kafka distributes messages across all consumers. In other words, Lambda doesn't receive all message for the Kafka topic. If you want Lambda to handle all messages in the topic, turn off any other pollers in that consumer group.

Additionally, if you specify a consumer group ID, and Kafka finds a valid existing consumer group with the same ID, Lambda ignores the `StartingPosition` parameter for your event source mapping. Instead, Lambda begins processing records according to the committed offset of the consumer group. If you specify a consumer group ID, and Kafka cannot find an existing consumer group, then Lambda configures your event source with the specified `StartingPosition`.

The consumer group ID that you specify must be unique among all your Kafka event sources. After creating a Kafka event source mapping with the consumer group ID specified, you cannot update this value.

Adding a self-managed Kafka cluster (console)

Follow these steps to add your self-managed Apache Kafka cluster and a Kafka topic as a trigger for your Lambda function.

To add an Apache Kafka trigger to your Lambda function (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of your Lambda function.
3. Under **Function overview**, choose **Add trigger**.
4. Under **Trigger configuration**, do the following:
 - a. Choose the **Apache Kafka** trigger type.
 - b. For **Bootstrap servers**, enter the host and port pair address of a Kafka broker in your cluster, and then choose **Add**. Repeat for each Kafka broker in the cluster.
 - c. For **Topic name**, enter the name of the Kafka topic used to store records in the cluster.
 - d. (Optional) For **Batch size**, enter the maximum number of records to receive in a single batch.
 - e. For **Batch window**, enter the maximum amount of seconds that Lambda spends gathering records before invoking the function.
 - f. (Optional) For **Consumer group ID**, enter the ID of a Kafka consumer group to join.
 - g. (Optional) For **Starting position**, choose **Latest** to start reading the stream from the latest record. Or, choose **Trim horizon** to start at the earliest available record.
 - h. (Optional) For **VPC**, choose the Amazon VPC for your Kafka cluster. Then, choose the **VPC subnets** and **VPC security groups**.

This setting is required if only users within your VPC access your brokers.

- i. (Optional) For **Authentication**, choose **Add**, and then do the following:
 - i. Choose the access or authentication protocol of the Kafka brokers in your cluster.
 - If your Kafka broker uses SASL/PLAIN authentication, choose **BASIC_AUTH**.
 - If your broker uses SASL/SCRAM authentication, choose one of the **SASL_SCRAM** protocols.
 - If you're configuring mTLS authentication, choose the **CLIENT_CERTIFICATE_TLS_AUTH** protocol.
 - ii. For SASL/SCRAM or mTLS authentication, choose the Secrets Manager secret key that contains the credentials for your Kafka cluster.
 - j. (Optional) For **Encryption**, choose the Secrets Manager secret containing the root CA certificate that your Kafka brokers use for TLS encryption, if your Kafka brokers use certificates signed by a private CA.
- This setting applies to TLS encryption for SASL/SCRAM or SASL/PLAIN, and to mTLS authentication.
- k. To create the trigger in a disabled state for testing (recommended), clear **Enable trigger**. Or, to enable the trigger immediately, select **Enable trigger**.
5. To create the trigger, choose **Add**.

Adding a self-managed Kafka cluster (AWS CLI)

Use the following example AWS CLI commands to create and view a self-managed Apache Kafka trigger for your Lambda function.

Using SASL/SCRAM

If Kafka users access your Kafka brokers over the internet, specify the Secrets Manager secret that you created for SASL/SCRAM authentication. The following example uses the [create-event-source-mapping](#) AWS CLI command to map a Lambda function named my-kafka-function to a Kafka topic named AWSKafkaTopic.

```
aws lambda create-event-source-mapping --topics AWSKafkaTopic
  --source-access-configuration
    Type=SASL_SCRAM_512_AUTH,URI=arn:aws:secretsmanager:us-
east-1:01234567890:secret:MyBrokerSecretName
    --function-name arn:aws:lambda:us-east-1:01234567890:function:my-kafka-function
    --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
      ["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

For more information, see the [CreateEventSourceMapping \(p. 1153\)](#) API reference documentation.

Using a VPC

If only Kafka users within your VPC access your Kafka brokers, you must specify your VPC, subnets, and VPC security group. The following example uses the [create-event-source-mapping](#) AWS CLI command to map a Lambda function named my-kafka-function to a Kafka topic named 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-1:01234567890:function:my-kafka-function  
--self-managed-event-source '{"Endpoints": {"KAFKA_BOOTSTRAP_SERVERS":  
["abc3.xyz.com:9092',  
 "abc2.xyz.com:9092"]}}'
```

For more information, see the [CreateEventSourceMapping \(p. 1153\)](#) API reference documentation.

Viewing the status using the AWS CLI

The following example uses the [get-event-source-mapping](#) AWS CLI command to describe the status of the event source mapping that you created.

```
aws lambda get-event-source-mapping  
--uuid dh38738e-992b-43a-1077-3478934hjkfd7
```

Using a Kafka cluster as an event source

When you add your Apache Kafka cluster as a trigger for your Lambda function, the cluster is used as an [event source \(p. 131\)](#).

Lambda reads event data from the Kafka topics that you specify as Topics in a [CreateEventSourceMapping \(p. 1153\)](#) request, based on the StartingPosition that you specify. After successful processing, your Kafka topic is committed to your Kafka cluster.

If you specify the StartingPosition as LATEST, Lambda starts reading from the latest message in each partition belonging to the topic. Because there can be some delay after trigger configuration before Lambda starts reading the messages, Lambda doesn't read any messages produced during this window.

Lambda processes records from one or more Kafka topic partitions that you specify and sends a JSON payload to your function. When more records are available, Lambda continues processing records in batches, based on the BatchSize value that you specify in a [CreateEventSourceMapping \(p. 1153\)](#) request, until your function catches up with the topic.

If your function returns an error for any of the messages in a batch, Lambda retries the whole batch of messages until processing succeeds or the messages expire.

Note

While Lambda functions typically have a maximum timeout limit of 15 minutes, event source mappings for Amazon MSK, self-managed Apache Kafka, and Amazon MQ for ActiveMQ and RabbitMQ only support functions with maximum timeout limits of 14 minutes. This constraint ensures that the event source mapping can properly handle function errors and retries.

Auto scaling of the Kafka event source

When you initially create an an Apache Kafka [event source \(p. 131\)](#), Lambda allocates one consumer to process all partitions in the Kafka topic. Each consumer has multiple processors running in parallel to handle increased workloads. Additionally, Lambda automatically scales up or down the number of consumers, based on workload. To preserve message ordering in each partition, the maximum number of consumers is one consumer per partition in the topic.

In one-minute intervals, Lambda evaluates the consumer offset lag of all the partitions in the topic. If the lag is too high, the partition is receiving messages faster than Lambda can process them. If necessary, Lambda adds or removes consumers from the topic. The scaling process of adding or removing consumers occurs within three minutes of evaluation.

If your target Lambda function is overloaded, Lambda reduces the number of consumers. This action reduces the workload on the function by reducing the number of messages that consumers can retrieve and send to the function.

To monitor the throughput of your Kafka topic, you can view the Apache Kafka consumer metrics, such as `consumer_lag` and `consumer_offset`. To check how many function invocations occur in parallel, you can also monitor the [concurrency metrics \(p. 872\)](#) for your function.

Event source API operations

When you add your Kafka cluster as an [event source \(p. 131\)](#) for your Lambda function using the Lambda console, an AWS SDK, or the AWS CLI, Lambda uses APIs to process your request.

To manage an event source with the [AWS Command Line Interface \(AWS CLI\)](#) or an [AWS SDK](#), you can use the following API operations:

- [CreateEventSourceMapping \(p. 1153\)](#)
- [ListEventSourceMappings \(p. 1279\)](#)
- [GetEventSourceMapping \(p. 1214\)](#)
- [UpdateEventSourceMapping \(p. 1356\)](#)
- [DeleteEventSourceMapping \(p. 1186\)](#)

Event source mapping errors

When you add your Apache Kafka cluster as an [event source \(p. 131\)](#) for your Lambda function, if your function encounters an error, your Kafka consumer stops processing records. Consumers of a topic partition are those that subscribe to, read, and process your records. Your other Kafka consumers can continue processing records, provided they don't encounter the same error.

To determine the cause of a stopped consumer, check the `StateTransitionReason` field in the response of `EventSourceMapping`. The following list describes the event source errors that you can receive:

ESM_CONFIG_NOT_VALID

The event source mapping configuration isn't valid.

EVENT_SOURCE_AUTHN_ERROR

Lambda couldn't authenticate the event source.

EVENT_SOURCE_AUTHZ_ERROR

Lambda doesn't have the required permissions to access the event source.

FUNCTION_CONFIG_NOT_VALID

The function configuration isn't valid.

Note

If your Lambda event records exceed the allowed size limit of 6 MB, they can go unprocessed.

Amazon CloudWatch metrics

Lambda emits the `OffsetLag` metric while your function processes records. The value of this metric is the difference in offset between the last record written to the Kafka event source topic and the last

record that your function's consumer group processed. You can use `OffsetLag` to estimate the latency between when a record is added and when your consumer group processes it.

An increasing trend in `OffsetLag` can indicate issues with pollers in your function's consumer group. For more information, see [Working with Lambda function metrics \(p. 870\)](#).

Self-managed Apache Kafka configuration parameters

All Lambda event source types share the same [CreateEventSourceMapping \(p. 1153\)](#) and [UpdateEventSourceMapping \(p. 1356\)](#) API operations. However, only some of the parameters apply to Apache Kafka.

Event source parameters that apply to self-managed Apache Kafka

Parameter	Required	Default	Notes
BatchSize	N	100	Maximum: 10,000
Enabled	N	Enabled	
FunctionName	Y		
FilterCriteria	N		Lambda event filtering (p. 136)
MaximumBatchingWindowInSeconds	N	500 ms	Batching behavior (p. 132)
SelfManagedEventSource	Y		List of Kafka Brokers. Can set only on Create
SelfManagedKafkaEventSourceConfig	N	Contains the <code>ConsumerGroupId</code> field which defaults to a unique value.	Can set only on Create
SourceAccessConfiguration	N	No credentials	VPC information or authentication credentials for the cluster For <code>SASL_PLAIN</code> , set to <code>BASIC_AUTH</code>
StartingPosition	Y		TRIM_HORIZON or LATEST Can set only on Create
Topics	Y		Topic name Can set only on Create

Using AWS Lambda with Amazon Kinesis Data Firehose

Amazon Kinesis Data Firehose captures, transforms, and loads streaming data into downstream services such as Kinesis Data Analytics or Amazon S3. You can write Lambda functions to request additional, customized processing of the data before it is sent downstream.

Example Amazon Kinesis Data Firehose message event

```
{  
    "invocationId": "invoked123",  
    "deliveryStreamArn": "aws:lambda:events",  
    "region": "us-west-2",  
    "records": [  
        {  
            "data": "SGVsbG8gV29ybGQ=",  
            "recordId": "record1",  
            "approximateArrivalTimestamp": 151077216000,  
            "kinesisRecordMetadata": {  
                "shardId": "shardId-000000000000",  
                "partitionKey": "4d1ad2b9-24f8-4b9d-a088-76e9947c317a",  
                "approximateArrivalTimestamp": "2012-04-23T18:25:43.511Z",  
                "sequenceNumber": "49546986683135544286507457936321625675700192471156785154",  
                "subsequenceNumber": ""  
            }  
        },  
        {  
            "data": "SGVsbG8gV29ybGQ=",  
            "recordId": "record2",  
            "approximateArrivalTimestamp": 151077216000,  
            "kinesisRecordMetadata": {  
                "shardId": "shardId-000000000001",  
                "partitionKey": "4d1ad2b9-24f8-4b9d-a088-76e9947c318a",  
                "approximateArrivalTimestamp": "2012-04-23T19:25:43.511Z",  
                "sequenceNumber": "49546986683135544286507457936321625675700192471156785155",  
                "subsequenceNumber": ""  
            }  
        }  
    ]  
}
```

For more information, see [Amazon Kinesis Data Firehose data transformation](#) in the Kinesis Data Firehose Developer Guide.

Using AWS Lambda with Amazon Kinesis

Note

If you want to send data to a target other than a Lambda function or enrich the data before sending it, see [Amazon EventBridge Pipes](#).

You can use an AWS Lambda function to process records in an [Amazon Kinesis data stream](#).

A Kinesis data stream is a set of [shards](#). Each shard contains a sequence of data records. A **consumer** is an application that processes the data from a Kinesis data stream. You can map a Lambda function to a shared-throughput consumer (standard iterator), or to a dedicated-throughput consumer with [enhanced fan-out](#).

For standard iterators, Lambda polls each shard in your Kinesis stream for records using HTTP protocol. The event source mapping shares read throughput with other consumers of the shard.

To minimize latency and maximize read throughput, you can create a data stream consumer with enhanced fan-out. Stream consumers get a dedicated connection to each shard that doesn't impact other applications reading from the stream. The dedicated throughput can help if you have many applications reading the same data, or if you're reprocessing a stream with large records. Kinesis pushes records to Lambda over HTTP/2.

For details about Kinesis data streams, see [Reading Data from Amazon Kinesis Data Streams](#).

Sections

- [Example event \(p. 684\)](#)
- [Polling and batching streams \(p. 685\)](#)
- [Configuring your data stream and function \(p. 686\)](#)
- [Execution role permissions \(p. 686\)](#)
- [Configuring a stream as an event source \(p. 687\)](#)
- [Filtering Kinesis events \(p. 688\)](#)
- [Event source mapping API \(p. 688\)](#)
- [Error handling \(p. 690\)](#)
- [Amazon CloudWatch metrics \(p. 691\)](#)
- [Time windows \(p. 691\)](#)
- [Reporting batch item failures \(p. 693\)](#)
- [Amazon Kinesis configuration parameters \(p. 695\)](#)
- [Tutorial: Using AWS Lambda with Amazon Kinesis \(p. 696\)](#)
- [Sample function code \(p. 701\)](#)
- [AWS SAM template for a Kinesis application \(p. 704\)](#)

Example event

Example

```
{  
    "Records": [  
        {  
            "kinesis": {  
                "kinesisSchemaVersion": "1.0",  
                "partitionKey": "1",  
                "sequenceNumber":  
                    "49590338271490256608559692538361571095921575989136588898",  
                "approximateArrivalTimestamp": 1571095921575989136588898  
            }  
        }  
    ]  
}
```

```

        "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": "VGhpccyBpcyBvbmx5IGEgdGVzdC4=",
        "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"
}
]
}

```

Polling and batching streams

Lambda reads records from the data stream and invokes your function [synchronously \(p. 120\)](#) with an event that contains stream records. Lambda reads records in batches and invokes your function to process records from the batch. Each batch contains records from a single shard/data stream.

By default, Lambda invokes your function as soon as records are available. If the batch that Lambda reads from the event source has only one record in it, Lambda sends only one record to the function. To avoid invoking the function with a small number of records, you can tell the event source to buffer records for up to 5 minutes by configuring a *batching window*. Before invoking the function, Lambda continues to read records from the event source until it has gathered a full batch, the batching window expires, or the batch reaches the payload limit of 6 MB. For more information, see [Batching behavior \(p. 132\)](#).

If your function returns an error, Lambda retries the batch until processing succeeds or the data expires. To avoid stalled shards, you can configure the event source mapping to retry with a smaller batch size, limit the number of retries, or discard records that are too old. To retain discarded events, you can configure the event source mapping to send details about failed batches to a standard SQS queue or standard SNS topic.

You can also increase concurrency by processing multiple batches from each shard in parallel. Lambda can process up to 10 batches in each shard simultaneously. If you increase the number of concurrent batches per shard, Lambda still ensures in-order processing at the partition key level.

Configure the `ParallelizationFactor` setting to process one shard of a Kinesis or DynamoDB data stream with more than one Lambda invocation simultaneously. You can specify the number of concurrent batches that Lambda polls from a shard via a parallelization factor from 1 (default) to 10. For example, when you set `ParallelizationFactor` to 2, you can have 200 concurrent

Lambda invocations at maximum to process 100 Kinesis data shards. This helps scale up the processing throughput when the data volume is volatile and the IteratorAge is high. Note that parallelization factor will not work if you are using Kinesis aggregation. For more information, see [New AWS Lambda scaling controls for Kinesis and DynamoDB event sources](#). Also, see the [Serverless Data Processing on AWS](#) workshop for complete tutorials.

Configuring your data stream and function

Your Lambda function is a consumer application for your data stream. It processes one batch of records at a time from each shard. You can map a Lambda function to a data stream (standard iterator), or to a consumer of a stream ([enhanced fan-out](#)).

For standard iterators, Lambda polls each shard in your Kinesis stream for records at a base rate of once per second. When more records are available, Lambda keeps processing batches until the function catches up with the stream. The event source mapping shares read throughput with other consumers of the shard.

To minimize latency and maximize read throughput, create a data stream consumer with enhanced fan-out. Enhanced fan-out consumers get a dedicated connection to each shard that doesn't impact other applications reading from the stream. Stream consumers use HTTP/2 to reduce latency by pushing records to Lambda over a long-lived connection and by compressing request headers. You can create a stream consumer with the Kinesis [RegisterStreamConsumer](#) API.

```
aws kinesis register-stream-consumer --consumer-name con1 \
--stream-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream
```

You should see the following output:

```
{
  "Consumer": {
    "ConsumerName": "con1",
    "ConsumerARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream/
consumer/con1:1540591608",
    "ConsumerStatus": "CREATING",
    "ConsumerCreationTimestamp": 1540591608.0
  }
}
```

To increase the speed at which your function processes records, add shards to your data stream. Lambda processes records in each shard in order. It stops processing additional records in a shard if your function returns an error. With more shards, there are more batches being processed at once, which lowers the impact of errors on concurrency.

If your function can't scale up to handle the total number of concurrent batches, [request a quota increase \(p. 1131\)](#) or [reserve concurrency \(p. 210\)](#) for your function.

Execution role permissions

Lambda needs the following permissions to manage resources that are related to your Kinesis data stream. Add them to your function's [execution role \(p. 816\)](#).

- [kinesis:DescribeStream](#)
- [kinesis:DescribeStreamSummary](#)
- [kinesis:GetRecords](#)
- [kinesis:GetShardIterator](#)

- [kinesis>ListShards](#)
- [kinesis>ListStreams](#)
- [kinesis>SubscribeToShard](#)

The AWSLambdaKinesisExecutionRole managed policy includes these permissions. For more information, see [Lambda execution role \(p. 816\)](#).

To send records of failed batches to a standard SQS queue or standard SNS topic, your function needs additional permissions. Each destination service requires a different permission, as follows:

- **Amazon SQS** – [sns:SendMessage](#)
- **Amazon SNS** – [sns:Publish](#)

Configuring a stream as an event source

Create an event source mapping to tell Lambda to send records from your data stream to a Lambda function. You can create multiple event source mappings to process the same data with multiple Lambda functions, or to process items from multiple data streams with a single function. When processing items from multiple data streams, each batch will only contain records from a single shard/stream.

To configure your function to read from Kinesis in the Lambda console, create a **Kinesis trigger**.

To create a trigger

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of a function.
3. Under **Function overview**, choose **Add trigger**.
4. Choose a trigger type.
5. Configure the required options, and then choose **Add**.

Lambda supports the following options for Kinesis event sources.

Event source options

- **Kinesis stream** – The Kinesis stream to read records from.
- **Consumer** (optional) – Use a stream consumer to read from the stream over a dedicated connection.
- **Batch size** – The number of records to send to the function in each batch, up to 10,000. Lambda passes all of the records in the batch to the function in a single call, as long as the total size of the events doesn't exceed the [payload limit \(p. 1131\)](#) for synchronous invocation (6 MB).
- **Batch window** – Specify the maximum amount of time to gather records before invoking the function, in seconds.
- **Starting position** – Process only new records, all existing records, or records created after a certain date.
 - **Latest** – Process new records that are added to the stream.
 - **Trim horizon** – Process all records in the stream.
 - **At timestamp** – Process records starting from a specific time.

After processing any existing records, the function is caught up and continues to process new records.

- **On-failure destination** – A standard SQS queue or standard SNS topic for records that can't be processed. When Lambda discards a batch of records that's too old or has exhausted all retries, Lambda sends details about the batch to the queue or topic.

- **Retry attempts** – The maximum number of times that Lambda retries when the function returns an error. This doesn't apply to service errors or throttles where the batch didn't reach the function.
- **Maximum age of record** – The maximum age of a record that Lambda sends to your function.
- **Split batch on error** – When the function returns an error, split the batch into two before retrying. Your original batch size setting remains unchanged.
- **Concurrent batches per shard** – Concurrently process multiple batches from the same shard.
- **Enabled** – Set to true to enable the event source mapping. Set to false to stop processing records. Lambda keeps track of the last record processed and resumes processing from that point when it's reenabled.

Note

Kinesis charges for each shard and, for enhanced fan-out, data read from the stream. For pricing details, see [Amazon Kinesis pricing](#).

To manage the event source configuration later, choose the trigger in the designer.

Filtering Kinesis events

When you configure Kinesis as an event source for Lambda, you can use [event filtering](#) to control which records from your stream Lambda sends to your function for processing. To learn more about using Lambda event filtering with Kinesis, see [Filtering with Kinesis](#).

Event source mapping API

To manage an event source with the [AWS Command Line Interface \(AWS CLI\)](#) or an [AWS SDK](#), you can use the following API operations:

- [CreateEventSourceMapping \(p. 1153\)](#)
- [ListEventSourceMappings \(p. 1279\)](#)
- [GetEventSourceMapping \(p. 1214\)](#)
- [UpdateEventSourceMapping \(p. 1356\)](#)
- [DeleteEventSourceMapping \(p. 1186\)](#)

To create the event source mapping with the AWS CLI, use the `create-event-source-mapping` command. The following example uses the AWS CLI to map a function named `my-function` to a Kinesis data stream. The data stream is specified by an Amazon Resource Name (ARN), with a batch size of 500, starting from the timestamp in Unix time.

```
aws lambda create-event-source-mapping --function-name my-function \
--batch-size 500 --starting-position AT_TIMESTAMP --starting-position-timestamp 1541139109 \
\
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream
```

You should see the following output:

```
{  
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",  
    "BatchSize": 500,  
    "MaximumBatchingWindowInSeconds": 0,  
    "ParallelizationFactor": 1,  
    "EventSourceArn": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream",  
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",  
    "LastModified": 1541139209.351,
```

```

    "LastProcessingResult": "No records processed",
    "State": "Creating",
    "StateTransitionReason": "User action",
    "DestinationConfig": {},
    "MaximumRecordAgeInSeconds": 604800,
    "BisectBatchOnFunctionError": false,
    "MaximumRetryAttempts": 10000
}

```

To use a consumer, specify the consumer's ARN instead of the stream's ARN.

Configure additional options to customize how batches are processed and to specify when to discard records that can't be processed. The following example updates an event source mapping to send a failure record to a standard SQS queue after two retry attempts, or if the records are more than an hour old.

```
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:sqs:us-east-2:123456789012:d1q"}}'
```

You should see this output:

```
{
    "UUID": "f89f8514-cdd9-4602-9e1f-01a5b77d449b",
    "BatchSize": 100,
    "MaximumBatchingWindowInSeconds": 0,
    "ParallelizationFactor": 1,
    "EventSourceArn": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "LastModified": 1573243620.0,
    "LastProcessingResult": "PROBLEM: Function call failed",
    "State": "Updating",
    "StateTransitionReason": "User action",
    "DestinationConfig": {},
    "MaximumRecordAgeInSeconds": 604800,
    "BisectBatchOnFunctionError": false,
    "MaximumRetryAttempts": 10000
}
```

Updated settings are applied asynchronously and aren't reflected in the output until the process completes. Use the `get-event-source-mapping` command to view the current status.

```
aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b
```

You should see this output:

```
{
    "UUID": "f89f8514-cdd9-4602-9e1f-01a5b77d449b",
    "BatchSize": 100,
    "MaximumBatchingWindowInSeconds": 0,
    "ParallelizationFactor": 1,
    "EventSourceArn": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "LastModified": 1573244760.0,
    "LastProcessingResult": "PROBLEM: Function call failed",
    "State": "Enabled",
    "StateTransitionReason": "User action",
    "DestinationConfig": {
        "OnFailure": {

```

```
        "Destination": "arn:aws:sqs:us-east-2:123456789012:d1q"
    },
    "MaximumRecordAgeInSeconds": 3600,
    "BisectBatchOnFunctionError": false,
    "MaximumRetryAttempts": 2
}
```

To process multiple batches concurrently, use the `--parallelization-factor` option.

```
aws lambda update-event-source-mapping --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284 \
--parallelization-factor 5
```

Error handling

The event source mapping that reads records from your Kinesis stream, invokes your function synchronously, and retries on errors. If Lambda throttles the function or returns an error without invoking the function, Lambda retries until the records expire or exceed the maximum age that you configure on the event source mapping.

If the function receives the records but returns an error, Lambda retries until the records in the batch expire, exceed the maximum age, or reach the configured retry quota. For function errors, you can also configure the event source mapping to split a failed batch into two batches. Retrying with smaller batches isolates bad records and works around timeout issues. Splitting a batch does not count towards the retry quota.

If the error handling measures fail, Lambda discards the records and continues processing batches from the stream. With the default settings, this means that a bad record can block processing on the affected shard for up to one week. To avoid this, configure your function's event source mapping with a reasonable number of retries and a maximum record age that fits your use case.

To retain a record of discarded batches, configure a failed-event destination. Lambda sends a document to the destination queue or topic with details about the batch.

To configure a destination for failed-event records

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Under **Function overview**, choose **Add destination**.
4. For **Source**, choose **Stream invocation**.
5. For **Stream**, choose a stream that is mapped to the function.
6. For **Destination type**, choose the type of resource that receives the invocation record.
7. For **Destination**, choose a resource.
8. Choose **Save**.

The following example shows an invocation record for a Kinesis stream.

Example invocation record

```
{
  "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"
    }
}
```

You can use this information to retrieve the affected records from the stream for troubleshooting. The actual records aren't included, so you must process this record and retrieve them from the stream before they expire and are lost.

Amazon CloudWatch metrics

Lambda emits the `IteratorAge` metric when your function finishes processing a batch of records. The metric indicates how old the last record in the batch was when processing finished. If your function is processing new events, you can use the iterator age to estimate the latency between when a record is added and when the function processes it.

An increasing trend in iterator age can indicate issues with your function. For more information, see [Working with Lambda function metrics \(p. 870\)](#).

Time windows

Lambda functions can run continuous stream processing applications. A stream represents unbounded data that flows continuously through your application. To analyze information from this continuously updating input, you can bound the included records using a window defined in terms of time.

Tumbling windows are distinct time windows that open and close at regular intervals. By default, Lambda invocations are stateless—you cannot use them for processing data across multiple continuous invocations without an external database. However, with tumbling windows, you can maintain your state across invocations. This state contains the aggregate result of the messages previously processed for the current window. Your state can be a maximum of 1 MB per shard. If it exceeds that size, Lambda terminates the window early.

Each record in a stream belongs to a specific window. Lambda will process each record at least once, but doesn't guarantee that each record will be processed only once. In rare cases, such as error handling, some records might be processed more than once. Records are always processed in order the first time. If records are processed more than once, they might be processed out of order.

Aggregation and processing

Your user managed function is invoked both for aggregation and for processing the final results of that aggregation. Lambda aggregates all records received in the window. You can receive these records in multiple batches, each as a separate invocation. Each invocation receives a state. Thus, when using tumbling windows, your Lambda function response must contain a `state` property. If the response does

not contain a state property, Lambda considers this a failed invocation. To satisfy this condition, your function can return a TimeWindowEventResponse object, which has the following JSON shape:

Example TimeWindowEventResponse values

```
{  
    "state": {  
        "1": 282,  
        "2": 715  
    },  
    "batchItemFailures": []  
}
```

Note

For Java functions, we recommend using a Map<String, String> to represent the state.

At the end of the window, the flag isFinalInvokeForWindow is set to true to indicate that this is the final state and that it's ready for processing. After processing, the window completes and your final invocation completes, and then the state is dropped.

At the end of your window, Lambda uses final processing for actions on the aggregation results. Your final processing is synchronously invoked. After successful invocation, your function checkpoints the sequence number and stream processing continues. If invocation is unsuccessful, your Lambda function suspends further processing until a successful invocation.

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  
        }  
    ]  
}
```

Configuration

You can configure tumbling windows when you create or update an [event source mapping \(p. 131\)](#). To configure a tumbling window, specify the window in seconds. The following example AWS Command Line Interface (AWS CLI) command creates a streaming event source mapping that has a tumbling window of 120 seconds. The Lambda function defined for aggregation and processing is named `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 "arn:aws:lambda:us-east-1:123456789018:function:tumbling-window-example-function" --region us-east-1 --starting-position TRIM_HORIZON --tumbling-window-in-seconds 120
```

Lambda determines tumbling window boundaries based on the time when records were inserted into the stream. All records have an approximate timestamp available that Lambda uses in boundary determinations.

Tumbling window aggregations do not support resharding. When the shard ends, Lambda considers the window closed, and the child shards start their own window in a fresh state.

Tumbling windows fully support the existing retry policies `maxRetryAttempts` and `maxRecordAge`.

Example Handler.py – Aggregation and processing

The following Python function demonstrates how to aggregate and then process your final state:

```
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}
```

Reporting batch item failures

When consuming and processing streaming data from an event source, by default Lambda checkpoints to the highest sequence number of a batch only when the batch is a complete success. Lambda treats all other results as a complete failure and retries processing the batch up to the retry limit. To allow for partial successes while processing batches from a stream, turn on `ReportBatchItemFailures`. Allowing partial successes can help to reduce the number of retries on a record, though it doesn't entirely prevent the possibility of retries in a successful record.

To turn on `ReportBatchItemFailures`, include the enum value `ReportBatchItemFailures` in the `FunctionResponseTypes` list. This list indicates which response types are enabled for your function. You can configure this list when you create or update an [event source mapping \(p. 131\)](#).

Report syntax

When configuring reporting on batch item failures, the `StreamsEventResponse` class is returned with a list of batch item failures. You can use a `StreamsEventResponse` object to return the sequence number of the first failed record in the batch. You can also create your own custom class using the correct response syntax. The following JSON structure shows the required response syntax:

```
{  
    "batchItemFailures": [  
        {  
            "itemIdentifier": "<id>"  
        }  
    ]  
}
```

Note

If the `batchItemFailures` array contains multiple items, Lambda uses the record with the lowest sequence number as the checkpoint. Lambda then retries all records starting from that checkpoint.

Success and failure conditions

Lambda treats a batch as a complete success if you return any of the following:

- An empty `batchItemFailure` list
- A null `batchItemFailure` list
- An empty `EventResponse`
- A null `EventResponse`

Lambda treats a batch as a complete failure if you return any of the following:

- An empty string `itemIdentifier`
- A null `itemIdentifier`
- An `itemIdentifier` with a bad key name

Lambda retries failures based on your retry strategy.

Bisecting a batch

If your invocation fails and `BisectBatchOnFunctionError` is turned on, the batch is bisected regardless of your `ReportBatchItemFailures` setting.

When a partial batch success response is received and both `BisectBatchOnFunctionError` and `ReportBatchItemFailures` are turned on, the batch is bisected at the returned sequence number and Lambda retries only the remaining records.

Java

Example Handler.java – return new StreamsEventResponse()

```
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);
    }
}

```

Python

Example Handler.py – return batchItemFailures[]

```

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": []}

```

Amazon Kinesis configuration parameters

All Lambda event source types share the same [CreateEventSourceMapping \(p. 1153\)](#) and [UpdateEventSourceMapping \(p. 1356\)](#) API operations. However, only some of the parameters apply to Kinesis.

Event source parameters that apply to Kinesis

Parameter	Required	Default	Notes
BatchSize	N	100	Maximum: 10,000
BisectBatchOnFunctionError	N	false	
DestinationConfig	N		standard Amazon SQS queue or standard Amazon SNS topic destination for discarded records
Enabled	N	true	
EventSourceArn	Y		ARN of the data stream or a stream consumer
FunctionName	Y		
MaximumBatchingWindowInSeconds	N	0	
MaximumRecordAgeInSeconds	N	-1	-1 means infinite: Lambda doesn't discard records Minimum: -1 Maximum: 604,800
MaximumRetryAttempts	N	-1	-1 means infinite: failed records are retried until the record expires Minimum: -1 Maximum: 10,000
ParallelizationFactor	N	1	Maximum: 10
StartingPosition	Y		AT_TIMESTAMP, TRIM_HORIZON, or LATEST
StartingPositionTimestamp	N		Only valid if StartingPosition is set to AT_TIMESTAMP. The time from which to start reading, in Unix time seconds
TumblingWindowInSeconds	N		Minimum: 0 Maximum: 900

Tutorial: Using AWS Lambda with Amazon Kinesis

In this tutorial, you create a Lambda function to consume events from a Kinesis stream.

1. Custom app writes records to the stream.
2. AWS Lambda polls the stream and, when it detects new records in the stream, invokes your Lambda function.
3. AWS Lambda runs the Lambda function by assuming the execution role you specified at the time you created the Lambda function.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console \(p. 4\)](#) to create your first Lambda function.

To complete the following steps, you need the [AWS Command Line Interface \(AWS CLI\) version 2](#). Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.0.57 Python/3.7.4 Darwin/19.6.0 exe/x86_64
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#). Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

Create the execution role

Create the [execution role \(p. 816\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – AWS Lambda.**
 - **Permissions – AWSLambdaKinesisExecutionRole.**
 - **Role name – lambda-kinesis-role.**

The **AWSLambdaKinesisExecutionRole** policy has the permissions that the function needs to read items from Kinesis and write logs to CloudWatch Logs.

Create the function

The following example code receives a Kinesis event input and processes the messages that it contains. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Note

For sample code in other languages, see [Sample function code \(p. 701\)](#).

Example index.js

```
console.log('Loading function');

exports.handler = function(event, context) {
    //console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        // Kinesis data is base64 encoded so decode here
        var payload = Buffer.from(record.kinesis.data, 'base64').toString('ascii');
        console.log('Decoded payload:', payload);
    });
};
```

To create the function

1. Copy the sample code into a file named `index.js`.
2. Create a deployment package.

```
zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command.

```
aws lambda create-function --function-name ProcessKinesisRecords \
--zip-file file://function.zip --handler index.handler --runtime nodejs12.x \
--role arn:aws:iam::123456789012:role/lambda-kinesis-role
```

Test the Lambda function

Invoke your Lambda function manually using the `invoke` AWS Lambda CLI command and a sample Kinesis event.

To test the Lambda function

1. Copy the following JSON into a file and save it as `input.txt`.

```
{
  "Records": [
    {
      "kinesis": {
        "kinesisSchemaVersion": "1.0",
        "partitionKey": "1",
        "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
        "data": "SGVsbG8sIHRoaXMgYSB0ZXN0Lg==",
        "approximateArrivalTimestamp": 1545084650.987
      },
      "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-2",
      "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
    stream"
  ]
}
```

```
        ]
    }
```

2. Use the `invoke` command to send the event to the function.

```
aws lambda invoke --function-name ProcessKinesisRecords --payload file://input.txt
out.txt
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

The response is saved to `out.txt`.

Create a Kinesis stream

Use the `create-stream` command to create a stream.

```
aws kinesis create-stream --stream-name lambda-stream --shard-count 1
```

Run the following `describe-stream` command to get the stream ARN.

```
aws kinesis describe-stream --stream-name lambda-stream
```

You should see the following output:

```
{
  "StreamDescription": {
    "Shards": [
      {
        "ShardId": "shardId-000000000000",
        "HashKeyRange": {
          "StartingHashKey": "0",
          "EndingHashKey": "340282366920746074317682119384634633455"
        },
        "SequenceNumberRange": {
          "StartingSequenceNumber":
            "49591073947768692513481539594623130411957558361251844610"
        }
      }
    ],
    "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/lambda-stream",
    "StreamName": "lambda-stream",
    "StreamStatus": "ACTIVE",
    "RetentionPeriodHours": 24,
    "EnhancedMonitoring": [
      {
        "ShardLevelMetrics": []
      }
    ],
    "EncryptionType": "NONE",
    "KeyId": null,
    "StreamCreationTimestamp": 1544828156.0
  }
}
```

You use the stream ARN in the next step to associate the stream with your Lambda function.

Add an event source in AWS Lambda

Run the following AWS CLI add-event-source command.

```
aws lambda create-event-source-mapping --function-name ProcessKinesisRecords \
--event-source arn:aws:kinesis:us-west-2:123456789012:stream/lambda-stream \
--batch-size 100 --starting-position LATEST
```

Note the mapping ID for later use. You can get a list of event source mappings by running the list-event-source-mappings command.

```
aws lambda list-event-source-mappings --function-name ProcessKinesisRecords \
--event-source arn:aws:kinesis:us-west-2:123456789012:stream/lambda-stream
```

In the response, you can verify the status value is enabled. Event source mappings can be disabled to pause polling temporarily without losing any records.

Test the setup

To test the event source mapping, add event records to your Kinesis stream. The --data value is a string that the CLI encodes to base64 prior to sending it to Kinesis. You can run the same command more than once to add multiple records to the stream.

```
aws kinesis put-record --stream-name lambda-stream --partition-key 1 \
--data "Hello, this is a test."
```

Lambda uses the execution role to read records from the stream. Then it invokes your Lambda function, passing in batches of records. The function decodes data from each record and logs it, sending the output to CloudWatch Logs. View the logs in the [CloudWatch console](#).

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

To delete the Kinesis stream

1. Sign in to the AWS Management Console and open the Kinesis console at <https://console.aws.amazon.com/kinesis>.

2. Select the stream you created.
3. Choose **Actions, Delete**.
4. Enter **delete** in the text input field.
5. Choose **Delete**.

Sample function code

To process events from Amazon Kinesis, iterate through the records included in the event object and decode the Base64-encoded data included in each.

Note

The code on this page does not support [aggregated records](#). You can disable aggregation in the Kinesis Producer Library [configuration](#), or use the [Kinesis Record Aggregation library](#) to deaggregate records.

Sample code is available for the following languages.

Topics

- [Node.js 12.x \(p. 701\)](#)
- [Java 11 \(p. 701\)](#)
- [C# \(p. 702\)](#)
- [Python 3 \(p. 703\)](#)
- [Go \(p. 703\)](#)

Node.js 12.x

The following example code receives a Kinesis event input and processes the messages that it contains. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Example index.js

```
console.log('Loading function');

exports.handler = function(event, context) {
    //console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        // Kinesis data is base64 encoded so decode here
        var payload = Buffer.from(record.kinesis.data, 'base64').toString('ascii');
        console.log('Decoded payload:', payload);
    });
};
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Node.js Lambda functions with .zip file archives \(p. 263\)](#).

Java 11

The following is example Java code that receives Kinesis event record data as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

In the code, `recordHandler` is the handler. The handler uses the predefined `KinesisEvent` class that is defined in the `aws-lambda-java-events` library.

Example ProcessKinesisEvents.java

```
package example;

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.KinesisEventRecord;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent, Void>{
    @Override
    public Void handleRequest(KinesisEvent event, Context context)
    {
        for(KinesisEventRecord rec : event.getRecords()) {
            System.out.println(new String(rec.getKinesis().getData().array()));
        }
        return null;
    }
}
```

If the handler returns normally without exceptions, Lambda considers the input batch of records as processed successfully and begins reading new records in the stream. If the handler throws an exception, Lambda considers the input batch of records as not processed and invokes the function with the same batch of records again.

Dependencies

- aws-lambda-java-core
- aws-lambda-java-events
- aws-java-sdk

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [Deploy Java Lambda functions with .zip or JAR file archives \(p. 393\)](#).

C#

The following is example C# code that receives Kinesis event record data as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

In the code, HandleKinesisRecord is the handler. The handler uses the predefined KinesisEvent class that is defined in the Amazon.Lambda.KinesisEvents library.

Example ProcessingKinesisEvents.cs

```
using System;
using System.IO;
using System.Text;

using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;

namespace KinesisStreams
{
    public class KinesisSample
    {
        [LambdaSerializer(typeof(JsonSerializer))]
        public void HandleKinesisRecord(KinesisEvent kinesisEvent)
        {
            Console.WriteLine($"Beginning to process {kinesisEvent.Records.Count} records...");
        }
    }
}
```

```
foreach (var record in kinesisEvent.Records)
{
    Console.WriteLine($"Event ID: {record.EventId}");
    Console.WriteLine($"Event Name: {record.EventName}");

    string recordData = GetRecordContents(record.Kinesis);
    Console.WriteLine($"Record Data:");
    Console.WriteLine(recordData);
}
Console.WriteLine("Stream processing complete.");
}

private string GetRecordContents(KinesisEvent.Record streamRecord)
{
    using (var reader = new StreamReader(streamRecord.Data, Encoding.ASCII))
    {
        return reader.ReadToEnd();
    }
}
```

Replace the `Program.cs` in a .NET Core project with the above sample. For instructions, see [Deploy C# Lambda functions with .zip file archives \(p. 497\)](#).

Python 3

The following is example Python code that receives Kinesis event record data as input and processes it. For illustration, the code writes to some of the incoming event data to CloudWatch Logs.

Example `ProcessKinesisRecords.py`

```
from __future__ import print_function
import json
import base64
def lambda_handler(event, context):
    for record in event['Records']:
        #Kinesis data is base64 encoded so decode here
        payload=base64.b64decode(record["kinesis"]["data"])
        print("Decoded payload: " + str(payload))
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Python Lambda functions with .zip file archives \(p. 324\)](#).

Go

The following is example Go code that receives Kinesis event record data as input and processes it. For illustration, the code writes to some of the incoming event data to CloudWatch Logs.

Example `ProcessKinesisRecords.go`

```
import (
    "strings"
    "github.com/aws/aws-lambda-go/events"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) {
    for _, record := range kinesisEvent.Records {
        kinesisRecord := record.Kinesis
```

```
    dataBytes := kinesisRecord.Data
    dataText := string(dataBytes)

    fmt.Printf("%s Data = %s \n", record.EventName, dataText)
}

}
```

Build the executable with `go build` and create a deployment package. For instructions, see [Deploy Go Lambda functions with .zip file archives \(p. 460\)](#).

AWS SAM template for a Kinesis application

You can build this application using [AWS SAM](#). To learn more about creating AWS SAM templates, see [AWS SAM template basics](#) in the *AWS Serverless Application Model Developer Guide*.

Below is a sample AWS SAM template for the Lambda application from the [tutorial \(p. 696\)](#). The function and handler in the template are for the Node.js code. If you use a different code sample, update the values accordingly.

Example template.yaml - Kinesis stream

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  LambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs12.x
      Timeout: 10
      Tracing: Active
      Events:
        Stream:
          Type: Kinesis
          Properties:
            Stream: !GetAtt stream.Arn
            BatchSize: 100
            StartingPosition: LATEST
      stream:
        Type: AWS::Kinesis::Stream
        Properties:
          ShardCount: 1
  Outputs:
    FunctionName:
      Description: "Function name"
      Value: !Ref LambdaFunction
    StreamARN:
      Description: "Stream ARN"
      Value: !GetAtt stream.Arn
```

The template creates a Lambda function, a Kinesis stream, and an event source mapping. The event source mapping reads from the stream and invokes the function.

To use an [HTTP/2 stream consumer \(p. 686\)](#), create the consumer in the template and configure the event source mapping to read from the consumer instead of from the stream.

Example template.yaml - Kinesis stream consumer

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

```
Description: A function that processes data from a Kinesis stream.
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs12.x
      Timeout: 10
      Tracing: Active
  Events:
    Stream:
      Type: Kinesis
      Properties:
        Stream: !GetAtt streamConsumer.ConsumerARN
        StartingPosition: LATEST
        BatchSize: 100
  stream:
    Type: "AWS::Kinesis::Stream"
    Properties:
      ShardCount: 1
  streamConsumer:
    Type: "AWS::Kinesis::StreamConsumer"
    Properties:
      StreamARN: !GetAtt stream.Arn
      ConsumerName: "TestConsumer"
Outputs:
  FunctionName:
    Description: "Function name"
    Value: !Ref function
  StreamARN:
    Description: "Stream ARN"
    Value: !GetAtt stream.Arn
  ConsumerARN:
    Description: "Stream consumer ARN"
    Value: !GetAtt streamConsumer.ConsumerARN
```

For information on how to package and deploy your serverless application using the package and deploy commands, see [Deploying serverless applications](#) in the *AWS Serverless Application Model Developer Guide*.

Using AWS Lambda with Amazon Lex

You can use Amazon Lex to integrate a conversation bot into your application. The Amazon Lex bot provides a conversational interface with your users. Amazon Lex provides prebuilt integration with Lambda, which enables you to use a Lambda function with your Amazon Lex bot.

When you configure an Amazon Lex bot, you can specify a Lambda function to perform validation, fulfillment, or both. For validation, Amazon Lex invokes the Lambda function after each response from the user. The Lambda function can validate the response and provide corrective feedback to the user, if necessary. For fulfillment, Amazon Lex invokes the Lambda function to fulfill the user request after the bot successfully collects all of the required information and receives confirmation from the user.

You can [manage the concurrency \(p. 197\)](#) of your Lambda function to control the maximum number of simultaneous bot conversations that you serve. The Amazon Lex API returns an HTTP 429 status code (Too Many Requests) if the function is at maximum concurrency.

The API returns an HTTP 424 status code (Dependency Failed Exception) if the Lambda function throws an exception.

The Amazon Lex bot invokes your Lambda function [synchronously \(p. 120\)](#). The event parameter contains information about the bot and the value of each slot in the dialog. For definitions of the event and response fields, see [Lambda event and response format](#) in the *Amazon Lex Developer Guide*. The invocationSource parameter in the Amazon Lex message event indicates whether the Lambda function should validate the inputs (DialogCodeHook) or fulfill the intent (FulfillmentCodeHook).

For an example tutorial that shows how to use Lambda with Amazon Lex, see [Exercise 1: Create Amazon Lex bot using a blueprint](#) in the *Amazon Lex Developer Guide*.

Roles and permissions

You need to configure a service-linked role as your function's [execution role \(p. 816\)](#). Amazon Lex defines the service-linked role with predefined permissions. When you create an Amazon Lex bot using the console, the service-linked role is created automatically. To create a service-linked role with the AWS CLI, use the `create-service-linked-role` command.

```
aws iam create-service-linked-role --aws-service-name lex.amazonaws.com
```

This command creates the following role.

```
{
  "Role": {
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Action": "sts:AssumeRole",
          "Effect": "Allow",
          "Principal": {
            "Service": "lex.amazonaws.com"
          }
        }
      ]
    },
    "RoleName": "AWSServiceRoleForLexBots",
    "Path": "/aws-service-role/lex.amazonaws.com/",
    "Arn": "arn:aws:iam::account-id:role/aws-service-role/lex.amazonaws.com/
AWSServiceRoleForLexBots"
  }
}
```

If your Lambda function uses other AWS services, you need to add the corresponding permissions to the service-linked role.

You use a resource-based permissions policy to allow the Amazon Lex intent to invoke your Lambda function. If you use the Amazon Lex console, the permissions policy is created automatically. From the AWS CLI, use the Lambda add-permission command to set the permission. The following example sets permission for the OrderFlowers intent.

```
aws lambda add-permission \
--function-name OrderFlowersCodeHook \
--statement-id LexGettingStarted-OrderFlowersBot \
--action lambda:InvokeFunction \
--principal lex.amazonaws.com \
--source-arn "arn:aws:lex:us-east-1:123456789012 ID:intent:OrderFlowers:*
```

Using Lambda with Amazon MQ

Note

If you want to send data to a target other than a Lambda function or enrich the data before sending it, see [Amazon EventBridge Pipes](#).

Amazon MQ is a managed message broker service for [Apache ActiveMQ](#) and [RabbitMQ](#). A *message broker* enables software applications and components to communicate using various programming languages, operating systems, and formal messaging protocols through either topic or queue event destinations.

Amazon MQ can also manage Amazon Elastic Compute Cloud (Amazon EC2) instances on your behalf by installing ActiveMQ or RabbitMQ brokers and by providing different network topologies and other infrastructure needs.

You can use a Lambda function to process records from your Amazon MQ message broker. Lambda invokes your function through an [event source mapping \(p. 131\)](#), a Lambda resource that reads messages from your broker and invokes the function [synchronously \(p. 120\)](#).

The Amazon MQ event source mapping has the following configuration restrictions:

- Cross account – Lambda does not support cross-account processing. You cannot use Lambda to process records from an Amazon MQ message broker that is in a different AWS account.
- Authentication – For ActiveMQ, only the ActiveMQ [SimpleAuthenticationPlugin](#) is supported. For RabbitMQ, only the [PLAIN](#) authentication mechanism is supported. Users must use AWS Secrets Manager to manage their credentials. For more information about ActiveMQ authentication, see [Integrating ActiveMQ brokers with LDAP](#) in the *Amazon MQ Developer Guide*.
- Connection quota – Brokers have a maximum number of allowed connections per wire-level protocol. This quota is based on the broker instance type. For more information, see the [Brokers](#) section of [Quotas in Amazon MQ](#) in the *Amazon MQ Developer Guide*.
- Connectivity – You can create brokers in a public or private virtual private cloud (VPC). For private VPCs, your Lambda function needs access to the VPC to receive messages. For more information, see the section called [“Event source mapping API” \(p. 712\)](#) later in this topic.
- Event destinations – Only queue destinations are supported. However, you can use a virtual topic, which behaves as a topic internally while interacting with Lambda as a queue. For more information, see [Virtual Destinations](#) on the Apache ActiveMQ website, and [Virtual Hosts](#) on the RabbitMQ website.
- Network topology – For ActiveMQ, only one single-instance or standby broker is supported per event source mapping. For RabbitMQ, only one single-instance broker or cluster deployment is supported per event source mapping. Single-instance brokers require a failover endpoint. For more information about these broker deployment modes, see [Active MQ Broker Architecture](#) and [Rabbit MQ Broker Architecture](#) in the *Amazon MQ Developer Guide*.
- Protocols – Supported protocols depend on the type of Amazon MQ integration.
 - For ActiveMQ integrations, Lambda consumes messages using the OpenWire/Java Message Service (JMS) protocol. No other protocols are supported for consuming messages. Within the JMS protocol, only [TextMessage](#) and [BytesMessage](#) are supported. Lambda also supports JMS custom properties. For more information about the OpenWire protocol, see [OpenWire](#) on the Apache ActiveMQ website.
 - For RabbitMQ integrations, Lambda consumes messages using the AMQP 0-9-1 protocol. No other protocols are supported for consuming messages. For more information about RabbitMQ's implementation of the AMQP 0-9-1 protocol, see [AMQP 0-9-1 Complete Reference Guide](#) on the RabbitMQ website.

Lambda automatically supports the latest versions of ActiveMQ and RabbitMQ that Amazon MQ supports. For the latest supported versions, see [Amazon MQ release notes](#) in the *Amazon MQ Developer Guide*.

Note

By default, Amazon MQ has a weekly maintenance window for brokers. During that window of time, brokers are unavailable. For brokers without standby, Lambda cannot process any messages during that window.

Sections

- [Lambda consumer group \(p. 709\)](#)
- [Execution role permissions \(p. 711\)](#)
- [Configuring a broker as an event source \(p. 712\)](#)
- [Event source mapping API \(p. 712\)](#)
- [Event source mapping errors \(p. 714\)](#)
- [Amazon MQ and RabbitMQ configuration parameters \(p. 715\)](#)

Lambda consumer group

To interact with Amazon MQ, Lambda creates a consumer group which can read from your Amazon MQ brokers. The consumer group is created with the same ID as the event source mapping UUID.

For Amazon MQ event sources, Lambda batches records together and sends them to your function in a single payload. To control behavior, you can configure the batching window and batch size. Lambda pulls messages until it processes the payload size maximum of 6 MB, the batching window expires, or the number of records reaches the full batch size. For more information, see [Batching behavior \(p. 132\)](#).

The consumer group retrieves the messages as a BLOB of bytes, base64-encodes them into a single JSON payload, and then invokes your function. If your function returns an error for any of the messages in a batch, Lambda retries the whole batch of messages until processing succeeds or the messages expire.

Note

While Lambda functions typically have a maximum timeout limit of 15 minutes, event source mappings for Amazon MSK, self-managed Apache Kafka, and Amazon MQ for ActiveMQ and RabbitMQ only support functions with maximum timeout limits of 14 minutes. This constraint ensures that the event source mapping can properly handle function errors and retries.

You can monitor a given function's concurrency usage using the `ConcurrentExecutions` metric in Amazon CloudWatch. For more information about concurrency, see [the section called "Configuring reserved concurrency" \(p. 210\)](#).

Example Amazon MQ record events

ActiveMQ

```
{  
    "eventSource": "aws:mq",  
    "eventSourceArn": "arn:aws:mq:us-west-2:111122223333:broker:test:b-9bcfa592-423a-4942-879d-eb284b418fc8",  
    "messages": [  
        {  
            "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-west-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,  
            "timestamp": 1577832000000  
        }  
    ]  
}
```

```

    "destination": {
        "physicalName": "testQueue"
    },
    "data": "QUJD0kFBQUE=",
    "timestamp": 1598827811958,
    "brokerInTime": 1598827811958,
    "brokerOutTime": 1598827811959,
    "properties": {
        "index": "1",
        "doAlarm": "false",
        "myCustomProperty": "value"
    }
},
{
    "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
west-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-
west-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": "eyJ0aW1lbnV0IjowLCJkYXRhIjoiQ1pybWwR3c4T3Y0YnFMUXhENEUiFQ=="
]
}
}
```

Note

In the RabbitMQ example, pizzaQueue is the name of the RabbitMQ queue, and / is the name of the virtual host. When receiving messages, the event source lists messages under pizzaQueue::/.

Execution role permissions

To read records from an Amazon MQ broker, your Lambda function needs the following permissions added to its [execution role \(p. 816\)](#):

- [mq:DescribeBroker](#)
- [secretsmanager:GetSecretValue](#)
- [ec2>CreateNetworkInterface](#)
- [ec2>DeleteNetworkInterface](#)
- [ec2:DescribeNetworkInterfaces](#)
- [ec2:DescribeSecurityGroups](#)
- [ec2:DescribeSubnets](#)
- [ec2:DescribeVpcs](#)
- [logs>CreateLogGroup](#)
- [logs>CreateLogStream](#)
- [logs:PutLogEvents](#)

Note

When using an encrypted customer managed key, add the [kms:Decrypt](#) permission as well.

Configuring a broker as an event source

Create an [event source mapping \(p. 131\)](#) to tell Lambda to send records from an Amazon MQ broker to a Lambda function. You can create multiple event source mappings to process the same data with multiple functions, or to process items from multiple sources with a single function.

To configure your function to read from Amazon MQ, create an **MQ** trigger in the Lambda console.

To create a trigger

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of a function.
3. Under **Function overview**, choose **Add trigger**.
4. Choose a trigger type.
5. Configure the required options, and then choose **Add**.

Lambda supports the following options for Amazon MQ event sources:

- **MQ broker** – Select an Amazon MQ broker.
- **Batch size** – Set the maximum number of messages to retrieve in a single batch.
- **Queue name** – Enter the Amazon MQ queue to consume.
- **Source access configuration** – Enter virtual host information and the Secrets Manager secret that stores your broker credentials.
- **Enable trigger** – Disable the trigger to stop processing records.

To enable or disable the trigger (or delete it), choose the **MQ** trigger in the designer. To reconfigure the trigger, use the event source mapping API operations.

Event source mapping API

To manage an event source with the [AWS Command Line Interface \(AWS CLI\)](#) or an [AWS SDK](#), you can use the following API operations:

- [CreateEventSourceMapping \(p. 1153\)](#)
- [ListEventSourceMappings \(p. 1279\)](#)
- [GetEventSourceMapping \(p. 1214\)](#)
- [UpdateEventSourceMapping \(p. 1356\)](#)
- [DeleteEventSourceMapping \(p. 1186\)](#)

Note

When you update, disable, or delete an event source mapping for Amazon MQ, it can take up to 15 minutes for your changes to take effect. Before this period has elapsed, your event source mapping may continue to process events and invoke your function using your previous settings. This is true even when the status of the event source mapping displayed in the console indicates that your changes have been applied.

To create the event source mapping with the AWS Command Line Interface (AWS CLI), use the [create-event-source-mapping](#) command.

By default, Amazon MQ brokers are created with the `PubliclyAccessible` flag set to false. It is only when `PubliclyAccessible` is set to true that the broker receives a public IP address.

For full access with your event source mapping, your broker must either use a public endpoint or provide access to the VPC. Note that when you add Amazon MQ as a trigger, Lambda assumes the VPC settings of the Amazon MQ broker, not the Lambda function's VPC settings. To meet the Amazon Virtual Private Cloud (Amazon VPC) access requirements, you can do one of the following:

- Configure one NAT gateway per public subnet. For more information, see [Internet and service access for VPC-connected functions \(p. 228\)](#).
- Create a connection between your Amazon VPC and Lambda. Your Amazon VPC must also connect to AWS Security Token Service (AWS STS) and Secrets Manager endpoints. For more information, see [Configuring interface VPC endpoints for Lambda \(p. 229\)](#).

The Amazon VPC security group rules that you configure should have the following settings at minimum:

- Inbound rules – For a broker without public accessibility, allow all traffic on all ports for the security group that's specified as your source. For a broker with public accessibility, allow all traffic on all ports for all destinations.
- Outbound rules – Allow all traffic on all ports for all destinations.

The Amazon VPC configuration is discoverable through the [Amazon MQ API](#) and does not need to be configured in the create-event-source-mapping setup.

The following example AWS CLI command creates an event source which maps a Lambda function named MQ-Example-Function to an Amazon MQ RabbitMQ-based broker named ExampleMQBroker. The command also provides the virtual host name and a Secrets Manager secret ARN that stores the broker credentials.

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:mq:us-east-1:123456789012:broker:ExampleMQBroker:b-24cacbb4-
b295-49b7-8543-7ce7ce9dfb98 \
--function-name arn:aws:lambda:us-east-1:123456789012:function:MQ-Example-Function \
--queues ExampleQueue \
--source-access-configuration Type=VIRTUAL_HOST,URI="/" \
Type=BASIC_AUTH,URI=arn:aws:secretsmanager:us-
east-1:123456789012:secret:ExampleMQBrokerUserPassword-xPBMTt \
```

You should see the following output:

```
{
    "UUID": "91eaeb7e-c976-1234-9451-8709db01f137",
    "BatchSize": 100,
    "EventSourceArn": "arn:aws:mq:us-east-1:123456789012:broker:ExampleMQBroker:b-b4d492ef-
bdc3-45e3-a781-cd1a3102ecca",
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:MQ-Example-Function",
    "LastModified": 1601927898.741,
    "LastProcessingResult": "No records processed",
    "State": "Creating",
    "StateTransitionReason": "USER_INITIATED",
    "Queues": [
        "ExampleQueue"
    ],
    "SourceAccessConfigurations": [
        {
            "Type": "BASIC_AUTH",
            "URI": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:ExampleMQBrokerUserPassword-xPBMTt"
        }
    ]
}
```

Using the [update-event-source-mapping](#) command, you can configure additional options such as how Lambda processes batches and to specify when to discard records that cannot be processed. The following example command updates an event source mapping to have a batch size of 2.

```
aws lambda update-event-source-mapping \
--uuid 91eaeb7e-c976-1234-9451-8709db01f137 \
--batch-size 2
```

You should see the following output:

```
{  
    "UUID": "91eaeb7e-c976-1234-9451-8709db01f137",  
    "BatchSize": 2,  
    "EventSourceArn": "arn:aws:mq:us-east-1:123456789012:broker:ExampleMQBroker:b-b4d492ef-bdc3-45e3-a781-cd1a3102ecca",  
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:MQ-Example-Function",  
    "LastModified": 1601928393.531,  
    "LastProcessingResult": "No records processed",  
    "State": "Updating",  
    "StateTransitionReason": "USER_INITIATED"  
}
```

Lambda updates these settings asynchronously. The output will not reflect changes until this process completes. To view the current status of your resource, use the [get-event-source-mapping](#) command.

```
aws lambda get-event-source-mapping \
--uuid 91eaeb7e-c976-4939-9451-8709db01f137
```

You should see the following output:

```
{  
    "UUID": "91eaeb7e-c976-4939-9451-8709db01f137",  
    "BatchSize": 2,  
    "EventSourceArn": "arn:aws:mq:us-east-1:123456789012:broker:ExampleMQBroker:b-b4d492ef-bdc3-45e3-a781-cd1a3102ecca",  
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:MQ-Example-Function",  
    "LastModified": 1601928393.531,  
    "LastProcessingResult": "No records processed",  
    "State": "Enabled",  
    "StateTransitionReason": "USER_INITIATED"  
}
```

Event source mapping errors

When a Lambda function encounters an unrecoverable error, your Amazon MQ consumer stops processing records. Any other consumers can continue processing, provided that they do not encounter the same error. To determine the potential cause of a stopped consumer, check the StateTransitionReason field in the return details of your EventSourceMapping for one of the following codes:

ESM_CONFIG_NOT_VALID

The event source mapping configuration is not valid.

EVENT_SOURCE_AUTHN_ERROR

Lambda failed to authenticate the event source.

EVENT_SOURCE_AUTHZ_ERROR

Lambda does not have the required permissions to access the event source.

FUNCTION_CONFIG_NOT_VALID

The function's configuration is not valid.

Records also go unprocessed if Lambda drops them due to their size. The size limit for Lambda records is 6 MB. To redeliver messages upon function error, you can use a dead-letter queue (DLQ). For more information, see [Message Redelivery and DLQ Handling](#) on the Apache ActiveMQ website and [Reliability Guide](#) on the RabbitMQ website.

Note

Lambda does not support custom redelivery policies. Instead, Lambda uses a policy with the default values from the [Redelivery Policy](#) page on the Apache ActiveMQ website, with maximumRedeliveries set to 5.

Amazon MQ and RabbitMQ configuration parameters

All Lambda event source types share the same [CreateEventSourceMapping \(p. 1153\)](#) and [UpdateEventSourceMapping \(p. 1356\)](#) API operations. However, only some of the parameters apply to Amazon MQ and RabbitMQ.

Event source parameters that apply to Amazon MQ and RabbitMQ

Parameter	Required	Default	Notes
BatchSize	N	100	Maximum: 10,000
Enabled	N	true	
FunctionName	Y		
FilterCriteria	N		Lambda event filtering (p. 136)
MaximumBatchingWindowInSeconds		500 ms	Batching behavior (p. 132)
Queues	N		The name of the Amazon MQ broker destination queue to consume.
SourceAccessConfiguration	N		For ActiveMQ, BASIC_AUTH credentials. For RabbitMQ, can contain both BASIC_AUTH credentials and VIRTUAL_HOST information.

Using Lambda with Amazon MSK

Note

If you want to send data to a target other than a Lambda function or enrich the data before sending it, see [Amazon EventBridge Pipes](#).

[Amazon Managed Streaming for Apache Kafka \(Amazon MSK\)](#) is a fully managed service that you can use to build and run applications that use Apache Kafka to process streaming data. Amazon MSK simplifies the setup, scaling, and management of clusters running Kafka. Amazon MSK also makes it easier to configure your application for multiple Availability Zones and for security with AWS Identity and Access Management (IAM). Amazon MSK supports multiple open-source versions of Kafka.

Amazon MSK as an event source operates similarly to using Amazon Simple Queue Service (Amazon SQS) or Amazon Kinesis. Lambda internally polls for new messages from the event source and then synchronously invokes the target Lambda function. Lambda reads the messages in batches and provides these to your function as an event payload. The maximum batch size is configurable. (The default is 100 messages.)

For an example of how to configure Amazon MSK as an event source, see [Using Amazon MSK as an event source for AWS Lambda](#) on the AWS Compute Blog. For a complete tutorial, see [Amazon MSK Lambda Integration](#) in the Amazon MSK Labs.

For Kafka-based event sources, Lambda supports processing control parameters, such as batching windows and batch size. For more information, see [Batching behavior \(p. 132\)](#).

Lambda reads the messages sequentially for each partition. A single Lambda payload can contain messages from multiple partitions. After Lambda processes each batch, it commits the offsets of the messages in that batch. If your function returns an error for any of the messages in a batch, Lambda retries the whole batch of messages until processing succeeds or the messages expire.

Note

While Lambda functions typically have a maximum timeout limit of 15 minutes, event source mappings for Amazon MSK, self-managed Apache Kafka, and Amazon MQ for ActiveMQ and RabbitMQ only support functions with maximum timeout limits of 14 minutes. This constraint ensures that the event source mapping can properly handle function errors and retries.

Topics

- [Example event \(p. 716\)](#)
- [MSK cluster authentication \(p. 717\)](#)
- [Managing API access and permissions \(p. 720\)](#)
- [Authentication and authorization errors \(p. 722\)](#)
- [Network configuration \(p. 723\)](#)
- [Adding Amazon MSK as an event source \(p. 724\)](#)
- [Auto scaling of the Amazon MSK event source \(p. 725\)](#)
- [Amazon CloudWatch metrics \(p. 726\)](#)
- [Amazon MSK configuration parameters \(p. 726\)](#)

Example event

Lambda sends the batch of messages in the event parameter when it invokes your function. The event payload contains an array of messages. Each array item contains details of the Amazon MSK topic and partition identifier, together with a timestamp and a base64-encoded message.

```
{
```

```
"eventSource":"aws:kafka",
"eventSourceArn":"arn:aws:kafka:sa-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":"abcDEFghiJKLMnnoPQRstuVWXYZ1234==",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers": [
                {
                    "headerKey": [
                        104,
                        101,
                        97,
                        100,
                        101,
                        114,
                        86,
                        97,
                        108,
                        117,
                        101
                    ]
                }
            ]
        }
    ]
}
```

MSK cluster authentication

Lambda needs permission to access the Amazon MSK cluster, retrieve records, and perform other tasks. Amazon MSK supports several options for controlling client access to the MSK cluster.

Cluster access options

- [Unauthenticated access \(p. 717\)](#)
- [SASL/SCRAM authentication \(p. 717\)](#)
- [IAM role-based authentication \(p. 718\)](#)
- [Mutual TLS authentication \(p. 718\)](#)
- [Configuring the mTLS secret \(p. 673\)](#)
- [How Lambda chooses a bootstrap broker \(p. 720\)](#)

Unauthenticated access

If no clients access the cluster over the internet, you can use unauthenticated access.

SASL/SCRAM authentication

Amazon MSK supports Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) authentication with Transport Layer Security (TLS) encryption.

For Lambda to connect to the cluster, you store the authentication credentials (user name and password) in an AWS Secrets Manager secret.

For more information about using Secrets Manager, see [User name and password authentication with AWS Secrets Manager](#) in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

Amazon MSK doesn't support SASL/PLAIN authentication.

IAM role-based authentication

You can use IAM to authenticate the identity of clients that connect to the MSK cluster. If IAM auth is active on your MSK cluster, and you don't provide a secret for auth, Lambda automatically defaults to using IAM auth. To create and deploy user or role-based policies, use the IAM console or API. For more information, see [IAM access control](#) in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

To allow Lambda to connect to the MSK cluster, read records, and perform other required actions, add the following permissions to your function's [execution role \(p. 816\)](#).

```
{  
    "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:region:account-id:cluster/cluster-name/cluster-uuid",  
                "arn:aws:kafka:region:account-id:topic/cluster-name/cluster-uuid/topic-name",  
                "arn:aws:kafka:region:account-id:group/cluster-name/cluster-uuid/consumer-group-id"  
            ]  
        }  
    ]  
}
```

You can scope these permissions to a specific cluster, topic, and group. For more information, see the [Amazon MSK Kafka actions](#) in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

Mutual TLS authentication

Mutual TLS (mTLS) provides two-way authentication between the client and server. The client sends a certificate to the server for the server to verify the client, and the server sends a certificate to the client for the client to verify the server.

For Amazon MSK, Lambda acts as the client. You configure a client certificate (as a secret in Secrets Manager) to authenticate Lambda with the brokers in your MSK cluster. The client certificate must be signed by a CA in the server's trust store. The MSK cluster sends a server certificate to Lambda to authenticate the brokers with Lambda. The server certificate must be signed by a certificate authority (CA) that's in the AWS trust store.

For instructions on how to generate a client certificate, see [Introducing mutual TLS authentication for Amazon MSK as an event source](#).

Amazon MSK doesn't support self-signed server certificates, because all brokers in Amazon MSK use [public certificates](#) signed by [Amazon Trust Services CAs](#), which Lambda trusts by default.

For more information about mTLS for Amazon MSK, see [Mutual TLS Authentication](#) in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

Configuring the mTLS secret

The CLIENT_CERTIFICATE_TLS_AUTH secret requires a certificate field and a private key field. For an encrypted private key, the secret requires a private key password. Both the certificate and private key must be in PEM format.

Note

Lambda supports the [PBES1](#) (but not PBES2) private key encryption algorithms.

The certificate field must contain a list of certificates, beginning with the client certificate, followed by any intermediate certificates, and ending with the root certificate. Each certificate must start on a new line with the following structure:

```
-----BEGIN CERTIFICATE-----  
<certificate contents>  
-----END CERTIFICATE-----
```

Secrets Manager supports secrets up to 65,536 bytes, which is enough space for long certificate chains.

The private key must be in [PKCS #8](#) format, with the following structure:

```
-----BEGIN PRIVATE KEY-----  
<private key contents>  
-----END PRIVATE KEY-----
```

For an encrypted private key, use the following structure:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----  
<private key contents>  
-----END ENCRYPTED PRIVATE KEY-----
```

The following example shows the contents of a secret for mTLS authentication using an encrypted private key. For an encrypted private key, you include the private key password in the secret.

```
{  
    "privateKeyPassword": "testpassword",  
    "certificate": "-----BEGIN CERTIFICATE-----  
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvvcNAQELBQAW  
...  
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPim040s1cRqtFHXoa10QQbIlxk  
cmUiAi9R0=  
-----END CERTIFICATE-----  
-----BEGIN CERTIFICATE-----  
MIIFgjCCA2qgAwIBAgIQdjNZd6uFF9hbNC5RdfmHzANBgkqhkiG9w0BAQsFADbb  
...  
rQoiowbbk5wXCheYSANQIFTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no  
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==  
-----END CERTIFICATE-----",  
    "privateKey": "-----BEGIN ENCRYPTED PRIVATE KEY-----  
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
```

```
...
QrSekqF+kWzmB6nAfSzg09IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----
}
```

How Lambda chooses a bootstrap broker

Lambda chooses a [bootstrap broker](#) based on the authentication methods available on your cluster, and whether you provide a secret for authentication. If you provide a secret for mTLS or SASL/SCRAM, Lambda automatically chooses that auth method. If you don't provide a secret, Lambda selects the strongest auth method that's active on your cluster. The following is the order of priority in which Lambda selects a broker, from strongest to weakest auth:

- mTLS (secret provided for mTLS)
- SASL/SCRAM (secret provided for SASL/SCRAM)
- SASL IAM (no secret provided, and IAM auth active)
- Unauthenticated TLS (no secret provided, and IAM auth not active)
- Plaintext (no secret provided, and both IAM auth and unauthenticated TLS are not active)

Note

If Lambda can't connect to the most secure broker type, Lambda doesn't attempt to connect to a different (weaker) broker type. If you want Lambda to choose a weaker broker type, deactivate all stronger auth methods on your cluster.

Managing API access and permissions

In addition to accessing the Amazon MSK cluster, your function needs permissions to perform various Amazon MSK API actions. You add these permissions to the function's execution role. If your users need access to any of the Amazon MSK API actions, add the required permissions to the identity policy for the user or role.

You can add each of the following permissions to your execution role manually. Alternatively, you can attach the AWS managed policy `AWSLambdaMSKExecutionRole` to your execution role. The `AWSLambdaMSKExecutionRole` policy contains all required API actions and VPC permissions listed below.

Required Lambda function execution role permissions

To create and store logs in a log group in Amazon CloudWatch Logs, your Lambda function must have the following permissions in its execution role:

- [logs>CreateLogGroup](#)
- [logs>CreateLogStream](#)
- [logs:PutLogEvents](#)

For Lambda to access your Amazon MSK cluster on your behalf, your Lambda function must have the following permissions in its execution role:

- [kafka:DescribeCluster](#)
- [kafka:DescribeClusterV2](#)
- [kafka:GetBootstrapBrokers](#)

You only need to add one of either `kafka:DescribeCluster` or `kafka:DescribeClusterV2`. For provisioned MSK clusters, either permission works. For serverless MSK clusters, you must use `kafka:DescribeClusterV2`.

Note

Lambda eventually plans to remove the `kafka:DescribeCluster` permission from the associated `AWSLambdaMSKExecutionRole` managed policy. If you use this policy, you should migrate any applications using `kafka:DescribeCluster` to use `kafka:DescribeClusterV2` instead.

VPC permissions

If only users within a VPC can access your Amazon MSK cluster, your Lambda function must have permission to access your Amazon VPC resources. These resources include your VPC, subnets, security groups, and network interfaces. To access these resources, your function's execution role must have the following permissions:

- [`ec2:CreateNetworkInterface`](#)
- [`ec2:DescribeNetworkInterfaces`](#)
- [`ec2:DescribeVpcs`](#)
- [`ec2:DeleteNetworkInterface`](#)
- [`ec2:DescribeSubnets`](#)
- [`ec2:DescribeSecurityGroups`](#)

Optional Lambda function permissions

Your Lambda function might also need permissions to:

- Access your SCRAM secret, if using SASL/SCRAM authentication.
- Describe your Secrets Manager secret.
- Access your AWS Key Management Service (AWS KMS) customer managed key.

Secrets Manager and AWS KMS permissions

Depending on the type of access control that you're configuring for your Amazon MSK brokers, your Lambda function might need permission to access your SCRAM secret (if using SASL/SCRAM authentication), or Secrets Manager secret to decrypt your AWS KMS customer managed key. To access these resources, your function's execution role must have the following permissions:

- [`kafka>ListScramSecrets`](#)
- [`secretsmanager:GetSecretValue`](#)
- [`kms:Decrypt`](#)

Adding permissions to your execution role

Follow these steps to add the AWS managed policy `AWSLambdaMSKExecutionRole` to your execution role using the IAM console.

To add an AWS managed policy

1. Open the [Policies page](#) of the IAM console.
2. In the search box, enter the policy name (`AWSLambdaMSKExecutionRole`).

3. Select the policy from the list, and then choose **Policy actions, Attach**.
4. On the **Attach policy** page, select your execution role from the list, and then choose **Attach policy**.

Granting users access with an IAM policy

By default, users and roles don't have permission to perform Amazon MSK API operations. To grant access to users in your organization or account, you can add or update an identity-based policy. For more information, see [Amazon MSK Identity-Based Policy Examples](#) in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

Using SASL/SCRAM authentication

Amazon MSK supports Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) authentication with TLS encryption. You can control access to your Amazon MSK clusters by setting up user name and password authentication using an AWS Secrets Manager secret. For more information, see [Username and password authentication with AWS Secrets Manager](#) in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

Note that Amazon MSK does not support SASL/PLAIN authentication.

Authentication and authorization errors

If any of the permissions required to consume data from the Amazon MSK cluster are missing, Lambda displays one of the following error messages in the event source mapping under **LastProcessingResult**.

Error messages

- [Cluster failed to authorize Lambda \(p. 722\)](#)
- [SASL authentication failed \(p. 723\)](#)
- [Server failed to authenticate Lambda \(p. 723\)](#)
- [Provided certificate or private key is invalid \(p. 723\)](#)

Cluster failed to authorize Lambda

For SASL/SCRAM or mTLS, this error indicates that the provided user doesn't have all of the following required Kafka access control list (ACL) permissions:

- `DescribeConfigs Cluster`
- `Describe Group`
- `Read Group`
- `Describe Topic`
- `Read Topic`

For IAM access control, your function's execution role is missing one or more of the permissions required to access the group or topic. Review the list of required permissions in [the section called "IAM role-based authentication" \(p. 718\)](#).

When you create either Kafka ACLs or an IAM policy with the required Kafka cluster permissions, specify the topic and group as resources. The topic name must match the topic in the event source mapping. The group name must match the event source mapping's UUID.

After you add the required permissions to the execution role, it might take several minutes for the changes to take effect.

SASL authentication failed

For SASL/SCRAM, this error indicates that the provided user name and password aren't valid.

For IAM access control, the execution role is missing the `kafka-cluster:Connect` permission for the MSK cluster. Add this permission to the role and specify the cluster's Amazon Resource Name (ARN) as a resource.

You might see this error occurring intermittently. The cluster rejects connections after the number of TCP connections exceeds the [Amazon MSK service quota](#). Lambda backs off and retries until a connection is successful. After Lambda connects to the cluster and polls for records, the last processing result changes to OK.

Server failed to authenticate Lambda

This error indicates that the Amazon MSK Kafka brokers failed to authenticate with Lambda. This can occur for any of the following reasons:

- You didn't provide a client certificate for mTLS authentication.
- You provided a client certificate, but the brokers aren't configured to use mTLS.
- A client certificate isn't trusted by the brokers.

Provided certificate or private key is invalid

This error indicates that the Amazon MSK consumer couldn't use the provided certificate or private key. Make sure that the certificate and key use PEM format, and that the private key encryption uses a PBES1 algorithm.

Network configuration

Lambda must have access to the Amazon Virtual Private Cloud (Amazon VPC) resources associated with your Amazon MSK cluster. We recommend that you deploy [AWS PrivateLink VPC endpoints](#) for Lambda and AWS Security Token Service (AWS STS). The poller needs access to AWS STS to assume the execution role associated with the Lambda function. Lambda must have access to the Lambda VPC endpoint to invoke the function. If you configured a secret in Secrets Manager to authenticate Lambda with the brokers, also [deploy a VPC endpoint for Secrets Manager](#).

Alternatively, ensure that the VPC associated with your MSK cluster includes one NAT gateway per public subnet. For more information, see [Internet and service access for VPC-connected functions \(p. 228\)](#).

Configure your Amazon VPC security groups with the following rules (at minimum):

- Inbound rules – Allow all traffic on the Amazon MSK broker port (9092 for plaintext, 9094 for TLS, 9096 for SASL, 9098 for IAM) for the security groups specified for your event source.
- Outbound rules – Allow all traffic on port 443 for all destinations. Allow all traffic on the Amazon MSK broker port (9092 for plaintext, 9094 for TLS, 9096 for SASL, 9098 for IAM) for the security groups specified for your event source.
- If you are using VPC endpoints instead of a NAT gateway, the security groups associated with the VPC endpoints must allow all inbound traffic on port 443 from the event source's security groups.

Note

Your Amazon VPC configuration is discoverable through the [Amazon MSK API](#). You don't need to configure it during setup using the `create-event-source-mapping` command.

For more information about configuring the network, see [Setting up AWS Lambda with an Apache Kafka cluster within a VPC](#) on the AWS Compute Blog.

Adding Amazon MSK as an event source

To create an [event source mapping \(p. 131\)](#), add Amazon MSK as a Lambda function [trigger \(p. 9\)](#) using the Lambda console, an [AWS SDK](#), or the [AWS Command Line Interface \(AWS CLI\)](#). Note that when you add Amazon MSK as a trigger, Lambda assumes the VPC settings of the Amazon MSK cluster, not the Lambda function's VPC settings.

This section describes how to create an event source mapping using the Lambda console and the AWS CLI.

Note

When you update, disable, or delete an event source mapping for Amazon MSK, it can take up to 15 minutes for your changes to take effect. Before this period has elapsed, your event source mapping may continue to process events and invoke your function using your previous settings. This is true even when the status of the event source mapping displayed in the console indicates that your changes have been applied.

Prerequisites

- An Amazon MSK cluster and a Kafka topic. For more information, see [Getting Started Using Amazon MSK](#) in the [Amazon Managed Streaming for Apache Kafka Developer Guide](#).
- An [execution role \(p. 816\)](#) with permission to access the AWS resources that your MSK cluster uses.

Customizable consumer group ID

When setting up Kafka as an event source, you can specify a consumer group ID. This consumer group ID is an existing identifier for the Kafka consumer group that you want your Lambda function to join. You can use this feature to seamlessly migrate any ongoing Kafka record processing setups from other consumers to Lambda.

If you specify a consumer group ID and there are other active pollers within that consumer group, Kafka distributes messages across all consumers. In other words, Lambda doesn't receive all message for the Kafka topic. If you want Lambda to handle all messages in the topic, turn off any other pollers in that consumer group.

Additionally, if you specify a consumer group ID, and Kafka finds a valid existing consumer group with the same ID, Lambda ignores the `StartingPosition` parameter for your event source mapping. Instead, Lambda begins processing records according to the committed offset of the consumer group. If you specify a consumer group ID, and Kafka cannot find an existing consumer group, then Lambda configures your event source with the specified `StartingPosition`.

The consumer group ID that you specify must be unique among all your Kafka event sources. After creating a Kafka event source mapping with the consumer group ID specified, you cannot update this value.

Adding an Amazon MSK trigger (console)

Follow these steps to add your Amazon MSK cluster and a Kafka topic as a trigger for your Lambda function.

To add an Amazon MSK trigger to your Lambda function (console)

1. Open the [Functions page](#) of the Lambda console.

2. Choose the name of your Lambda function.
3. Under **Function overview**, choose **Add trigger**.
4. Under **Trigger configuration**, do the following:
 - a. Choose the **MSK** trigger type.
 - b. For **MSK cluster**, select your cluster.
 - c. For **Batch size**, enter the maximum number of messages to receive in a single batch.
 - d. For **Batch window**, enter the maximum amount of seconds that Lambda spends gathering records before invoking the function.
 - e. For **Topic name**, enter the name of a Kafka topic.
 - f. (Optional) For **Consumer group ID**, enter the ID of a Kafka consumer group to join.
 - g. (Optional) For **Starting position**, choose **Latest** to start reading the stream from the latest record. Or, choose **Trim horizon** to start at the earliest available record.
 - h. (Optional) For **Authentication**, choose the secret key for authenticating with the brokers in your MSK cluster.
 - i. To create the trigger in a disabled state for testing (recommended), clear **Enable trigger**. Or, to enable the trigger immediately, select **Enable trigger**.
5. To create the trigger, choose **Add**.

Adding an Amazon MSK trigger (AWS CLI)

Use the following example AWS CLI commands to create and view an Amazon MSK trigger for your Lambda function.

Creating a trigger using the AWS CLI

The following example uses the [create-event-source-mapping](#) AWS CLI command to map a Lambda function named `my-kafka-function` to a Kafka topic named `AWSKafkaTopic`. The topic's starting position is set to `LATEST`.

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:kafka:us-west-2:111111111111:cluster/my-cluster/fc2f5bdf-
fd1b-45ad-85dd-15b4a5a6247e-2 \
--topics AWSKafkaTopic \
--starting-position LATEST \
--function-name my-kafka-function
```

For more information, see the [CreateEventSourceMapping \(p. 1153\)](#) API reference documentation.

Viewing the status using the AWS CLI

The following example uses the [get-event-source-mapping](#) AWS CLI command to describe the status of the event source mapping that you created.

```
aws lambda get-event-source-mapping \
--uuid 6d9bce8e-836b-442c-8070-74e77903c815
```

Auto scaling of the Amazon MSK event source

When you initially create an Amazon MSK event source, Lambda allocates one consumer to process all partitions in the Kafka topic. Each consumer has multiple processors running in parallel to handle

increased workloads. Additionally, Lambda automatically scales up or down the number of consumers, based on workload. To preserve message ordering in each partition, the maximum number of consumers is one consumer per partition in the topic.

In one-minute intervals, Lambda evaluates the consumer offset lag of all the partitions in the topic. If the lag is too high, the partition is receiving messages faster than Lambda can process them. If necessary, Lambda adds or removes consumers from the topic. The scaling process of adding or removing consumers occurs within three minutes of evaluation.

If your target Lambda function is throttled, Lambda reduces the number of consumers. This action reduces the workload on the function by reducing the number of messages that consumers can retrieve and send to the function.

To monitor the throughput of your Kafka topic, view the [Offset lag metric \(p. 726\)](#) Lambda emits while your function processes records.

To check how many function invocations occur in parallel, you can also monitor the [concurrency metrics \(p. 872\)](#) for your function.

Amazon CloudWatch metrics

Lambda emits the OffsetLag metric while your function processes records. The value of this metric is the difference in offset between the last record written to the Kafka event source topic and the last record that your function's consumer group processed. You can use OffsetLag to estimate the latency between when a record is added and when your consumer group processes it.

An increasing trend in OffsetLag can indicate issues with pollers in your function's consumer group. For more information, see [Working with Lambda function metrics \(p. 870\)](#).

Amazon MSK configuration parameters

All Lambda event source types share the same [CreateEventSourceMapping \(p. 1153\)](#) and [UpdateEventSourceMapping \(p. 1356\)](#) API operations. However, only some of the parameters apply to Amazon MSK.

Event source parameters that apply to Amazon MSK

Parameter	Required	Default	Notes
AmazonManagedKafkaEventSourceConfig	N	Contains the ConsumerGroupId field, which defaults to a unique value.	Can set only on Create
BatchSize	N	100	Maximum: 10,000
Enabled	N	Enabled	
EventSourceArn	Y		Can set only on Create
FunctionName	Y		
FilterCriteria	N		Lambda event filtering (p. 136)
MaximumBatchingWindowInSeconds	N	500 ms	Batching behavior (p. 132)

Parameter	Required	Default	Notes
SourceAccessConfiguration	N	No credentials	SASL/SCRAM or CLIENT_CERTIFICATE_TLS_AUTH (MutualTLS) authentication credentials for your event source
StartingPosition	Y		TRIM_HORIZON or LATEST Can set only on Create
Topics	Y		Kafka topic name Can set only on Create

Using AWS Lambda with Amazon RDS

You can use AWS Lambda to process event notifications from an Amazon Relational Database Service (Amazon RDS) database. Amazon RDS sends notifications to an Amazon Simple Notification Service (Amazon SNS) topic, which you can configure to invoke a Lambda function. Amazon SNS wraps the message from Amazon RDS in its own event document and sends it to your function.

Example Amazon RDS message in an Amazon SNS event

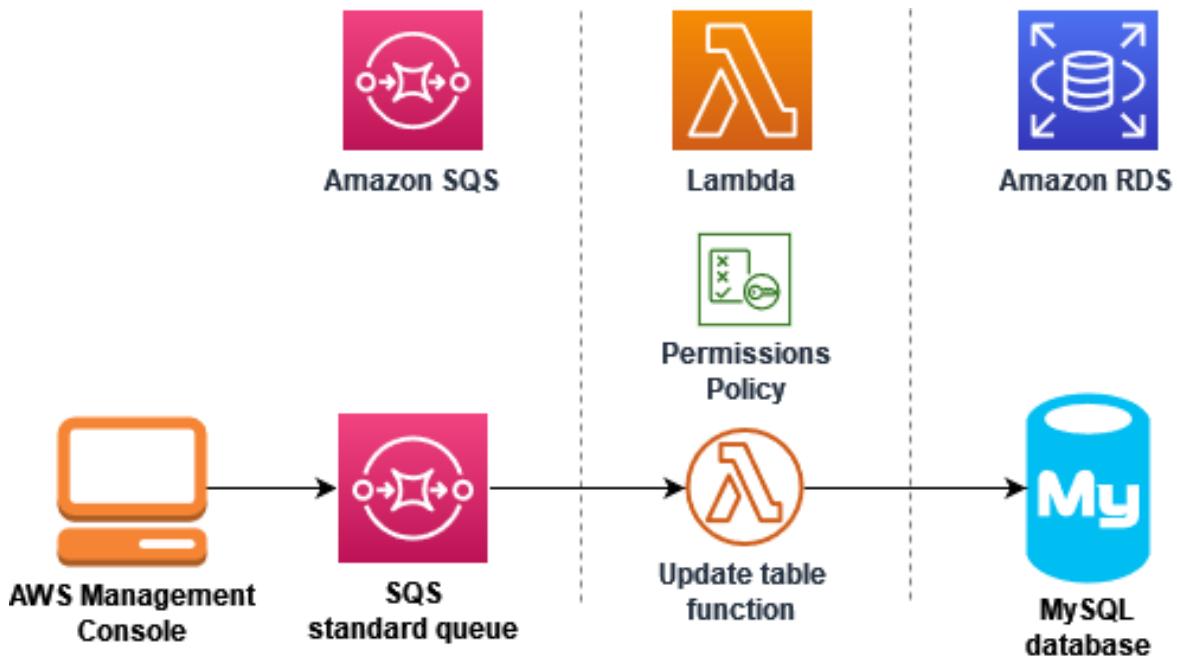
```
{  
    "Records": [  
        {  
            "EventVersion": "1.0",  
            "EventSubscriptionArn": "arn:aws:sns:us-east-2:123456789012:rds-lambda:21be56ed-a058-49f5-8c98-aedd2564c486",  
            "EventSource": "aws:sns",  
            "Sns": {  
                "SignatureVersion": "1",  
                "Timestamp": "2019-01-02T12:45:07.000Z",  
                "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\":\"2019-01-02  
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.amazonaws.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&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"  
            }  
        }  
    ]  
}
```

Topics

- [Tutorial: Using a Lambda function to access Amazon RDS in an Amazon VPC \(p. 728\)](#)
- [Configuring the function \(p. 740\)](#)

Tutorial: Using a Lambda function to access Amazon RDS in an Amazon VPC

In this tutorial, you use a Lambda function to write data to an [Amazon Relational Database Service](#) (Amazon RDS) database. Your Lambda function reads records from an Amazon Simple Queue Service (Amazon SQS) queue and writes a new item to a table in your database whenever a message is added. In this example, you use the AWS Management Console to manually add messages to your queue. The following diagram shows the AWS resources you use to complete the tutorial.



With Amazon RDS, you can run a managed relational database in the cloud using common database products like Microsoft SQL Server, MySQL, and PostgreSQL. By using Lambda to access your database, you can read and write data in response to events, such as a new customer registering with your website. Your function and database instance also scale automatically to meet periods of high demand.

To complete this tutorial, you carry out the following tasks:

1. Launch an Amazon RDS MySQL database instance in your AWS account's default Amazon Virtual Private Cloud (Amazon VPC).
2. Create and test a Lambda function that creates a new table in your database and writes data to it.
3. Create an Amazon SQS queue and configure it to invoke your Lambda function whenever a new message is added.
4. Test the complete set-up by adding messages to your queue using the AWS Management Console and monitoring the results using CloudWatch Logs.

By completing these steps, you learn:

- How to use Lambda to open a connection to an Amazon RDS database instance
- How to use Lambda to perform create and read operations on an Amazon RDS database
- How to use Amazon SQS to invoke a Lambda function

You can complete this tutorial using the AWS Management Console or the AWS Command Line Interface (AWS CLI).

Prerequisites

[Sign up for an AWS account](#)

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, [assign administrative access to an administrative user](#), and use only the root user to perform [tasks that require root user access](#).

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to <https://aws.amazon.com/> and choosing **My Account**.

Create an administrative user

After you sign up for an AWS account, create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the [AWS Management Console](#) as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

For help signing in by using root user, see [Signing in as the root user](#) in the *AWS Sign-In User Guide*.
2. Turn on multi-factor authentication (MFA) for your root user.

For instructions, see [Enable a virtual MFA device for your AWS account root user \(console\)](#) in the *IAM User Guide*.

Create an administrative user

- For your daily administrative tasks, grant administrative access to an administrative user in AWS IAM Identity Center (successor to AWS Single Sign-On).

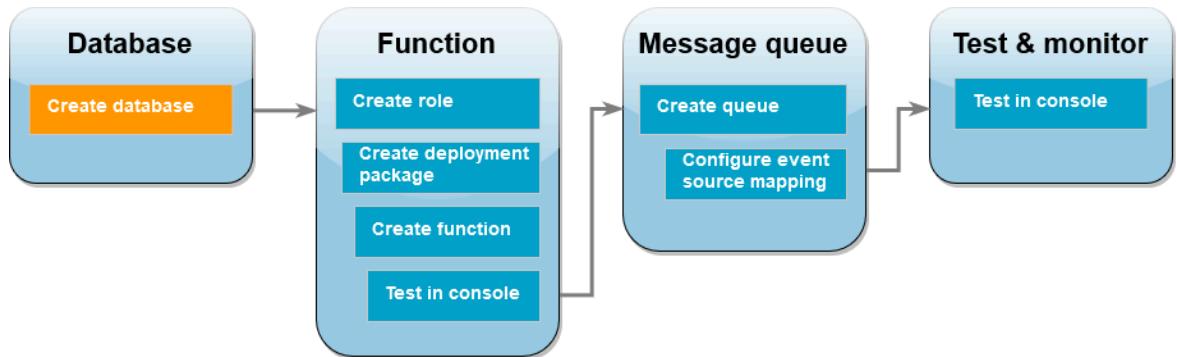
For instructions, see [Getting started](#) in the *AWS IAM Identity Center (successor to AWS Single Sign-On) User Guide*.

Sign in as the administrative user

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

For help signing in using an IAM Identity Center user, see [Signing in to the AWS access portal](#) in the *AWS Sign-In User Guide*.

Create an Amazon RDS database instance



An Amazon RDS database instance is an isolated database environment running in the AWS Cloud. An instance can contain one or more user-created databases. Unless you specify otherwise, Amazon RDS creates new database instances in the default Amazon VPC included in your AWS account. For more information about Amazon VPCs, see the [Amazon Virtual Private Cloud User Guide](#).

In this tutorial, you create a new instance in your AWS account's default VPC and create a database named ExampleDB in that instance. You can create your Amazon RDS instance and database using either the AWS Management Console or the AWS CLI.

AWS Management Console

To create a database instance and database (console)

1. Open the [Databases page](#) of the Amazon RDS console and choose **Create database**.
2. Leave the **Standard create** option selected, then in **Engine options**, choose **MySQL**.
3. In **Templates**, choose **Free tier**.
4. In **Settings**, for **DB instance identifier**, enter **MySQLForLambda**.
5. Set your username and password by doing the following:
 - a. In **Credentials settings**, leave **Master username** set to **admin**
 - b. For **Master password**, enter and confirm a password to access your database.
6. Specify the database name by doing the following:
 - a. Leave all the remaining default options selected and scroll down to the **Additional configuration** pane.
 - b. Expand this pane and enter **ExampleDB** as the **Initial database name**.
7. Leave all the remaining default options selected and choose **Create database**.

AWS CLI

To create a database instance and database (CLI)

- To create your database instance and database using the AWS CLI, choose your own password and run the following command. Lambda uses this password to access the database.

```
aws rds create-db-instance --db-name ExampleDB --engine MySQL \
--db-instance-identifier MySQLForLambda \
--db-instance-class db.t2.micro --allocated-storage 5 --no-publicly-accessible \
--master-username admin --master-user-password password
```

For Lambda to connect to your database, you need to know its host address (endpoint) and VPC configuration.

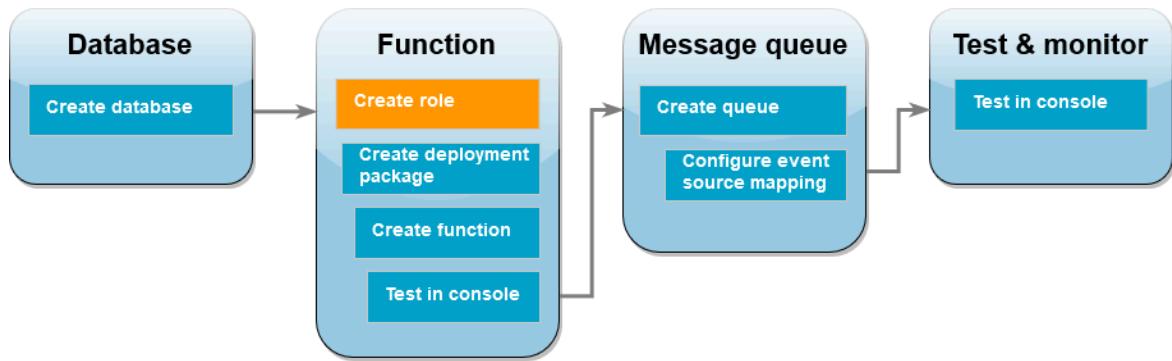
To find the database host address and VPC configuration

1. Open the [Databases page](#) of the Amazon RDS console.
2. Choose the database instance you just created (mysqlforlambda), and select the **Connectivity and security** pane.
3. Record the values of the endpoint, VPC subnets, and VPC security group.

Note

The host endpoint will not be available until your database instance has finished initializing and the status displayed in the console has changed from **Creating** to **Backing up**. This can take several minutes.

Create a function execution role



Before you create your Lambda function, you create an execution role to give your function the necessary permissions. For this tutorial, Lambda needs permission to manage the network connection to the Amazon VPC containing your database instance and to poll messages from an Amazon SQS queue.

To give your Lambda function the permissions it needs, this tutorial uses IAM managed policies. These are policies that grant permissions for many common use cases and are available in your AWS account. For more information about using managed policies, see [Policy best practices \(p. 853\)](#).

To create the Lambda execution role

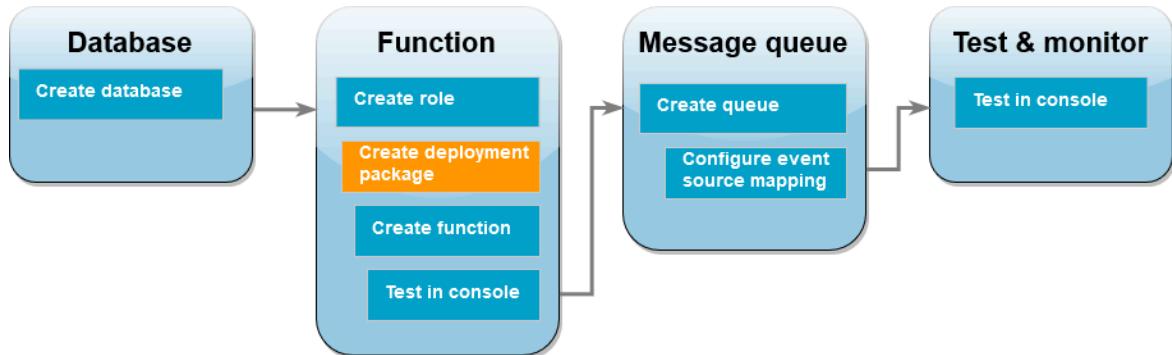
1. Open the [Roles page](#) of the IAM console and choose **Create role**.
2. For the **Trusted entity type**, choose AWS Service, and for the **Use case**, choose Lambda.
3. Choose **Next**.
4. Add the IAM managed policies by doing the following:
 - a. Using the policy search box, search for **AWSLambdaVPCAccessExecutionRole**.
 - b. In the results list, select the check box next to the role, then choose **Clear filters**.
 - c. Using the policy search box, search for **AWSLambdaSQSQueueExecutionRole**.
 - d. In the results list, select the check box next to the role, then choose **Next**.
5. For the **Role name**, enter **lambda-vpc-sqs-role**, then choose **Create role**.

Later in the tutorial, you need the Amazon Resource Name (ARN) of the execution role you just created.

To find the execution role ARN

1. Open the [Roles](#) page of the IAM console and choose your role (`lambda-vpc-sqs-role`)
2. Copy the **Role ARN** displayed in the **Summary** section.

Create a Lambda deployment package



The following example Python code uses the [PyMySQL](#) package to open a connection to your database. The first time you invoke your function, it also creates a new table called `Customer`. The table uses the following schema, where `CustID` is the primary key:

```
Customer(CustID, Name)
```

The function also uses PyMySQL to add records to this table. The function adds records using customer IDs and names specified in messages you will add to your Amazon SQS queue.

Note that the code creates the connection to your database outside of the handler function. Creating the connection in the initialization code allows the connection to be re-used by subsequent invocations of your function and improves performance. In a production application, you can also use [provisioned concurrency](#) to initialize a requested number of database connections. These connections are available as soon as your function is invoked.

```

import sys
import logging
import pymysql
import json

# rds settings
rds_host = "mysqlforlambda.cdipnbm2csku.us-west-2.rds.amazonaws.com"
user_name = "admin"
password = "password"
db_name = "ExampleDB"

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

# create the database connection outside of the handler to allow connections to be
# re-used by subsequent function invocations.
try:
    conn = pymysql.connect(host=rds_host, user=user_name, passwd=password, db=db_name,
                           connect_timeout=5)
except pymysql.MySQLError as e:
    logger.error("ERROR: Unexpected error: Could not connect to MySQL instance.")
    logger.error(e)
    sys.exit()
  
```

```
logger.info("SUCCESS: Connection to RDS MySQL instance succeeded")

def lambda_handler(event, context):
    """
    This function creates a new RDS database table and writes records to it
    """
    message = event['Records'][0]['body']
    data = json.loads(message)
    CustID = data['CustID']
    Name = data['Name']

    item_count = 0
    sql_string = f"insert into Customer (CustID, Name) values({CustID}, '{Name}')"

    with conn.cursor() as cur:
        cur.execute("create table if not exists Customer ( CustID  int NOT NULL, Name
varchar(255) NOT NULL, PRIMARY KEY (CustID))")
        cur.execute(sql_string)
        conn.commit()
        cur.execute("select * from Customer")
        logger.info("The following items have been added to the database:")
        for row in cur:
            item_count += 1
            logger.info(row)
        conn.commit()

    return "Added %d items to RDS MySQL table" %(item_count)
```

Note

In this example code, your database name, username, and password are hardcoded into your function. In a production application, you should not hardcode these parameters. Use [AWS Secrets Manager](#) to securely store database access credentials.

To include the PyMySQL dependency with your function code, create a .zip deployment package.

To create a .zip deployment package

1. Save the example code as a file named `lambda_function.py`. Use your own Amazon RDS host endpoint and replace the password in the example code with the password you chose when you created your database instance.
2. In the same directory in which you created your `lambda_function.py` file, create a new directory named `package` and install the PyMySQL library.

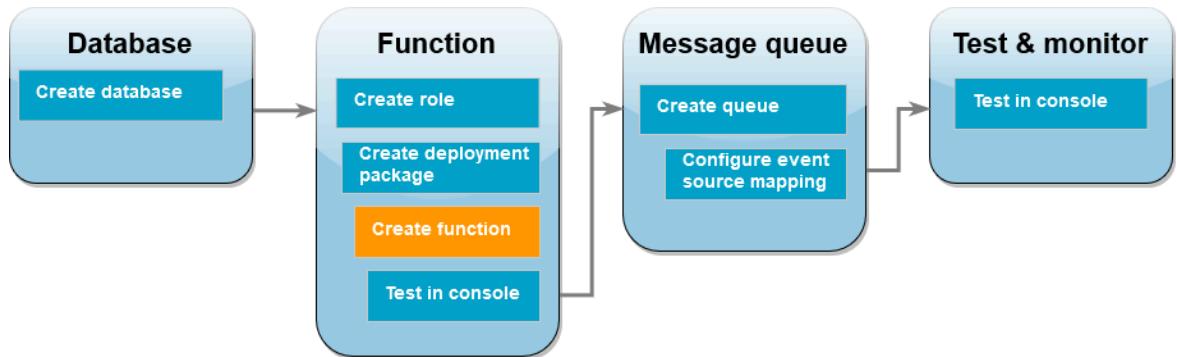
```
mkdir package
pip install --target package pymysql
```

3. Create a zip file containing your application code and the PyMySQL library. In Linux or MacOS, run the following CLI commands. In Windows, use your preferred zip tool to create the `lambda_function.zip` file.

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

You can also create your deployment package using a Python virtual environment. See [Deploy Python Lambda functions with .zip file archives](#).

Create the Lambda function



Using the .zip package you just created, you now create a Lambda function using either the AWS CLI or the Lambda console.

AWS Management Console

To create the function (console)

1. Open the [Functions](#) page of the Lambda console and choose **Create function**.
2. Leave **Author from scratch** selected, and in **Basic information**, enter **LambdaFunctionWithRDS** for the function name.
3. Select Python3.9 as the runtime.
4. Choose **Create function**.
5. In the **Code** pane, choose **Upload from** and then **.zip file**.
6. Select the `lambda_function.zip` file you created in the previous stage and choose **Save**.

Now configure the function with the execution role you created earlier and your VPC settings. This grants the function the permissions it needs to access your database instance and poll an Amazon SQS queue.

To configure the function

1. In the [Functions](#) page of the Lambda console, select the **Configuration** tab, then choose **Permissions**.
2. In the **Execution role** pane, choose **Edit**.
3. In **Existing role**, choose your execution role (`lambda-vpc-sqs-role`) from the dropdown list.
4. Choose **Save**.
5. Configure the VPC settings by doing the following:
 - a. In the **Configuration** tab, choose **VPC**, then select **Edit**.
 - b. In **VPC**, choose your AWS account's default VPC from the dropdown list.
 - c. In **Subnets**, select the checkboxes for the VPC subnets you noted earlier.
 - d. In **Security groups**, select the checkboxes for the VPC security groups you noted earlier, then choose **Save**.

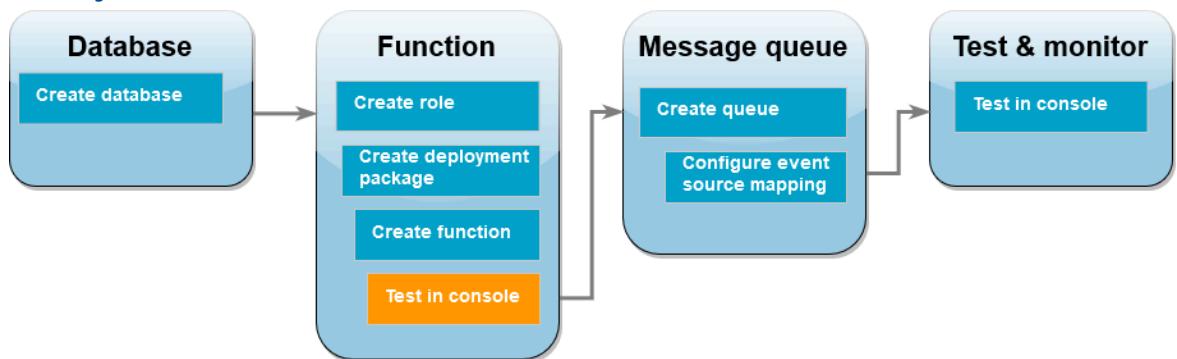
AWS CLI

To create the function (CLI)

- Run the following command with your own role ARN, VPC subnet IDs and VPC security group values.

```
aws lambda create-function --function-name LambdaFunctionWithRDS --runtime
    python3.9 \
    --zip-file fileb://lambda_function.zip --handler lambda_function.lambda_handler \
    --role arn:aws:iam::111122223333:role/lambda-vpc-sqs-role \
    --vpc-config SubnetIds=subnet-1234567890abcdef0,subnet-abcdef01234567890, \
    subnet-021345abcdef6789,subnet-1234abcdef567890,SecurityGroupIds=sg-1234567890abcdef0
```

Test your Lambda function in the console



You can now use the Lambda console to test your function. You create a test event which mimics the data your function will receive when you invoke it using Amazon SQS in the final stage of the tutorial. Your test event contains a JSON object specifying a customer ID and customer name to add to the Customer table your function creates.

To test the Lambda function

- Open the [Functions](#) page of the Lambda console and choose your function.
- Choose the **Code** tab.
- In the **Code source** pane, choose **Test** and enter **myTestEvent** for the event name.
- Copy the following code into **Event JSON** and choose **Save**.

```
{
  "Records": [
    {
      "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
      "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCxlaS3SLy0a...",
      "body": "{\n        \"CustID\": 1021,\n        \"Name\": \"Martha Rivera\"\n    }",
      "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIEQZJ0L023YVJ4V0",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
      },
      "messageAttributes": {},
      "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
      "eventSource": "aws:sqs",
      "eventSourceARN": "arn:aws:sqs:us-west-2:123456789012:my-queue",
      "awsRegion": "us-west-2"
    }
  ]
}
```

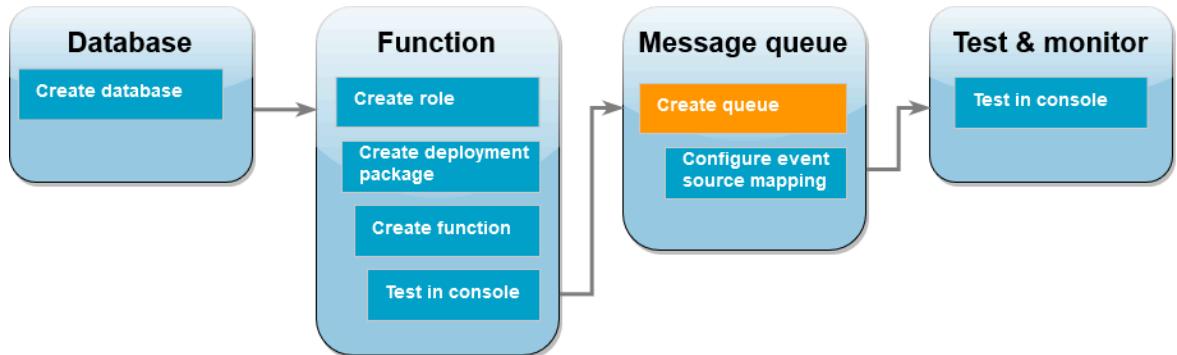
```
}
```

5. Choose **Test**.

In the **Execution results** tab, you should see results similar to the following displayed in the Function Logs:

```
[INFO] 2023-02-14T19:31:35.149Z bdd06682-00c7-4d6f-9abb-89f4bbb4a27f The following items have been added to the database:  
[INFO] 2023-02-14T19:31:35.149Z bdd06682-00c7-4d6f-9abb-89f4bbb4a27f (1021, 'Martha Rivera')
```

Create an Amazon SQS queue



You have successfully tested the integration of your Lambda function and Amazon RDS database instance. Now you create the Amazon SQS queue you will use to invoke your Lambda function in the final stage of the tutorial.

AWS Management Console

To create the Amazon SQS queue (console)

1. Open the [Queues](#) page of the Amazon SQS console and select **Create queue**.
2. Leave the **Type** as **Standard** and enter **LambdaRDSQueue** for the name of your queue.
3. Leave all the default options selected and choose **Create queue**.

AWS CLI

To create the Amazon SQS queue (CLI)

- Run the following command and record the URL the AWS CLI returns:

```
aws sqs create-queue --queue-name LambdaRDSQueue
```

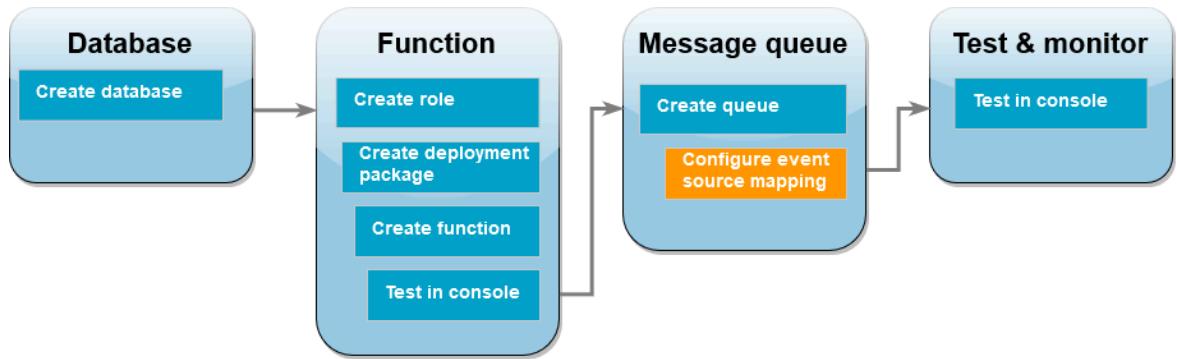
In the next part of the tutorial, you will need the Amazon Resource Name (ARN) of the queue you just created.

To find the queue ARN

- Using the URL you recorded in the previous step, run the following command and record the ARN the AWS CLI returns:

```
aws sqs get-queue-attributes \
--queue-url https://sqs.us-west-2.amazonaws.com/111122223333/LambdaRDSQueue \
--attribute-names QueueArn
```

Create an event source mapping to invoke your Lambda function



An [event source mapping](#) is a Lambda resource which reads items from a stream or queue and invokes a Lambda function. When you configure an event source mapping, you can specify a batch size so that records from your stream or queue are batched together into a single payload. In this example, you set the batch size to 1 so that your Lambda function is invoked every time you send a message to your queue. You can configure the event source mapping using either the AWS CLI or the Lambda console.

AWS Management Console

To create an event source mapping (console)

1. Open the [Functions](#) page of the Lambda console and select your function (LambdaFunctionWithRDS).
2. In the **Function overview** pane, choose **Add trigger**.
3. For the source, select Amazon SQS, then select the name of your queue (LambdaRDSQueue).
4. For **Batch size**, enter **1**.
5. Leave all the other options set to the default values and choose **Add**.

AWS CLI

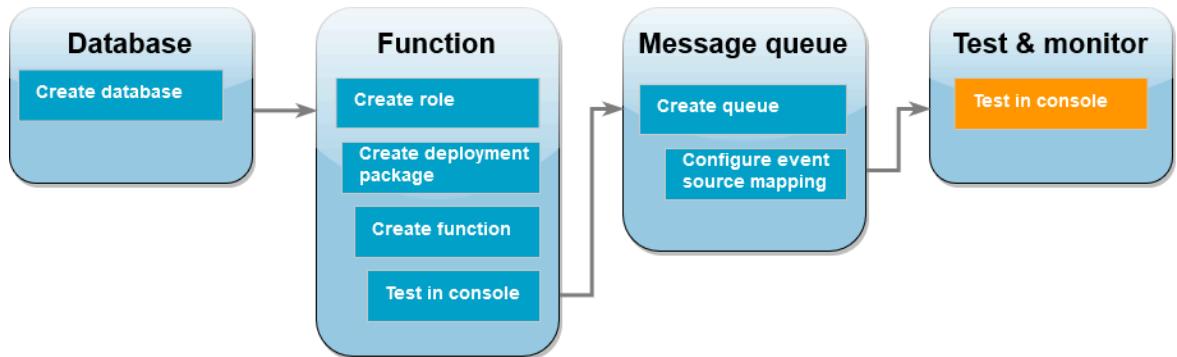
To create an event source mapping (CLI)

- To create an event source mapping using the AWS CLI, run the following command using the ARN for your own Amazon SQS queue:

```
aws lambda create-event-source-mapping --function-name LambdaFunctionWithRDS --batch-size 1 \
--event-source-arn arn:aws:sqs:us-west-2:111122223333:LambdaRDSQueue
```

You are now ready to test your complete setup by adding a message to your Amazon SQS queue.

Test and monitor your setup



To test your complete setup, add messages to your Amazon SQS queue using the console. You then use CloudWatch Logs to confirm that your Lambda function is writing records to your database as expected.

To test and monitor your setup

1. Open the [Queues](#) page of the Amazon SQS console and select your queue (LambdaRDSQueue).
2. Choose **Send and receive messages** and paste the following JSON into the **Message body** in the **Send message** pane.

```
{
    "CustID": 1054,
    "Name": "Richard Roe"
}
```

3. Choose **Send message**.

Sending your message to the queue will cause Lambda to invoke your function through your event source mapping. To confirm that Lambda has invoked your function as expected, use CloudWatch Logs to verify that the function has written the customer name and ID to your database table:

4. Open the [Log groups](#) page of the CloudWatch console and select the log group for your function (aws/lambda/LambdaFunctionWithRDS).
5. In the **Log streams** pane, choose the most recent log stream.

Your table should contain two customer records, one from each invocation of your function. In the log stream, you should see messages similar to the following:

```
[INFO] 2023-02-14T19:06:43.873Z 45368126-3eee-47f7-88ca-3086ae6d3a77 The following items have been added to the database:
[INFO] 2023-02-14T19:06:43.873Z 45368126-3eee-47f7-88ca-3086ae6d3a77 (1021, 'Martha Rivera')
[INFO] 2023-02-14T19:06:43.873Z 45368126-3eee-47f7-88ca-3086ae6d3a77 (1054, 'Richard Roe')
```

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.

2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

To delete the execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the MySQL DB instance

1. Open the [Databases page](#) of the Amazon RDS console.
2. Select the database you created.
3. Choose **Actions, Delete**.
4. Clear the **Create final snapshot** check box.
5. Enter **delete me** in the text box.
6. Choose **Delete**.

To delete the Amazon SQS queue

1. Sign in to the AWS Management Console and open the Amazon SQS console at <https://console.aws.amazon.com/sqs/>.
2. Select the queue you created.
3. Choose **Delete**.
4. Enter **delete** in the text input field.
5. Choose **Delete**.

Configuring the function

The following section shows additional configurations and topics we recommend as part of this tutorial.

- If too many function instances run concurrently, one or more instances may fail to obtain a database connection. You can use reserved concurrency to limit the maximum concurrency of the function. Set the reserved concurrency to be less than the number of database connections. Reserved concurrency also reserves those instances for this function, which may not be ideal. If you are invoking the Lambda functions from your application, we recommend you write code that limits the number of concurrent instances. For more information, see [Managing concurrency for a Lambda function](#).
- For more information on configuring an Amazon RDS database to send notifications, see [Using Amazon RDS event notifications](#).
- For more information on using Amazon SNS as trigger, see [Using AWS Lambda with Amazon SNS \(p. 770\)](#).

Using AWS Lambda with Amazon S3

You can use Lambda to process [event notifications](#) from Amazon Simple Storage Service. Amazon S3 can send an event to a Lambda function when an object is created or deleted. You configure notification settings on a bucket, and grant Amazon S3 permission to invoke a function on the function's resource-based permissions policy.

Warning

If your Lambda function uses the same bucket that triggers it, it could cause the function to run in a loop. For example, if the bucket triggers a function each time an object is uploaded, and the function uploads an object to the bucket, then the function indirectly triggers itself. To avoid this, use two buckets, or configure the trigger to only apply to a prefix used for incoming objects.

Amazon S3 invokes your function [asynchronously \(p. 123\)](#) with an event that contains details about the object. The following example shows an event that Amazon S3 sent when a deployment package was uploaded to Amazon S3.

Example Amazon S3 notification event

```
{  
  "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":  
        "v1R7PnpV2Ce81l0PRw6jlUpck7Jo5ZsQjryTjKlc5aLWGVHPZLj5NeC6qMa0emYBDX0o6QBU0Wo=",  
      },  
      "s3": {  
        "s3SchemaVersion": "1.0",  
        "configurationId": "828aa6fc-f7b5-4305-8584-487c791949c1",  
        "bucket": {  
          "name": "DOC-EXAMPLE-BUCKET",  
          "ownerIdentity": {  
            "principalId": "A3I5XTEXAMAI3E"  
          },  
          "arn": "arn:aws:s3:::lambda-artifacts-deafc19498e3f2df"  
        },  
        "object": {  
          "key": "b21b84d653bb07b05b1e6b33684dc11b",  
          "size": 1305107,  
          "eTag": "b21b84d653bb07b05b1e6b33684dc11b",  
          "sequencer": "0C0F6F405D6ED209E1"  
        }  
      }  
    }  
  ]  
}
```

To invoke your function, Amazon S3 needs permission from the function's [resource-based policy \(p. 832\)](#). When you configure an Amazon S3 trigger in the Lambda console, the console modifies

the resource-based policy to allow Amazon S3 to invoke the function if the bucket name and account ID match. If you configure the notification in Amazon S3, you use the Lambda API to update the policy. You can also use the Lambda API to grant permission to another account, or restrict permission to a designated alias.

If your function uses the AWS SDK to manage Amazon S3 resources, it also needs Amazon S3 permissions in its [execution role \(p. 816\)](#).

Topics

- [Tutorial: Using an Amazon S3 trigger to invoke a Lambda function \(p. 742\)](#)
- [Tutorial: Using an Amazon S3 trigger to create thumbnail images \(p. 749\)](#)

Tutorial: Using an Amazon S3 trigger to invoke a Lambda function

In this tutorial, you use the console to create a Lambda function and configure a trigger for Amazon Simple Storage Service (Amazon S3). The trigger invokes your function every time that you add an object to your Amazon S3 bucket.

The Lambda function in this tutorial uses the Node.js runtime. For code samples that use other runtimes, see [Amazon S3 trigger to invoke a Lambda function](#).

Prerequisites

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, [assign administrative access to an administrative user](#), and use only the root user to perform [tasks that require root user access](#).

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to <https://aws.amazon.com/> and choosing **My Account**.

Create an administrative user

After you sign up for an AWS account, create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the [AWS Management Console](#) as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

For help signing in by using root user, see [Signing in as the root user](#) in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

For instructions, see [Enable a virtual MFA device for your AWS account root user \(console\)](#) in the *IAM User Guide*.

Create an administrative user

- For your daily administrative tasks, grant administrative access to an administrative user in AWS IAM Identity Center (successor to AWS Single Sign-On).

For instructions, see [Getting started](#) in the *AWS IAM Identity Center (successor to AWS Single Sign-On) User Guide*.

Sign in as the administrative user

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

For help signing in using an IAM Identity Center user, see [Signing in to the AWS access portal](#) in the *AWS Sign-In User Guide*.

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console \(p. 4\)](#) to create your first Lambda function.

Create an Amazon S3 bucket and upload a sample object

Follow these steps to create an Amazon S3 bucket and upload an object.

1. Open the [Amazon S3 console](#).
2. Choose **Create bucket**.
3. Under **General configuration**, do the following:
 - a. For **Bucket name**, enter a unique name.
 - b. For **AWS Region**, choose a Region. Note that you must create your Lambda function in the same Region.
4. Choose **Create bucket**.
5. [Upload an object](#) to the bucket (for example, HappyFace.jpg).

You must create this sample object before you test your Lambda function. When you [test the function manually \(p. 746\)](#) later in this tutorial, you pass sample event data to the function that specifies the bucket name and object name.

Create the IAM execution role

Your Lambda function needs permissions to access objects in Amazon S3. Follow these steps to create an execution role that grants your function full Amazon S3 permissions.

To create the execution role

1. Open the [Roles page](#) in the IAM console.
2. Choose **Create role**.
3. On the first page, make the following selections:

- Trusted entity type – AWS service
- Use case – Lambda

Choose **Next**.

4. On the **Add permissions** page, type s3 in the search box. Select the checkbox next to the AmazonS3FullAccess managed AWS policy.
5. Still on the **Add permissions** page, remove the previous s3 search filter and type basic in the search box. Select the checkbox next to the AWSLambdaBasicExecutionRole managed AWS policy.
6. Choose **Next**.
7. On the **Name, review, and create** page, enter my-s3-function-role. Choose **Create role**.

Create the Lambda function

Follow these steps to create the Lambda function.

To create the Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Choose **Create function**.
3. Choose **Author from scratch**.
4. Under **Basic information**, do the following:
 - a. For **Function name**, enter **my-s3-function**.
 - b. In this tutorial, we'll use the Node.js runtime. For **Runtime**, choose **Node.js 16.x**.
 - c. For **Architecture**, enter **x86_64**.
 - d. Expand the **Change default execution role** tab. Choose **Use an existing role**, then choose the my-s3-function-role you created earlier.
5. Choose **Create function**.

Create the Amazon S3 trigger

Follow these steps to set up an Amazon S3 trigger to your Lambda function.

To create the Amazon S3 trigger

1. In the [Functions page](#) of the Lambda console, choose the function (my-s3-function) that you created earlier.
2. Choose **Add trigger**.
3. Choose **S3** as the source.
4. For **Bucket**, choose the bucket you created earlier. Keep the other default settings.
5. Acknowledge the **Recursive invocation** warning.
6. Choose **Add**.

Write the function code

Your Lambda function will retrieve the source S3 bucket name and the key name of the uploaded object from the event parameter that it receives. The function uses the Amazon S3 getObject API to retrieve the content type of the object.

While viewing your function in the [Lambda console](#), you can view the function code on the **Code** tab, under **Code source**. The code looks like the following:

Node.js

Example index.js

```
console.log('Loading function');

const aws = require('aws-sdk');

const s3 = new aws.S3({ apiVersion: '2006-03-01' });

exports.handler = async (event, context) => {
    //console.log('Received event:', JSON.stringify(event, null, 2));

    // 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.getObject(params).promise();
        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);
    }
};
```

Python

For a Python example, see [Amazon S3 trigger to invoke a Lambda function](#) on the Serverless Land website.

Java

For a Java example, see [Amazon S3 trigger to invoke a Lambda function](#) on the Serverless Land website.

.NET

For a .NET example, see [Amazon S3 trigger to invoke a Lambda function](#) on the Serverless Land website.

Go

For a Go example, see [Amazon S3 trigger to invoke a Lambda function](#) on the Serverless Land website.

Ruby

For a Ruby example, see [Amazon S3 trigger to invoke a Lambda function](#) on the Serverless Land website.

TypeScript

For a TypeScript example, see [Amazon S3 trigger to invoke a Lambda function](#) on the Serverless Land website.

Rust

For a Rust example, [Amazon S3 trigger to invoke a Lambda function](#) on the Serverless Land website.

Other languages

For examples in other languages, see [Amazon S3 trigger to invoke a Lambda function](#) on the Serverless Land website.

Copy paste the Javascript code in the Node.js tab into the code editor for your function, and then choose **Deploy**. The tablist also contains sample code for other runtimes that Lambda supports.

Test in the console

Invoke the Lambda function manually using sample Amazon S3 event data.

To test the Lambda function using the console

1. For your function, in the **Code** tab, under **Code source**, choose the arrow next to **Test**, and then choose **Configure test event** from the dropdown list.
2. In the **Configure test event** window, do the following:
 - a. Choose **Create new test event**.
 - b. For **Event name**, enter a name for the test event. For example, **mys3testevent**.
 - c. For **Event sharing settings**, choose **Private**.
 - d. For **Template**, choose **S3 Put (s3-put)**.
 - e. In the **Event JSON**, replace the following values:
 - **us-east-1** – The AWS Region where you created the Amazon S3 bucket and the Lambda function.
 - **example-bucket** – The Amazon S3 bucket that you created earlier.
 - **test%2Fkey** – The name of the sample object that you uploaded to the bucket (for example, **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": "example-bucket",  
          "arn": "arn:aws:s3:::example-bucket"  
        },  
        "object": {  
          "key": "test%2Fkey",  
          "size": 1024,  
          "type": "Image/JPEG",  
          "eTag": "d41d8cd98f00b204e9800998ecf8427",  
          "sequencer": "0A19E16649B0DCE1"  
        }  
      }  
    }  
  ]  
}
```

```
        "name": "example-bucket",
        "ownerIdentity": {
            "principalId": "EXAMPLE"
        },
        "arn": "arn:aws:s3:::example-bucket"
    },
    "object": {
        "key": "test%2Fkey",
        "size": 1024,
        "eTag": "0123456789abcdef0123456789abcdef",
        "sequencer": "0A1B2C3D4E5F678901"
    }
}
]
```

- f. Choose **Save**.
3. To invoke the function with your test event, under **Code source**, choose **Test**.

The **Execution results** tab displays the response, function logs, and request ID, similar to the following:

```
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: 'my-s3-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

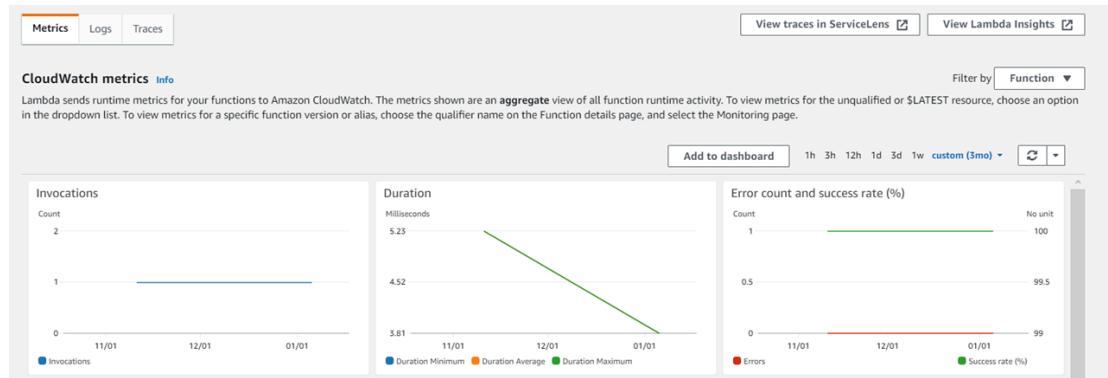
Request ID
12b3cae7-5f4e-415e-93e6-416b8f8b66e6
```

Test the S3 trigger

Invoke your function when you upload a file to the Amazon S3 source bucket.

To test the Lambda function using the S3 trigger

1. On the [Buckets page](#) of the Amazon S3 console, choose the name of the source bucket that you created earlier.
2. On the [Upload](#) page, upload more .jpg or .png image files to the bucket.
3. Open the [Functions page](#) of the Lambda console.
4. Choose the name of your function (**my-s3-function**).
5. To verify that the function ran once for each file that you uploaded, choose the **Monitor** tab. This page shows graphs for the metrics that Lambda sends to CloudWatch. The count in the **Invocations** graph should match the number of files that you uploaded to the Amazon S3 bucket.



For more information on these graphs, see [Monitoring functions on the Lambda console \(p. 861\)](#).

6. (Optional) To view the logs in the CloudWatch console, choose **View logs in CloudWatch**. Choose a log stream to view the logs output for one of the function invocations.

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

To delete the execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the S3 bucket

1. Open the [Amazon S3 console](#).
2. Select the bucket you created.
3. Choose **Delete**.
4. Enter the name of the bucket in the text input field.
5. Choose **Delete bucket**.

Next steps

Try the more advanced tutorial. In this tutorial, the S3 trigger invokes a function to [create a thumbnail image \(p. 749\)](#) for each image file that is uploaded to your S3 bucket. This tutorial requires a moderate

level of AWS and Lambda domain knowledge. You use the AWS Command Line Interface (AWS CLI) to create resources, and you create a .zip file archive deployment package for your function and its dependencies.

Tutorial: Using an Amazon S3 trigger to create thumbnail images

In this tutorial, you create a Lambda function and configure a trigger for Amazon Simple Storage Service (Amazon S3). Amazon S3 invokes the `CreateThumbnail` function for each image file that is uploaded to an S3 bucket. The function reads the image object from the source S3 bucket and creates a thumbnail image to save in a target S3 bucket.

Note

This tutorial requires a moderate level of knowledge about AWS and Lambda. We recommend that you first try [Tutorial: Using an Amazon S3 trigger to invoke a Lambda function \(p. 742\)](#).

In this tutorial, you use the AWS Command Line Interface (AWS CLI) to create the following AWS resources:

Lambda resources

- A Lambda function. You can choose Node.js, Python, or Java for the function code.
- A .zip file archive deployment package for the function.
- An access policy that grants Amazon S3 permission to invoke the function.

AWS Identity and Access Management (IAM) resources

- An execution role with an associated permissions policy to grant permissions that your function needs.

Amazon S3 resources

- A source S3 bucket with a notification configuration that invokes the function.
- A target S3 bucket where the function saves the resized images.

Topics

- [Prerequisites \(p. 749\)](#)
- [Step 1. Create S3 buckets and upload a sample object \(p. 750\)](#)
- [Step 2. Create the IAM policy \(p. 750\)](#)
- [Step 3. Create the execution role \(p. 751\)](#)
- [Step 4. Create the deployment package \(p. 751\)](#)
- [Step 5. Create the Lambda function \(p. 759\)](#)
- [Step 6. Test the Lambda function \(p. 760\)](#)
- [Step 7. Configure Amazon S3 to publish events \(p. 761\)](#)
- [Step 8. Test using the Amazon S3 trigger \(p. 762\)](#)
- [Step 9. Clean up your resources \(p. 762\)](#)

Prerequisites

- Install the [AWS Command Line Interface \(AWS CLI\) version 2](#) and configure it with your AWS credentials.

- Install the language support tools and a package manager for the language that you want to use: Node.js, Python, or Java. For suggested tools, see [Authoring and deploying functions \(p. 6\)](#).

Step 1. Create S3 buckets and upload a sample object

Follow these steps to create S3 buckets and upload an object.

1. Open the [Amazon S3 console](#).
2. [Create two S3 buckets](#). The target bucket must be named **source-resized**, where **source** is the name of the source bucket. For example, a source bucket named sourcebucket and a target bucket named sourcebucket-resized.

Note
Make sure that you create the buckets in the same AWS Region that you plan to use for the Lambda function.
3. In the source bucket, [upload](#) a .jpg or .png object, for example, HappyFace.jpg.

You must create this sample object before you test your Lambda function. When you test the function manually in step 6, you pass sample event data to the function that specifies the source bucket name and image file name.

Step 2. Create the IAM policy

Create an IAM policy that defines the permissions for the Lambda function. The function must have permissions to:

- Get the object from the source S3 bucket.
- Put the resized object into the target S3 bucket.
- Write logs to Amazon CloudWatch Logs.

To create an IAM policy

1. Open the [Policies page](#) in the IAM console.
2. Choose **Create policy**.
3. Choose the **JSON** tab, and then paste the following policy. Be sure to replace **sourcebucket** with the name of the source bucket that you created previously.

```
{  
    "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:::sourcebucket/*"  
        }  
    ]  
}
```

```
        },
        {
          "Effect": "Allow",
          "Action": [
            "s3:PutObject"
          ],
          "Resource": "arn:aws:s3:::sourcebucket-resized/*"
        }
      ]
    }
```

4. Choose **Next: Tags**.
5. Choose **Next: Review**.
6. Under **Review policy**, for **Name**, enter **AWSLambdaS3Policy**.
7. Choose **Create policy**.

Step 3. Create the execution role

Create the [execution role \(p. 816\)](#) that gives your Lambda function permission to access AWS resources.

To create an execution role

1. Open the [Roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties:
 - Trusted entity – Lambda
 - Permissions policy – **AWSLambdaS3Policy**
 - Role name – **lambda-s3-role**

Step 4. Create the deployment package

The deployment package is a [zip file archive \(p. 18\)](#) containing your Lambda function code and its dependencies.

Node.js

1. Open a command line terminal or shell in a Linux environment. Ensure that the Node.js version in your local environment matches the Node.js version of your function.
2. Create a directory named `lambda-s3`.

```
mkdir lambda-s3
```

3. Save the function code as `index.js`.

```
// dependencies
const AWS = require('aws-sdk');
const util = require('util');
const sharp = require('sharp');

// get reference to S3 client
const s3 = new AWS.S3();

exports.handler = async (event, context, callback) => {
```

```
// Read options from the event parameter.
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;
}

// Download the image from the S3 source bucket.

try {
  const params = {
    Bucket: srcBucket,
    Key: srcKey
  };
  var origimage = await s3.getObject(params).promise();

} 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 buffer = await sharp(origimage.Body).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: buffer,
    ContentType: "image"
  };

  const putResult = await s3.putObject(destparams).promise();

} catch (error) {
  console.log(error);
  return;
}
```

```
    console.log('Successfully resized ' + srcBucket + '/' + srcKey +  
      ' and uploaded to ' + dstBucket + '/' + dstKey);  
};
```

4. In the `lambda-s3` directory, create a `node_modules` directory.

```
mkdir node_modules  
cd node_modules
```

5. In the `node_modules` directory, install the `sharp` library with `npm`.

```
npm install sharp
```

After this step, you have the following directory structure:

```
lambda-s3  
|- index.js  
|- /node_modules/...  
# /node_modules/sharp
```

6. Return to the `lambda-s3` directory.

```
cd lambda-s3
```

7. Create a deployment package with the function code and its dependencies. Set the `-r` (recursive) option for the `zip` command to compress the subfolders.

```
zip -r function.zip .
```

Python

To create a .zip deployment package

1. Save the example code as a file named `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. In the same directory in which you created your `lambda_function.py` file, create a new directory named `package` and install the [Pillow \(PIL\)](#) library.

```
mkdir package
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python 3.9 \
--only-binary=:all: --upgrade \
pillow
```

The Pillow library contains C/C++ code. By using the `--platform manylinux_2014_x86_64` and `--only-binary=:all:` options, pip will download and install a version of Pillow that contains pre-compiled binaries compatible with the Amazon Linux 2 operating system. This ensures that your deployment package will work in the Lambda execution environment, regardless of the operating system and architecture of your local build machine.

3. Create a `.zip` file containing your application code and the Pillow library. In Linux or MacOS, run the following commands from your command line interface. In Windows, use your preferred zip tool to create the `lambda_function.zip` file.

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

You can also create your deployment package using a Python virtual environment. See [Deploy Python Lambda functions with .zip file archives \(p. 324\)](#)

Java

Prerequisites

Make sure that Java 11 or later and Apache Maven are installed in your development environment.

- For Java, use [Oracle Java SE Development Kit](#), [Amazon Corretto](#), [Red Hat OpenJDK](#), or [AdoptOpenJDK](#).
- For Maven, go to <https://maven.apache.org/>.

To create the project

1. Use the following command to create a new directory called `create-thumbnail` with a project configuration file (`pom.xml`) and a basic Java class.

```
mvn -B archetype:generate \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DgroupId=com.example.CreateThumbnail \
-DartifactId=create-thumbnail
```

2. In the `create-thumbnail` directory that you created in the previous step, open the `pom.xml`. Replace its contents with the following code, and then save your changes.

```
<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.CreateThumbnail</groupId>
<artifactId>create-thumbnail</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<name>create-thumbnail-function</name>
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.release>11</maven.compiler.release>
</properties>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>bom</artifactId>
            <version>2.17.201</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>software.amazon.awssdk</groupId>
        <artifactId>s3</artifactId>
    </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>3.11.0</version>
    </dependency>
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>[2.8.9,)</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.22.2</version>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.3.0</version>
            <configuration>
                <createDependencyReducedPom>false</createDependencyReducedPom>
            </configuration>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <filters>
```

```
<filter>
    <artifact>*</artifact>
    <excludes>
        <exclude>META-INF/*</exclude>
        <exclude>**/module-info.class</exclude>
    </excludes>
</filter>
</filters>
</configuration>
</execution>
</executions>
</plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>3.8.1</version>
<configuration>
    <source>11</source>
    <target>11</target>
</configuration>
</plugin>
</plugins>
</build>
</project>
```

- The dependencyManagement section contains a dependency to the AWS SDK for Java.
 - The dependencies section has a dependency for Amazon S3.
 - The build section includes the [Maven Shade plugin](#). This plugin creates a JAR file that contains the compiled function code and all of its dependencies. The Apache Maven Compiler Plugin is configured to use Java 11.
3. In the /create-thumbnail/src/main/java/com/example/CreateThumbnail directory, change the name of the App.java file to Handler.java.
 4. Open Handler.java and replace its contents with the following code and save the file.

Example Handler.java

```
package com.example.CreateThumbnail;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.image.BufferedImage;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.imageio.ImageIO;

import software.amazon.awssdk.awscore.exception.AwsServiceException;
import software.amazon.awssdk.core.sync.RequestBody;
import software.amazon.awssdk.services.s3.model.GetObjectRequest;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;
import software.amazon.awssdk.services.s3.S3Client;

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.S3Event;
```

```

import com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNotification;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

// Handler value: com.example.CreateThumbnail.Handler
public class Handler implements RequestHandler<S3Event, String> {
    Gson gson = new GsonBuilder().setPrettyPrinting().create();
    private static final float MAX_DIMENSION = 100;
    private final String REGEX = ".*\\.(\\[^\\.]*\\.)";
    private final String JPG_TYPE = "jpg";
    private final String JPG_MIME = "image/jpeg";
    private final String PNG_TYPE = "png";
    private final String PNG_MIME = "image/png";
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        final LambdaLogger logger = context.getLogger();
        try {
            logger.log("EVENT: " + gson.toJson(s3event));
            S3EventNotificationRecord record = s3event.getRecords().get(0);

            String srcBucket = record.getS3().getBucket().getName();

            // Object key may have spaces or unicode non-ASCII characters.
            String srcKey = record.getS3().getObject().getUrlDecodedKey();

            String dstBucket = srcBucket + "-resized";
            String dstKey = "resized-" + srcKey;

            // Infer the image type.
            Matcher matcher = Pattern.compile(REGEX).matcher(srcKey);
            if (!matcher.matches()) {
                logger.log("Unable to infer image type for key " + srcKey);
                return "";
            }
            String imageType = matcher.group(1);
            if (!(JPG_TYPE.equals(imageType)) && !(PNG_TYPE.equals(imageType))) {
                logger.log("Skipping non-image " + srcKey);
                return "";
            }

            // Download the image from S3 into a stream
            S3Client s3Client = S3Client.builder().build();
            InputStream s3Object = getObject(s3Client, srcBucket, srcKey);

            // Read the source image and resize it
            BufferedImage srcImage = ImageIO.read(s3Object);
            BufferedImage newImage = resizeImage(srcImage);

            // Re-encode image to target format
            ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
            ImageIO.write(newImage, imageType, outputStream);

            // Upload new image to S3
            putObject(s3Client, outputStream, dstBucket, dstKey, imageType, logger);

            logger.log("Successfully resized " + srcBucket + "/"
                    + srcKey + " and uploaded to " + dstBucket + "/" + dstKey);
            return "Ok";
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    private InputStream getObject(S3Client s3Client, String bucket, String key) {

```

```

GetObjectRequest get0bjectRequest = GetObjectRequest.builder()
    .bucket(bucket)
    .key(key)
    .build();
return s3Client.getObject(get0bjectRequest);
}

private void putObject(S3Client s3Client, ByteArrayOutputStream outputStream,
    String bucket, String key, String imageType, LambdaLogger logger) {
    Map<String, String> metadata = new HashMap<>();
    metadata.put("Content-Length", Integer.toString(outputStream.size()));
    if (JPG_TYPE.equals(imageType)) {
        metadata.put("Content-Type", JPG_MIME);
    } else if (PNG_TYPE.equals(imageType)) {
        metadata.put("Content-Type", PNG_MIME);
    }

    PutObjectRequest put0bjectRequest = PutObjectRequest.builder()
        .bucket(bucket)
        .key(key)
        .metadata(metadata)
        .build();

    // Uploading to S3 destination bucket
    logger.log("Writing to: " + bucket + "/" + key);
    try {
        s3Client.putObject(put0bjectRequest,
            RequestBody.fromBytes(outputStream.toByteArray()));
    }
    catch(AwsServiceException e)
    {
        logger.log(e.awsErrorDetails().errorMessage());
        System.exit(1);
    }
}

/**
 * Resizes (shrinks) an image into a small, thumbnail-sized image.
 *
 * The new image is scaled down proportionally based on the source
 * image. The scaling factor is determined based on the value of
 * MAX_DIMENSION. The resulting new image has max(height, width)
 * = MAX_DIMENSION.
 *
 * @param srcImage BufferedImage to resize.
 * @return New BufferedImage that is scaled down to thumbnail size.
 */
private BufferedImage resizeImage(BufferedImage srcImage) {
    int srcHeight = srcImage.getHeight();
    int srcWidth = srcImage.getWidth();
    // Infer scaling factor to avoid stretching image unnaturally
    float scalingFactor = Math.min(
        MAX_DIMENSION / srcWidth, MAX_DIMENSION / srcHeight);
    int width = (int) (scalingFactor * srcWidth);
    int height = (int) (scalingFactor * srcHeight);

    BufferedImage resizedImage = new BufferedImage(width, height,
        BufferedImage.TYPE_INT_RGB);
    Graphics2D graphics = resizedImage.createGraphics();
    // Fill with white before applying semi-transparent (alpha) images
    graphics.setPaint(Color.white);
    graphics.fillRect(0, 0, width, height);
    // Simple bilinear resize
    graphics.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
        RenderingHints.VALUE_INTERPOLATION_BILINEAR);
    graphics.drawImage(srcImage, 0, 0, width, height, null);
}

```

```
    graphics.dispose();
    return resizedImage;
}
```

5. The `mvn -B archetype:generate` command from step 1 also generated a dummy test case in the `src/test` directory. For the purposes of this tutorial, skip over running tests by deleting this entire generated `/test` directory.

Note

As a best practice, always thoroughly test your code. For a working example of how you might unit test this application, see [the s3-java example on GitHub](#).

6. Use the following command to build your project.

```
mvn package
```

This command generates a JAR file in the target directory.

Step 5. Create the Lambda function

Node.js

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs16.x \
--timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/lambda-s3-role
```

For the `role` parameter, replace `123456789012` with your [AWS account ID](#).

The `create-function` command specifies the function handler as `index.handler`. This handler name reflects the function name as `handler`, and the name of the file where the handler code is stored as `index.js`. For more information, see [AWS Lambda function handler in Node.js \(p. 258\)](#). The command specifies a runtime of `nodejs16.x`. For more information, see [Lambda runtimes \(p. 37\)](#).

Python

Run the following AWS CLI command to deploy the package and create the Lambda function. For the `role` parameter, replace `123456789012` with your [AWS account ID](#).

```
aws lambda create function --function-name CreateThumbnail --runtime python3.9 \
--handler lambda_function.lambda_handler --zip-file fileb://lambda_function.zip \
--role arn:aws:iam::123456789012:role/lambda-s3-role \
--timeout 10 --memory-size 1024
```

Java

First, ensure that you're in the target directory of your project.

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://create-thumbnail-1.0-SNAPSHOT.jar --handler
com.example.CreateThumbnail.Handler --runtime java11 \
--timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/lambda-s3-role
```

For the `role` parameter, replace `123456789012` with your [AWS account ID](#).

The **create-function** command specifies the function handler as `com.example.CreateThumbnail.handler`. The command specifies a runtime of `java11`. For more information, see [Lambda runtimes \(p. 37\)](#).

The function configuration includes a 10-second timeout value. Depending on the size of objects that you upload, you might need to increase the timeout value using the following AWS CLI command:

```
aws lambda update-function-configuration --function-name CreateThumbnail --timeout 30
```

Step 6. Test the Lambda function

Invoke the Lambda function manually using sample Amazon S3 event data.

To test the Lambda function

1. In the project directory that you created earlier, save the following Amazon S3 sample event data in a file named `inputFile.txt`. Be sure to replace the following values:
 - `us-west-2` – The AWS Region where you created the Amazon S3 bucket and the Lambda function.
 - `sourcebucket` – The Amazon S3 source bucket that you created in step 1.
 - `HappyFace.jpg` – The object key of the .jpg or .png image that you uploaded to the source bucket.

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-west-2",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "AIDAJDPLRKLG7UEXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "C3D13FE58DE4C810",
        "x-amz-id-2": "FMyUVURIY8/IgAtTv8xRjskZQpcIZ9KG4V5Wp6S7S/JRWeUWerMUE5JgHvAN0jpD"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "sourcebucket",
          "ownerIdentity": {
            "principalId": "A3NL1KOZZKExample"
          },
          "arn": "arn:aws:s3:::sourcebucket"
        },
        "object": {
          "key": "HappyFace.jpg",
          "size": 1024,
          "eTag": "d41d8cd98f00b204e9800998ecf8427e",
          "versionId": "096fKKXTRTtl3on89fV0.nfljtsv6qko"
        }
      }
    }
  ]
}
```

```
        }  
    ]  
}
```

2. Invoke the function with the following **invoke** command. Note that the command requests asynchronous execution (--invocation-type Event). Optionally, you can invoke the function synchronously by specifying RequestResponse as the invocation-type parameter value.

```
aws lambda invoke --function-name CreateThumbnail \  
  --cli-binary-format raw-in-base64-out \  
  --invocation-type Event \  
  --payload file://inputFile.txt outputFile.txt
```

- The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run aws configure set cli-binary-format raw-in-base64-out. For more information, see [AWS CLI supported global command line options](#).
 - If you get the error "Error parsing parameter '--payload': Unable to load paramfile file:// inputFile.txt", make sure that you're in the directory where the inputFile.txt is saved.
3. Verify that the thumbnail is created in the target S3 bucket.

Step 7. Configure Amazon S3 to publish events

Complete the configuration so that Amazon S3 can publish object-created events to Lambda and invoke your Lambda function. In this step, you do the following:

- Add permissions to the function access policy to allow Amazon S3 to invoke the function.
- Add a notification configuration to your source S3 bucket. In the notification configuration, you provide the following:
 - The event type for which you want Amazon S3 to publish events. For this tutorial, specify the s3:ObjectCreated:* event type so that Amazon S3 publishes events when objects are created.
 - The function to invoke.

To add permissions to the function policy

1. Run the following **add-permission** command to grant Amazon S3 service principal (s3.amazonaws.com) permissions to perform the lambda:InvokeFunction action. Note that Amazon S3 is granted permission to invoke the function only if the following conditions are met:
 - An object-created event is detected on a specific S3 bucket.
 - The S3 bucket is owned by your AWS account. If you delete a bucket, it is possible for another AWS account to create a bucket with the same Amazon Resource Name (ARN).

```
aws lambda add-permission --function-name CreateThumbnail --principal s3.amazonaws.com \  
  \  
  --statement-id s3invoke --action "lambda:InvokeFunction" \  
  --source-arn arn:aws:s3:::sourcebucket \  
  --source-account account-id
```

2. Verify the function's access policy by running the **get-policy** command.

```
aws lambda get-policy --function-name CreateThumbnail
```

To have Amazon S3 publish object-created events to Lambda, add a notification configuration on the source S3 bucket.

Important

This procedure configures the S3 bucket to invoke your function every time that an object is created in the bucket. Be sure to configure this option only on the source bucket. Do not have your function create objects in the source bucket, or your function could be [invoked continuously in a loop \(p. 741\)](#).

To configure notifications

1. Open the [Amazon S3 console](#).
2. Choose the name of the source S3 bucket.
3. Choose the **Properties** tab.
4. Under **Event notifications**, choose **Create event notification** to configure a notification with the following settings:
 - **Event name – lambda-trigger**
 - **Event types – All object create events**
 - **Destination – Lambda function**
 - **Lambda function – CreateThumbnail**.

For more information on event configuration, see [Enabling and configuring event notifications using the Amazon S3 console](#) in the *Amazon Simple Storage Service User Guide*.

Step 8. Test using the Amazon S3 trigger

Test the setup as follows:

1. Upload .jpg or .png objects to the source S3 bucket using the [Amazon S3 console](#).
2. Verify for each image object that a thumbnail is created in the target S3 bucket using the CreateThumbnail Lambda function.
3. View logs in the [CloudWatch console](#).

Step 9. Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

To delete the policy that you created

1. Open the [Policies page](#) of the IAM console.
2. Select the policy that you created (**AWSLambdaS3Policy**).
3. Choose **Policy actions, Delete**.

4. Choose **Delete**.

To delete the execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the S3 bucket

1. Open the [Amazon S3 console](#).
2. Select the bucket you created.
3. Choose **Delete**.
4. Enter the name of the bucket in the text input field.
5. Choose **Delete bucket**.

Using AWS Lambda with Amazon S3 batch operations

You can use Amazon S3 batch operations to invoke a Lambda function on a large set of Amazon S3 objects. Amazon S3 tracks the progress of batch operations, sends notifications, and stores a completion report that shows the status of each action.

To run a batch operation, you create an Amazon S3 [batch operations job](#). When you create the job, you provide a manifest (the list of objects) and configure the action to perform on those objects.

When the batch job starts, Amazon S3 invokes the Lambda function [synchronously \(p. 120\)](#) for each object in the manifest. The event parameter includes the names of the bucket and the object.

The following example shows the event that Amazon S3 sends to the Lambda function for an object that is named **customerImage1.jpg** in the **examplebucket** bucket.

Example Amazon S3 batch request event

```
{  
    "invocationSchemaVersion": "1.0",  
    "invocationId": "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",  
    "job": {  
        "id": "f3cc4f60-61f6-4a2b-8a21-d07600c373ce"  
    },  
    "tasks": [  
        {  
            "taskId": "dGFza2lkZ29lc2hlcmUK",  
            "s3Key": "customerImage1.jpg",  
            "s3VersionId": "1",  
            "s3BucketArn": "arn:aws:s3:::examplebucket"  
        }  
    ]  
}
```

Your Lambda function must return a JSON object with the fields as shown in the following example. You can copy the `invocationId` and `taskId` from the event parameter. You can return a string in the `resultString`. Amazon S3 saves the `resultString` values in the completion report.

Example Amazon S3 batch request response

```
{  
    "invocationSchemaVersion": "1.0",  
    "treatMissingKeysAs" : "PermanentFailure",  
    "invocationId" : "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",  
    "results": [  
        {  
            "taskId": "dGFza2lkZ29lc2hlcmUK",  
            "resultCode": "Succeeded",  
            "resultString": "[\"Alice\", \"Bob\"]"  
        }  
    ]  
}
```

Invoking Lambda functions from Amazon S3 batch operations

You can invoke the Lambda function with an unqualified or qualified function ARN. If you want to use the same function version for the entire batch job, configure a specific function version in the `FunctionARN` parameter when you create your job. If you configure an alias or the `$LATEST` qualifier, the batch job immediately starts calling the new version of the function if the alias or `$LATEST` is updated during the job execution.

Note that you can't reuse an existing Amazon S3 event-based function for batch operations. This is because the Amazon S3 batch operation passes a different event parameter to the Lambda function and expects a return message with a specific JSON structure.

In the [resource-based policy \(p. 832\)](#) that you create for the Amazon S3 batch job, ensure that you set permission for the job to invoke your Lambda function.

In the execution role for the function, set a [trust policy for Amazon S3 to assume the role when it runs your function](#).

If your function uses the AWS SDK to manage Amazon S3 resources, you need to add Amazon S3 permissions in the execution role.

When the job runs, Amazon S3 starts multiple function instances to process the Amazon S3 objects in parallel, up to the [concurrency limit \(p. 197\)](#) of the function. Amazon S3 limits the initial ramp-up of instances to avoid excess cost for smaller jobs.

If the Lambda function returns a `TemporaryFailure` response code, Amazon S3 retries the operation.

For more information about Amazon S3 batch operations, see [Performing batch operations](#) in the [Amazon S3 Developer Guide](#).

For an example of how to use a Lambda function in Amazon S3 batch operations, see [Invoking a Lambda function from Amazon S3 batch operations](#) in the [Amazon S3 Developer Guide](#).

Transforming S3 Objects with S3 Object Lambda

With S3 Object Lambda you can add your own code to Amazon S3 GET, HEAD, and LIST requests to modify and process data before it is returned to an application. You can use custom code to modify the data returned by standard S3 GET, HEAD, or LIST requests to filter rows, dynamically resize images, redact confidential data, and more. Powered by AWS Lambda functions, your code runs on infrastructure that is fully managed by AWS, eliminating the need to create and store derivative copies of your data or to run proxies, all with no changes required to applications.

For more information, see [Transforming objects with S3 Object Lambda](#).

Tutorials

- [Transforming data for your application with Amazon S3 Object Lambda](#)
- [Detecting and redacting PII data with Amazon S3 Object Lambda and Amazon Comprehend](#)
- [Using Amazon S3 Object Lambda to Dynamically Watermark Images as They Are Retrieved](#)

Using AWS Lambda with Secrets Manager

Your AWS Lambda function can interact with AWS Secrets Manager using the [Secrets Manager API](#) or any of the AWS Software Development Kits (SDKs). You can also use the AWS Parameters and Secrets Lambda Extension to retrieve and cache AWS Secrets Manager secrets in Lambda functions without using an SDK. See [Use AWS Secrets Manager secrets in AWS Lambda functions](#) for more information.

Using AWS Lambda with Amazon SES

When you use Amazon SES to receive messages, you can configure Amazon SES to call your Lambda function when messages arrive. The service can then invoke your Lambda function by passing in the incoming email event, which in reality is an Amazon SES message in an Amazon SNS event, as a parameter.

Example Amazon SES message event

```
{
  "Records": [
    {
      "eventVersion": "1.0",
      "ses": {
        "mail": {
          "commonHeaders": {
            "from": [
              "Jane Doe <janedoe@example.com>"
            ],
            "to": [
              "johndoe@example.com"
            ],
            "returnPath": "janedoe@example.com",
            "messageId": "<0123456789example.com>",
            "date": "Wed, 7 Oct 2015 12:34:56 -0700",
            "subject": "Test Subject"
          },
          "source": "janedoe@example.com",
          "timestamp": "1970-01-01T00:00:00.000Z",
          "destination": [
            "johndoe@example.com"
          ],
          "headers": [
            {
              "name": "Return-Path",
              "value": "<janedoe@example.com>"
            },
            {
              "name": "Received",
              "value": "from mailer.example.com (mailer.example.com [203.0.113.1]) by inbound-smtp.us-west-2.amazonaws.com with SMTP id o3vrnil0e2ic for johndoe@example.com; Wed, 07 Oct 2015 12:34:56 +0000 (UTC)"
            },
            {
              "name": "DKIM-Signature",
              "value": "v=1; a=rsa-sha256; c=relaxed/relaxed; d=example.com; s=example; h=mime-version:from:date:message-id:subject:to:content-type; bh=jX3F0bCAI7sIbkHyy3mLY028ieDQz2R0P8HwQkk1Fj4=; b=sQwJ+LMe9RjkesGu+vqU56asvMhrLRRYrWCbV"
            },
            {
              "name": "MIME-Version",
              "value": "1.0"
            },
            {
              "name": "From",
              "value": "Jane Doe <janedoe@example.com>"
            },
            {
              "name": "Date",
              "value": "Wed, 7 Oct 2015 12:34:56 -0700"
            },
            {
              "name": "Message-ID",
              "value": "<0123456789example.com>@"
            }
          ]
        }
      }
    }
  ]
}
```

```
        "value": "<0123456789example.com>"  
    },  
    {  
        "name": "Subject",  
        "value": "Test Subject"  
    },  
    {  
        "name": "To",  
        "value": "johndoe@example.com"  
    },  
    {  
        "name": "Content-Type",  
        "value": "text/plain; charset=UTF-8"  
    }  
],  
"headersTruncated": false,  
"messageId": "o3vrnil0e2ic28tr"  
},  
"receipt": {  
    "recipients": [  
        "johndoe@example.com"  
    ],  
    "timestamp": "1970-01-01T00:00:00.000Z",  
    "spamVerdict": {  
        "status": "PASS"  
    },  
    "dkimVerdict": {  
        "status": "PASS"  
    },  
    "processingTimeMillis": 574,  
    "action": {  
        "type": "Lambda",  
        "invocationType": "Event",  
        "functionArn": "arn:aws:lambda:us-west-2:111122223333:function:Example"  
    },  
    "spfVerdict": {  
        "status": "PASS"  
    },  
    "virusVerdict": {  
        "status": "PASS"  
    }  
},  
"eventSource": "aws:ses"  
}  
]  
}
```

For more information, see [Lambda action](#) in the *Amazon SES Developer Guide*.

Using AWS Lambda with Amazon SNS

You can use a Lambda function to process Amazon Simple Notification Service (Amazon SNS) notifications. Amazon SNS supports Lambda functions as a target for messages sent to a topic. You can subscribe your function to topics in the same account or in other AWS accounts.

Amazon SNS invokes your function [asynchronously \(p. 123\)](#) with an event that contains a message and metadata.

Example Amazon SNS message event

```
{  
  "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&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"  
      }  
    }  
  ]  
}
```

For asynchronous invocation, Lambda queues the message and handles retries. If Amazon SNS can't reach Lambda or the message is rejected, Amazon SNS retries at increasing intervals over several hours. For details, see [Reliability](#) in the Amazon SNS FAQs.

To perform cross-account Amazon SNS deliveries to Lambda, you must authorize Amazon SNS to invoke your Lambda function. In turn, Amazon SNS must allow the AWS account with the Lambda function to subscribe to the Amazon SNS topic. For example, if the Amazon SNS topic is in account A and the Lambda function is in account B, both accounts must grant permissions to the other to access their respective resources. Since not all the options for setting up cross-account permissions are available from the AWS Management Console, you must use the AWS Command Line Interface (AWS CLI) for setup.

For more information, see [Fanout to Lambda functions](#) in the *Amazon Simple Notification Service Developer Guide*.

Input types for Amazon SNS events

For input type examples for Amazon SNS events in Java, .NET, and Go, see the following on the AWS GitHub repository:

- [SNSEvent.java](#)
- [SNSEvent.cs](#)
- [sns.go](#)

Topics

- [Tutorial: Using AWS Lambda with Amazon Simple Notification Service \(p. 771\)](#)
- [Sample function code \(p. 775\)](#)

Tutorial: Using AWS Lambda with Amazon Simple Notification Service

You can use a Lambda function in one AWS account to subscribe to an Amazon SNS topic in a separate AWS account. In this tutorial, you use the AWS Command Line Interface to perform AWS Lambda operations such as creating a Lambda function, creating an Amazon SNS topic and granting permissions to allow these two resources to access each other.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console \(p. 4\)](#) to create your first Lambda function.

To complete the following steps, you need the [AWS Command Line Interface \(AWS CLI\) version 2](#). Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.0.57 Python/3.7.4 Darwin/19.6.0 exe/x86_64
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#). Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

In the tutorial, you use two accounts. The AWS CLI commands illustrate this by using two [named profiles](#), each configured for use with a different account. If you use profiles with different names, or the default profile and one named profile, modify the commands as needed.

Create an Amazon SNS topic (account A)

In **Account A**, create the source Amazon SNS topic.

```
aws sns create-topic --name sns-topic-for-lambda --profile accountA
```

After creating the topic, record its Amazon Resource Name (ARN). You need it later when you add permissions to the Lambda function to subscribe to the topic.

Create the execution role (account B)

In **Account B**, create the [execution role \(p. 816\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – AWS Lambda**.
 - **Permissions – AWSLambdaBasicExecutionRole**.
 - **Role name – lambda-sns-role**.

The **AWSLambdaBasicExecutionRole** policy has the permissions that the function needs to write logs to CloudWatch Logs.

Create a Lambda function (account B)

In **Account B**, create the function that processes events from Amazon SNS. The following example code receives an Amazon SNS event input and processes the messages that it contains. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Note

For sample code in other languages, see [Sample function code \(p. 775\)](#).

Example index.js

```
console.log('Loading function');

exports.handler = function(event, context, callback) {
// console.log('Received event:', JSON.stringify(event, null, 4));

    var message = event.Records[0].Sns.Message;
    console.log('Message received from SNS:', message);
    callback(null, "Success");
};
```

To create the function

1. Copy the sample code into a file named `index.js`.
2. Create a deployment package.

```
zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command.

```
aws lambda create-function --function-name Function-With-SNS \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs12.x \
```

```
--role arn:aws:iam::<AccountB_ID>:role/lambda-sns-role \
--timeout 60 --profile accountB
```

After creating the function, record its function ARN. You need it later when you add permissions to allow Amazon SNS to invoke your function.

Set up cross-account permissions (account A and B)

In **Account A**, grant permission to **Account B** to subscribe to the topic:

```
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
```

In **Account B**, add the Lambda permission to allow invocation from Amazon SNS.

```
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
```

You should see the following output:

```
{
  "Statement": "{\"Condition\": {\"ArnLike\": {\"AWS:SourceArn\": \"arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda\"}},\n    \"Action\": [\"lambda:InvokeFunction\"],\n    \"Resource\": \"arn:aws:lambda:us-east-1:<AccountB_ID>:function:Function-With-SNS\", \n    \"Effect\": \"Allow\", \"Principal\": {\"Service\": \"sns.amazonaws.com\"},\n    \"Sid\": \"function-with-sns1\"}"
}
```

Do not use the `--source-account` parameter to add a source account to the Lambda policy when adding the policy. Source account is not supported for Amazon SNS event sources and will result in access being denied.

Note

If the account with the SNS topic is hosted in an [opt-in AWS Region](#), you need to specify the Region in the principal. For example, if you're working with an SNS topic in the Asia Pacific (Hong Kong) Region, you need to specify `sns.ap-east-1.amazonaws.com` instead of `sns.amazonaws.com` for the principal.

Create a subscription (account B)

In **Account B**, subscribe the Lambda function to the topic. When a message is sent to the `sns-topic-for-lambda` topic in **Account A**, Amazon SNS invokes the `Function-With-SNS` function in **Account B**.

```
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
```

Note

The `--region` option ensures that the command runs in the Region where the SNS topic resides. You must include this option if you [configured the AWS CLI for a different Region](#).

You should see the following output:

```
{  
    "SubscriptionArn": "arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda:5d906xxxx-7c8x-45dx-a9dx-0484e31c98xx"  
}
```

The output contains the ARN of the topic subscription.

Test subscription (account A)

In **Account A**, test the subscription. Type Hello World into a text file and save it as `message.txt`. Then run the following command:

```
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
```

This will return a message id with a unique identifier, indicating the message has been accepted by the Amazon SNS service. Amazon SNS will then attempt to deliver it to the topic's subscribers. Alternatively, you could supply a JSON string directly to the message parameter, but using a text file allows for line breaks in the message.

To learn more about Amazon SNS, see [What is Amazon Simple Notification Service](#).

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

In **Account A**, clean up your Amazon SNS topic.

To delete the Amazon SNS topic

1. Open the [Topics page](#) of the Amazon SNS console.
2. Select the topic you created.
3. Choose **Delete**.
4. Enter **delete me** in the text input field.
5. Choose **Delete**.

In **Account B**, clean up your execution role, Lambda function, and Amazon SNS subscription.

To delete the execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.

3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

To delete the Amazon SNS subscription

1. Open the [Subscriptions page](#) of the Amazon SNS console.
2. Select the subscription you created.
3. Choose **Delete, Delete**.

Sample function code

Sample code is available for the following languages.

Topics

- [Node.js 12.x \(p. 775\)](#)
- [Java 11 \(p. 775\)](#)
- [Go \(p. 776\)](#)
- [Python 3 \(p. 776\)](#)

Node.js 12.x

The following example processes messages from Amazon SNS, and logs their contents.

Example index.js

```
console.log('Loading function');

exports.handler = function(event, context, callback) {
// console.log('Received event:', JSON.stringify(event, null, 4));

    var message = event.Records[0].Sns.Message;
    console.log('Message received from SNS:', message);
    callback(null, "Success");
};
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Node.js Lambda functions with .zip file archives \(p. 263\)](#).

Java 11

The following example processes messages from Amazon SNS, and logs their contents.

Example LogEvent.java

```
package example;

import java.text.SimpleDateFormat;
import java.util.Calendar;

import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;

public class LogEvent implements RequestHandler<SNSEvent, Object> {
```

```
public Object handleRequest(SNSEvent request, Context context){  
    String timeStamp = new SimpleDateFormat("yyyy-MM-  
dd_HH:mm:ss").format(Calendar.getInstance().getTime());  
    context.getLogger().log("Invocation started: " + timeStamp);  
    context.getLogger().log(request.getRecords().get(0).getSNS().getMessage());  
  
    timeStamp = new SimpleDateFormat("yyyy-MM-  
dd_HH:mm:ss").format(Calendar.getInstance().getTime());  
    context.getLogger().log("Invocation completed: " + timeStamp);  
    return null;  
}  
}
```

Dependencies

- aws-lambda-java-core
- aws-lambda-java-events

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [Deploy Java Lambda functions with .zip or JAR file archives \(p. 393\)](#).

Go

The following example processes messages from Amazon SNS, and logs their contents.

Example lambda_handler.go

```
package main  
  
import (  
    "context"  
    "fmt"  
    "github.com/aws/aws-lambda-go/lambda"  
    "github.com/aws/aws-lambda-go/events"  
)  
  
func handler(ctx context.Context, snsEvent events.SNSEvent) {  
    for _, record := range snsEvent.Records {  
        snsRecord := record.SNS  
        fmt.Printf("[%s %s] Message = %s \n", record.EventSource, snsRecord.Timestamp,  
        snsRecord.Message)  
    }  
}  
  
func main() {  
    lambda.Start(handler)  
}
```

Build the executable with `go build` and create a deployment package. For instructions, see [Deploy Go Lambda functions with .zip file archives \(p. 460\)](#).

Python 3

The following example processes messages from Amazon SNS, and logs their contents.

Example lambda_handler.py

```
from __future__ import print_function  
import json
```

```
print('Loading function')

def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))
    message = event['Records'][0]['Sns']['Message']
    print("From SNS: " + message)
    return message
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Python Lambda functions with .zip file archives \(p. 324\)](#).

Using Lambda with Amazon SQS

Note

If you want to send data to a target other than a Lambda function or enrich the data before sending it, see [Amazon EventBridge Pipes](#).

You can use a Lambda function to process messages in an Amazon Simple Queue Service (Amazon SQS) queue. Lambda [event source mappings \(p. 131\)](#) support [standard queues](#) and [first-in, first-out \(FIFO\) queues](#). With Amazon SQS, you can offload tasks from one component of your application by sending them to a queue and processing them asynchronously.

Lambda polls the queue and invokes your Lambda function [synchronously \(p. 120\)](#) with an event that contains queue messages. Lambda reads messages in batches and invokes your function once for each batch. When your function successfully processes a batch, Lambda deletes its messages from the queue.

When Lambda reads a batch, the messages stay in the queue but are hidden for the length of the queue's [visibility timeout](#). If your function successfully processes the batch, Lambda deletes the messages from the queue. By default, if your function encounters an error while processing a batch, all messages in that batch become visible in the queue again. For this reason, your function code must be able to process the same message multiple times without unintended side effects.

To prevent Lambda from processing a message multiple times, you can either configure your event source mapping to include [batch item failures \(p. 784\)](#) in your function response, or you can use the Amazon SQS API action [DeleteMessage](#) to remove messages from the queue as your Lambda function successfully processes them. For more information on using the Amazon SQS API, see the [Amazon Simple Queue Service API Reference](#)

Sections

- [Example standard queue message event \(p. 778\)](#)
- [Example FIFO queue message event \(p. 779\)](#)
- [Configuring a queue to use with Lambda \(p. 780\)](#)
- [Execution role permissions \(p. 780\)](#)
- [Configuring a queue as an event source \(p. 780\)](#)
- [Scaling and processing \(p. 782\)](#)
- [Configuring maximum concurrency for Amazon SQS event sources \(p. 782\)](#)
- [Event source mapping APIs \(p. 783\)](#)
- [Backoff strategy for failed invocations \(p. 784\)](#)
- [Implementing partial batch responses \(p. 784\)](#)
- [Amazon SQS configuration parameters \(p. 786\)](#)
- [Tutorial: Using Lambda with Amazon SQS \(p. 787\)](#)
- [Tutorial: Using a cross-account Amazon SQS queue as an event source \(p. 791\)](#)
- [Sample Amazon SQS function code \(p. 795\)](#)
- [AWS SAM template for an Amazon SQS application \(p. 798\)](#)

Example standard queue message event

Example Amazon SQS message event (standard queue)

```
{
```

```

"Records": [
    {
        "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
        "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCxlaS3SLy0a...",
        "body": "Test message.",
        "attributes": {
            "ApproximateReceiveCount": "1",
            "SentTimestamp": "1545082649183",
            "SenderId": "AIDAENQZJ0L023YVJ4V0",
            "ApproximateFirstReceiveTimestamp": "1545082649185"
        },
        "messageAttributes": {},
        "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": "AIDAENQZJ0L023YVJ4V0",
            "ApproximateFirstReceiveTimestamp": "1545082650649"
        },
        "messageAttributes": {},
        "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
        "eventSource": "aws:sqs",
        "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
        "awsRegion": "us-east-2"
    }
]
}

```

By default, Lambda polls up to 10 messages in your queue at once and sends that batch to your function. To avoid invoking the function with a small number of records, you can tell the event source to buffer records for up to 5 minutes by configuring a batch window. Before invoking the function, Lambda continues to poll messages from the SQS standard queue until the batch window expires, the [invocation payload size quota \(p. 1131\)](#) is reached, or the configured maximum batch size is reached.

Note

If you're using a batch window and your SQS queue contains very low traffic, Lambda might wait for up to 20 seconds before invoking your function. This is true even if you set a batch window lower than 20 seconds.

Example FIFO queue message event

For FIFO queues, records contain additional attributes that are related to deduplication and sequencing.

Example Amazon SQS message event (FIFO queue)

```

{
    "Records": [
        {
            "messageId": "11d6ee51-4cc7-4302-9e22-7cd8afdaadf5",
            "receiptHandle": "AQEBBX8nesZExmkhsmZeyIE8iQAMig7qw...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1573251510774",

```

```
        "SequenceNumber": "18849496460467696128",
        "MessageGroupId": "1",
        "SenderId": "AIDAI023YVJENQZJOL4VO",
        "MessageDuplicationId": "1",
        "ApproximateFirstReceiveTimestamp": "1573251510774"
    },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:fifo fifo",
    "awsRegion": "us-east-2"
}
]
```

Configuring a queue to use with Lambda

Create an [SQS queue](#) to serve as an event source for your Lambda function. Then configure the queue to allow time for your Lambda function to process each batch of events—and for Lambda to retry in response to throttling errors as it scales up.

To allow your function time to process each batch of records, set the source queue's [visibility timeout](#) to at least six times the [timeout that you configure \(p. 72\)](#) on your function. The extra time allows for Lambda to retry if your function is throttled while processing a previous batch.

If your function fails to process a message multiple times, Amazon SQS can send it to a [dead-letter queue](#). If your function returns an error, all items in the batch return to the queue. After the [visibility timeout](#) occurs, Lambda receives the message again. To send messages to a second queue after a number of receives, configure a dead-letter queue on your source queue.

Note

Make sure that you configure the dead-letter queue on the source queue, not on the Lambda function. The dead-letter queue that you configure on a function is used for the function's [asynchronous invocation queue \(p. 123\)](#), not for event source queues.

If your function returns an error, or can't be invoked because it's at maximum concurrency, processing might succeed with additional attempts. To give messages a better chance to be processed before sending them to the dead-letter queue, set the `maxReceiveCount` on the source queue's redrive policy to at least 5.

Execution role permissions

Lambda needs the following permissions to manage messages in your Amazon SQS queue. Add them to your function's [execution role \(p. 816\)](#).

- [sq:ReceiveMessage](#)
- [sq:DeleteMessage](#)
- [sq:GetQueueAttributes](#)

Configuring a queue as an event source

Create an event source mapping to tell Lambda to send items from your queue to a Lambda function. You can create multiple event source mappings to process items from multiple queues with a single function. When Lambda invokes the target function, the event can contain multiple items, up to a configurable maximum *batch size*.

To configure your function to read from Amazon SQS in the Lambda console, create an **SQS** trigger.

To create a trigger

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of a function.
3. Under **Function overview**, choose **Add trigger**.
4. Choose the **SQS** trigger type.
5. Configure the required options, and then choose **Add**.

Lambda supports the following options for Amazon SQS event sources:

SQS queue

The Amazon SQS queue to read records from.

Enable trigger

The status of the event source mapping. **Enable trigger** is selected by default.

Batch size

The number of records to send to the function in each batch. For a standard queue, this can be up to 10,000 records. For a FIFO queue, the maximum is 10. For a batch size over 10, you must also set the batch window (`MaximumBatchingWindowInSeconds`) to at least 1 second.

Configure your [function timeout](#) to allow enough time to process an entire batch of items. If items take a long time to process, choose a smaller batch size. A large batch size can improve efficiency for workloads that are very fast or have a lot of overhead. If you configure [reserved concurrency \(p. 210\)](#) on your function, set a minimum of five concurrent executions to reduce the chance of throttling errors when Lambda invokes your function.

Lambda passes all of the records in the batch to the function in a single call, as long as the total size of the events doesn't exceed the [invocation payload size quota \(p. 1131\)](#) for synchronous invocation (6 MB). Both Lambda and Amazon SQS generate metadata for each record. This additional metadata is counted towards the total payload size and can cause the total number of records sent in a batch to be lower than your configured batch size. The metadata fields that Amazon SQS sends can be variable in length. For more information about the Amazon SQS metadata fields, see the [ReceiveMessage](#) API operation documentation in the *Amazon Simple Queue Service API Reference*.

Batch window

The maximum amount of time to gather records before invoking the function, in seconds. This applies only to standard queues.

If you're using a batch window greater than 0 seconds, you must account for the increased processing time in your queue's [visibility timeout](#). We recommend setting your queue's visibility timeout to six times your [function timeout \(p. 72\)](#), plus the value of `MaximumBatchingWindowInSeconds`. This allows time for your Lambda function to process each batch of events and to retry in the event of a throttling error.

Note

If your batch window is greater than 0, and $(\text{batch window}) + (\text{function timeout}) > (\text{queue visibility timeout})$, then your effective queue visibility timeout is $(\text{batch window}) + (\text{function timeout}) + 30s$.

When messages become available, Lambda starts processing messages in batches. Lambda starts processing five batches at a time with five concurrent invocations of your function. If messages are still available, Lambda adds up to 60 more instances of your function a minute, up to a maximum

of 1,000 function instances. To learn more about function scaling and concurrency, see [Lambda function scaling](#).

To process more messages, you can optimize your Lambda function for higher throughput. See [Understanding how AWS Lambda scales with Amazon SQS standard queues](#).

Maximum concurrency

The maximum number of concurrent functions that the event source can invoke. For more information, see [Configuring maximum concurrency for Amazon SQS event sources \(p. 782\)](#).

Filter criteria

Add filter criteria to control which events Lambda sends to your function for processing. For more information, see [Lambda event filtering \(p. 136\)](#).

Scaling and processing

For standard queues, Lambda uses [long polling](#) to poll a queue until it becomes active. When messages are available, Lambda reads up to five batches and sends them to your function. If messages are still available, Lambda increases the number of processes that are reading batches by up to 60 more instances per minute. The maximum number of batches that an event source mapping can process simultaneously is 1,000.

For FIFO queues, Lambda sends messages to your function in the order that it receives them. When you send a message to a FIFO queue, you specify a [message group ID](#). Amazon SQS ensures that messages in the same group are delivered to Lambda in order. Lambda sorts the messages into groups and sends only one batch at a time for a group. If your function returns an error, the function attempts all retries on the affected messages before Lambda receives additional messages from the same group.

Your function can scale in concurrency to the number of active message groups. For more information, see [SQS FIFO as an event source](#) on the AWS Compute Blog.

Configuring maximum concurrency for Amazon SQS event sources

The maximum concurrency setting limits the number of concurrent instances of the function that an Amazon SQS event source can invoke. Maximum concurrency is an event source-level setting. If you have multiple Amazon SQS event sources mapped to one function, each event source can have a separate maximum concurrency setting. You can use maximum concurrency to prevent one queue from using all of the function's [reserved concurrency \(p. 210\)](#) or the rest of the [account's concurrency quota \(p. 1131\)](#). You can use maximum concurrency and reserved concurrency together or independently.

Note

You can't set maximum concurrency higher than the function's reserved concurrency. After configuring maximum concurrency, make sure that you don't reduce the function's reserved concurrency to less than the total maximum concurrency for all Amazon SQS event sources on the function. Otherwise, Lambda might throttle your messages. For FIFO queues, maximum concurrency is capped by the number of message groups.

There is no charge for configuring maximum concurrency on an Amazon SQS event source.

You can configure maximum concurrency on new and existing Amazon SQS event source mappings.

Configure maximum concurrency using the Lambda console

1. Open the [Functions page](#) of the Lambda console.

2. Choose the name of a function.
3. Under **Function overview**, choose **SQS**. This opens the **Configuration** tab.
4. Select the Amazon SQS trigger and choose **Edit**.
5. For **Maximum concurrency**, enter a number between 2 and 1,000. To turn off maximum concurrency, leave the box empty.
6. Choose **Save**.

Configure maximum concurrency using the AWS Command Line Interface (AWS CLI)

Use the [update-event-source-mapping](#) command with the `--scaling-config` option. Example:

```
aws lambda update-event-source-mapping \
--uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
--scaling-config '{"MaximumConcurrency":5}'
```

To turn off maximum concurrency, enter an empty value for `--scaling-config`:

```
aws lambda update-event-source-mapping \
--uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
--scaling-config "{}"
```

Configure maximum concurrency using the Lambda API

Use the [CreateEventSourceMapping \(p. 1153\)](#) or [UpdateEventSourceMapping \(p. 1356\)](#) action with a [ScalingConfig \(p. 1458\)](#) object.

Event source mapping APIs

To manage an event source with the [AWS Command Line Interface \(AWS CLI\)](#) or an [AWS SDK](#), you can use the following API operations:

- [CreateEventSourceMapping \(p. 1153\)](#)
- [ListEventSourceMappings \(p. 1279\)](#)
- [GetEventSourceMapping \(p. 1214\)](#)
- [UpdateEventSourceMapping \(p. 1356\)](#)
- [DeleteEventSourceMapping \(p. 1186\)](#)

The following example uses the AWS CLI to map a function named `my-function` to an Amazon SQS queue that is specified by its Amazon Resource Name (ARN), with a batch size of 5 and a batch window of 60 seconds.

```
aws lambda create-event-source-mapping --function-name my-function --batch-size 5 \
--maximum-batching-window-in-seconds 60 \
--event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue
```

You should see the following output:

```
{  
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",  
    "BatchSize": 5,
```

```
"MaximumBatchingWindowInSeconds": 60,  
"EventSourceArn": "arn:aws:sqs:us-east-2:123456789012:my-queue",  
"FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",  
"LastModified": 1541139209.351,  
"State": "Creating",  
"StateTransitionReason": "USER_INITIATED"  
}
```

Backoff strategy for failed invocations

When an invocation fails, Lambda attempts to retry the invocation while implementing a backoff strategy. The backoff strategy differs slightly depending on whether Lambda encountered the failure due to an error in your function code, or due to throttling.

- If your **function code** caused the error, Lambda gradually backs off retries by reducing the amount of concurrency allocated to your Amazon SQS event source mapping. If invocations continue to fail, Lambda eventually drops the message without retrying.
- If the invocation fails due to **throttling**, Lambda gradually backs off retries by reducing the amount of concurrency allocated to your Amazon SQS event source mapping. Lambda continues to retry the message until the message's timestamp exceeds your queue's visibility timeout, at which point Lambda drops the message.

Implementing partial batch responses

When your Lambda function encounters an error while processing a batch, all messages in that batch become visible in the queue again by default, including messages that Lambda processed successfully. As a result, your function can end up processing the same message several times.

To avoid reprocessing successfully processed messages in a failed batch, you can configure your event source mapping to make only the failed messages visible again. This is called a partial batch response. To turn on partial batch responses, specify `ReportBatchItemFailures` for the [FunctionResponseTypes \(p. 1359\)](#) action when configuring your event source mapping. This lets your function return a partial success, which can help reduce the number of unnecessary retries on records.

When `ReportBatchItemFailures` is activated, Lambda doesn't [scale down message polling \(p. 784\)](#) when function invocations fail. If you expect some messages to fail—and you don't want those failures to impact the message processing rate—use `ReportBatchItemFailures`.

Note

Keep the following in mind when using partial batch responses:

- If your function throws an exception, the entire batch is considered a complete failure.
- If you're using this feature with a FIFO queue, your function should stop processing messages after the first failure and return all failed and unprocessed messages in `batchItemFailures`. This helps preserve the ordering of messages in your queue.

To activate partial batch reporting

1. Review the [Best practices for implementing partial batch responses](#).
2. Run the following command to activate `ReportBatchItemFailures` for your function. To retrieve your event source mapping's UUID, run the [list-event-source-mappings](#) AWS CLI command.

```
aws lambda update-event-source-mapping \  
--uuid "a1b2c3d4-5678-90ab-cdef-1111EXAMPLE" \  
\\
```

```
--function-response-types "ReportBatchItemFailures"
```

3. Update your function code to catch all exceptions and return failed messages in a batchItemFailures JSON response. The batchItemFailures response must include a list of message IDs, as itemIdentifier JSON values.

For example, suppose you have a batch of five messages, with message IDs id1, id2, id3, id4, and id5. Your function successfully processes id1, id3, and id5. To make messages id2 and id4 visible again in your queue, your function should return the following response:

```
{  
  "batchItemFailures": [  
    {  
      "itemIdentifier": "id2"  
    },  
    {  
      "itemIdentifier": "id4"  
    }  
  ]  
}
```

Here's an example of function code that returns the list of failed message IDs in the batch:

Python

Example – Python function code for batchItemFailures

```
import json  
def lambda_handler(event, context):  
    if event:  
        batch_item_failures = []  
        sqs_batch_response = {}  
  
        for record in event["Records"]:  
            try:  
                # process message  
            except Exception as e:  
                batch_item_failures.append({"itemIdentifier": record['messageId']})  
  
        sqs_batch_response["batchItemFailures"] = batch_item_failures  
    return sqs_batch_response
```

Java

Example – Java function code for batchItemFailures

```
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>();
```

```
String messageId = "";
for (SQSEvent.SQSMessages message : sqsEvent.getRecords()) {
    try {
        //process your message
        messageId = message.getMessageId();
    } catch (Exception e) {
        //Add failed message identifier to the batchItemFailures list
        batchItemFailures.add(new
SQSBatchResponse.BatchItemFailure(messageId));
    }
}
return new SQSBatchResponse(batchItemFailures);
}
```

If the failed events do not return to the queue, see [How do I troubleshoot Lambda function SQS ReportBatchItemFailures?](#) in the AWS Knowledge Center.

Success and failure conditions

Lambda treats a batch as a complete success if your function returns any of the following:

- An empty `batchItemFailures` list
- A null `batchItemFailures` list
- An empty `EventResponse`
- A null `EventResponse`

Lambda treats a batch as a complete failure if your function returns any of the following:

- An invalid JSON response
- An empty string `itemIdentifier`
- A null `itemIdentifier`
- An `itemIdentifier` with a bad key name
- An `itemIdentifier` value with a message ID that doesn't exist

CloudWatch metrics

To determine whether your function is correctly reporting batch item failures, you can monitor the `NumberOfMessagesDeleted` and `ApproximateAgeOfOldestMessage` Amazon SQS metrics in Amazon CloudWatch.

- `NumberOfMessagesDeleted` tracks the number of messages removed from your queue. If this drops to 0, this is a sign that your function response is not correctly returning failed messages.
- `ApproximateAgeOfOldestMessage` tracks how long the oldest message has stayed in your queue. A sharp increase in this metric can indicate that your function is not correctly returning failed messages.

Amazon SQS configuration parameters

All Lambda event source types share the same [CreateEventSourceMapping \(p. 1153\)](#) and [UpdateEventSourceMapping \(p. 1356\)](#) API operations. However, only some of the parameters apply to Amazon SQS.

Event source parameters that apply to Amazon SQS

Parameter	Required	Default	Notes
BatchSize	N	10	For standard queues, the maximum is 10,000. For FIFO queues, the maximum is 10.
Enabled	N	true	
EventSourceArn	Y		The ARN of the data stream or a stream consumer
FunctionName	Y		
FilterCriteria	N		Lambda event filtering (p. 136)
FunctionResponseTypes	N		To let your function report specific failures in a batch, include the value ReportBatchItemFailures in FunctionResponseTypes. For more information, see Implementing partial batch responses (p. 784) .
MaximumBatchingWindowNhSeconds	N	0	
ScalingConfig	N		Configuring maximum concurrency for Amazon SQS event sources (p. 782)

Tutorial: Using Lambda with Amazon SQS

In this tutorial, you create a Lambda function that consumes messages from an [Amazon Simple Queue Service \(Amazon SQS\) queue](#).

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console \(p. 4\)](#) to create your first Lambda function.

To complete the following steps, you need the [AWS Command Line Interface \(AWS CLI\) version 2](#). Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.0.57 Python/3.7.4 Darwin/19.6.0 exe/x86_64
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#). Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

Create the execution role

Create an [execution role \(p. 816\)](#) that gives your function permission to access the required AWS resources.

To create an execution role

1. Open the [Roles page](#) of the AWS Identity and Access Management (IAM) console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – AWS Lambda.**
 - **Permissions – AWSLambdaSQSQueueExecutionRole.**
 - **Role name – lambda-sqs-role.**

The **AWSLambdaSQSQueueExecutionRole** policy has the permissions that the function needs to read items from Amazon SQS and to write logs to Amazon CloudWatch Logs.

Create the function

Create a Lambda function that processes your Amazon SQS messages. The following Node.js 12 code example writes each message to a log in CloudWatch Logs.

Note

For code examples in other languages, see [Sample Amazon SQS function code \(p. 795\)](#).

Example index.js

```
exports.handler = async function(event, context) {
  event.Records.forEach(record => {
    const { body } = record;
    console.log(body);
  });
  return {};
}
```

To create the function

Note

Following these steps creates a function in Node.js 12. For other languages, the steps are similar, but some details are different.

1. Save the code example as a file named `index.js`.

2. Create a deployment package.

```
zip function.zip index.js
```

3. Create the function using the `create-function` AWS Command Line Interface (AWS CLI) command.

```
aws lambda create-function --function-name ProcessSQSRecord \
--zip-file file://function.zip --handler index.handler --runtime nodejs12.x \
--role arn:aws:iam::123456789012:role/lambda-sqs-role
```

Test the function

Invoke your Lambda function manually using the `invoke` AWS CLI command and a sample Amazon SQS event.

If the handler returns normally without exceptions, Lambda considers the message successfully processed and begins reading new messages in the queue. After successfully processing a message, Lambda automatically deletes it from the queue. If the handler throws an exception, Lambda considers the batch of messages not successfully processed, and Lambda invokes the function with the same batch of messages.

1. Save the following JSON as a file named `input.txt`.

```
{
  "Records": [
    {
      "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
      "receiptHandle": "AQEBwJnKyHigUMZj6rYigCgxlS3SLy0a...",
      "body": "test",
      "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAENQZJ0L023YVJ4V0",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
      },
      "messageAttributes": {},
      "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
      "eventSource": "aws:sqs",
      "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
      "awsRegion": "us-east-2"
    }
  ]
}
```

The preceding JSON simulates an event that Amazon SQS might send to your Lambda function, where "body" contains the actual message from the queue.

2. Run the following `invoke` AWS CLI command.

```
aws lambda invoke --function-name ProcessSQSRecord \
--payload file://input.txt outputfile.txt
```

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

3. Verify the output in the file `outputfile.txt`.

Create an Amazon SQS queue

Create an Amazon SQS queue that the Lambda function can use as an event source.

To create a queue

1. Open the [Amazon SQS console](#).
2. Choose **Create queue**, and then configure the queue. For detailed instructions, see [Creating an Amazon SQS queue \(console\)](#) in the *Amazon Simple Queue Service Developer Guide*.
3. After creating the queue, record its Amazon Resource Name (ARN). You need this in the next step when you associate the queue with your Lambda function.

Configure the event source

To create a mapping between your Amazon SQS queue and your Lambda function, run the following `create-event-source-mapping` AWS CLI command.

```
aws lambda create-event-source-mapping --function-name ProcessSQSRecord --batch-size 10 \
--event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue
```

To get a list of your event source mappings, run the following command.

```
aws lambda list-event-source-mappings --function-name ProcessSQSRecord \
--event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue
```

Test the setup

Now you can test the setup as follows:

1. Open the [Amazon SQS console](#).
2. Choose the name of the queue that you created earlier.
3. Choose **Send and receive messages**.
4. Under **Message body**, enter a test message.
5. Choose **Send message**.

Lambda polls the queue for updates. When there is a new message, Lambda invokes your function with this new event data from the queue. Your function runs and creates logs in Amazon CloudWatch. You can view the logs in the [CloudWatch console](#).

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.

4. Enter the name of the role in the text input field and choose **Delete**.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

To delete the Amazon SQS queue

1. Sign in to the AWS Management Console and open the Amazon SQS console at <https://console.aws.amazon.com/sqs/>.
2. Select the queue you created.
3. Choose **Delete**.
4. Enter **delete** in the text input field.
5. Choose **Delete**.

Tutorial: Using a cross-account Amazon SQS queue as an event source

In this tutorial, you create a Lambda function that consumes messages from an Amazon Simple Queue Service (Amazon SQS) queue in a different AWS account. This tutorial involves two AWS accounts: **Account A** refers to the account that contains your Lambda function, and **Account B** refers to the account that contains the Amazon SQS queue.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console \(p. 4\)](#) to create your first Lambda function.

To complete the following steps, you need the [AWS Command Line Interface \(AWS CLI\) version 2](#). Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.0.57 Python/3.7.4 Darwin/19.6.0 exe/x86_64
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#). Example CLI commands

in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

Create the execution role (Account A)

In **Account A**, create an [execution role \(p. 816\)](#) that gives your function permission to access the required AWS resources.

To create an execution role

1. Open the [Roles page](#) in the AWS Identity and Access Management (IAM) console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – AWS Lambda**
 - **Permissions – AWSLambdaSQSQueueExecutionRole**
 - **Role name – cross-account-lambda-sqs-role**

The **AWSLambdaSQSQueueExecutionRole** policy has the permissions that the function needs to read items from Amazon SQS and to write logs to Amazon CloudWatch Logs.

Create the function (Account A)

In **Account A**, create a Lambda function that processes your Amazon SQS messages. The following Node.js 12 code example writes each message to a log in CloudWatch Logs.

Note

For code examples in other languages, see [Sample Amazon SQS function code \(p. 795\)](#).

Example index.js

```
exports.handler = async function(event, context) {  
    event.Records.forEach(record => {  
        const { body } = record;  
        console.log(body);  
    });  
    return {};  
}
```

To create the function

Note

Following these steps creates a function in Node.js 12. For other languages, the steps are similar, but some details are different.

1. Save the code example as a file named `index.js`.
2. Create a deployment package.

```
zip function.zip index.js
```

3. Create the function using the `create-function` AWS Command Line Interface (AWS CLI) command.

```
aws lambda create-function --function-name CrossAccountSQSExample \  
--zip-file fileb://function.zip --handler index.handler --runtime nodejs12.x \  
--role arn:aws:iam::<AccountA_ID>:role/cross-account-lambda-sqs-role
```

Test the function (Account A)

In **Account A**, test your Lambda function manually using the `invoke` AWS CLI command and a sample Amazon SQS event.

If the handler returns normally without exceptions, Lambda considers the message to be successfully processed and begins reading new messages in the queue. After successfully processing a message, Lambda automatically deletes it from the queue. If the handler throws an exception, Lambda considers the batch of messages not successfully processed, and Lambda invokes the function with the same batch of messages.

1. Save the following JSON as a file named `input.txt`.

```
{  
    "Records": [  
        {  
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",  
            "receiptHandle": "AQEBwJnKyHigUMZj6rYigCgxlAS3SLy0a...",  
            "body": "test",  
            "attributes": {  
                "ApproximateReceiveCount": "1",  
                "SentTimestamp": "1545082649183",  
                "SenderId": "AIDAENQZJ0L023YVJ4V0",  
                "ApproximateFirstReceiveTimestamp": "1545082649185"  
            },  
            "messageAttributes": {},  
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",  
            "eventSource": "aws:sqs",  
            "eventSourceARN": "arn:aws:sqs:us-east-1:123456789012:example-queue",  
            "awsRegion": "us-east-1"  
        }  
    ]  
}
```

The preceding JSON simulates an event that Amazon SQS might send to your Lambda function, where "body" contains the actual message from the queue.

2. Run the following `invoke` AWS CLI command.

```
aws lambda invoke --function-name CrossAccountSQSExample \  
--payload file://input.txt outputfile.txt
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

3. Verify the output in the file `outputfile.txt`.

Create an Amazon SQS queue (Account B)

In **Account B**, create an Amazon SQS queue that the Lambda function in **Account A** can use as an event source.

To create a queue

1. Open the [Amazon SQS console](#).
2. Choose **Create queue**.
3. Create a queue with the following properties.

- **Type – Standard**
- **Name – LambdaCrossAccountQueue**
- **Configuration** – Keep the default settings.
- **Access policy** – Choose **Advanced**. Paste in the following JSON policy:

```
{  
    "Version": "2012-10-17",  
    "Id": "Queue1_Policy_UUID",  
    "Statement": [  
        {  
            "Sid": "Queue1_AllActions",  
            "Effect": "Allow",  
            "Principal": {  
                "AWS": [  
                    "arn:aws:iam::<AccountA_ID>:role/cross-account-lambda-sqs-role"  
                ]  
            },  
            "Action": "sns:*",  
            "Resource": "arn:aws:sns:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue"  
        }  
    ]  
}
```

This policy grants the Lambda execution role in **Account A** permissions to consume messages from this Amazon SQS queue.

4. After creating the queue, record its Amazon Resource Name (ARN). You need this in the next step when you associate the queue with your Lambda function.

Configure the event source (Account A)

In **Account A**, create an event source mapping between the Amazon SQS queue in **Account B** and your Lambda function by running the following `create-event-source-mapping` AWS CLI command.

```
aws lambda create-event-source-mapping --function-name CrossAccountSQSExample --batch-size  
10 \  
--event-source-arn arn:aws:sns:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue
```

To get a list of your event source mappings, run the following command.

```
aws lambda list-event-source-mappings --function-name CrossAccountSQSExample \  
--event-source-arn arn:aws:sns:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue
```

Test the setup

You can now test the setup as follows:

1. In **Account B**, open the [Amazon SQS console](#).
2. Choose **LambdaCrossAccountQueue**, which you created earlier.
3. Choose **Send and receive messages**.
4. Under **Message body**, enter a test message.
5. Choose **Send message**.

Your Lambda function in **Account A** should receive the message. Lambda will continue to poll the queue for updates. When there is a new message, Lambda invokes your function with this new event data from

the queue. Your function runs and creates logs in Amazon CloudWatch. You can view the logs in the [CloudWatch console](#).

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

In **Account A**, clean up your execution role and Lambda function.

To delete the execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose **Actions, Delete**.
4. Type **delete** in the text input field and choose **Delete**.

In **Account B**, clean up the Amazon SQS queue.

To delete the Amazon SQS queue

1. Sign in to the AWS Management Console and open the Amazon SQS console at <https://console.aws.amazon.com/sqs/>.
2. Select the queue you created.
3. Choose **Delete**.
4. Enter **delete** in the text input field.
5. Choose **Delete**.

Sample Amazon SQS function code

Sample code is available for the following languages.

Topics

- [Node.js \(p. 795\)](#)
- [Java \(p. 796\)](#)
- [C# \(p. 796\)](#)
- [Go \(p. 797\)](#)
- [Python \(p. 798\)](#)

Node.js

The following is example code that receives an Amazon SQS event message as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

Example index.js (Node.js 12)

```
exports.handler = async function(event, context) {
  event.Records.forEach(record => {
    const { body } = record;
    console.log(body);
  });
  return {};
}
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Node.js Lambda functions with .zip file archives \(p. 263\)](#).

Java

The following is example Java code that receives an Amazon SQS event message as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

In the code, handleRequest is the handler. The handler uses the predefined SQSEvent class that is defined in the aws-lambda-java-events library.

Example Handler.java

```
package example;

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 Handler implements RequestHandler<SQSEvent, Void>{
    @Override
    public Void handleRequest(SQSEvent event, Context context)
    {
        for(SQSMessage msg : event.getRecords()){
            System.out.println(new String(msg.getBody()));
        }
        return null;
    }
}
```

Dependencies

- aws-lambda-java-core
- aws-lambda-java-events

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [Deploy Java Lambda functions with .zip or JAR file archives \(p. 393\)](#).

C#

The following is example C# code that receives an Amazon SQS event message as input and processes it. For illustration, the code writes some of the incoming event data to the console.

In the code, handleRequest is the handler. The handler uses the predefined SQSEvent class that is defined in the AWS.Lambda.SQSEvents library.

Example ProcessingSQSRecords.cs

```
[assembly: LambdaSerializer(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]

namespace SQSLambdaFunction
{
    public class SQSLambdaFunction
    {
        public string HandleSQSEvent(SQSEvent sqsEvent, ILambdaContext context)
        {
            Console.WriteLine($"Beginning to process {sqsEvent.Records.Count} records...");

            foreach (var record in sqsEvent.Records)
            {
                Console.WriteLine($"Message ID: {record.MessageId}");
                Console.WriteLine($"Event Source: {record.EventSource}");

                Console.WriteLine($"Record Body:");
                Console.WriteLine(record.Body);
            }

            Console.WriteLine("Processing complete.");
            return $"Processed {sqsEvent.Records.Count} records.";
        }
    }
}
```

Replace the `Program.cs` in a .NET Core project with the above sample. For instructions, see [Deploy C# Lambda functions with .zip file archives \(p. 497\)](#).

Go

The following is example Go code that receives an Amazon SQS event message as input and processes it. For illustration, the code writes some of the incoming event data to CloudWatch Logs.

In the code, `handler` is the handler. The handler uses the predefined `SQSEvent` class that is defined in the `aws-lambda-go-events` library.

Example ProcessSQSRecords.go

```
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) error {
    for _, message := range sqsEvent.Records {
        fmt.Printf("The message %s for event source %s = %s \n",
            message.MessageId,
            message.EventSource, message.Body)
    }
    return nil
}

func main() {
    lambda.Start(handler)
```

}

Build the executable with `go build` and create a deployment package. For instructions, see [Deploy Go Lambda functions with .zip file archives \(p. 460\)](#).

Python

The following is example Python code that accepts an Amazon SQS record as input and processes it. For illustration, the code writes to some of the incoming event data to CloudWatch Logs.

Example ProcessSQSRecords.py

```
from __future__ import print_function

def lambda_handler(event, context):
    for record in event['Records']:
        print("test")
        payload = record["body"]
        print(str(payload))
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Python Lambda functions with .zip file archives \(p. 324\)](#).

AWS SAM template for an Amazon SQS application

You can build this application using [AWS SAM](#). To learn more about creating AWS SAM templates, see [AWS SAM template basics](#) in the *AWS Serverless Application Model Developer Guide*.

Below is a sample AWS SAM template for the Lambda application from the [tutorial \(p. 787\)](#). Copy the text below to a .yaml file and save it next to the ZIP package you created previously. Note that the Handler and Runtime parameter values should match the ones you used when you created the function in the previous section.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Example of processing messages on an SQS queue with Lambda
Resources:
  MySQSQueueFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs12.x
    Events:
      MySQSEvent:
        Type: SQS
        Properties:
          Queue: !GetAtt MySqsQueue.Arn
          BatchSize: 10
  MySqsQueue:
    Type: AWS::SQS::Queue
```

For information on how to package and deploy your serverless application using the package and deploy commands, see [Deploying serverless applications](#) in the *AWS Serverless Application Model Developer Guide*.

Using AWS Lambda with Amazon VPC Lattice

You can register your Lambda functions as targets within an [Amazon VPC Lattice service network](#). By doing this, your Lambda function becomes a service within the network, and clients that have access to the VPC Lattice service network can call your service. If your Lambda function needs to access services within a service network, you can connect your function to a VPC that's already associated with the service network. Having your services live within a VPC Lattice network can help you more easily discover, connect, access, and monitor them.

Topics

- [VPC Lattice concepts \(p. 799\)](#)
- [Prerequisites and permissions \(p. 800\)](#)
- [Limitations \(p. 801\)](#)
- [Registering your Lambda function with a VPC Lattice network \(p. 801\)](#)
- [Updating the target of a service in a VPC Lattice network \(p. 803\)](#)
- [Deregistering a Lambda function target \(p. 804\)](#)
- [Cross-account networking \(p. 804\)](#)
- [Receiving events from VPC Lattice \(p. 805\)](#)
- [Sending responses back to VPC Lattice \(p. 805\)](#)
- [Monitoring a service in a VPC Lattice network \(p. 806\)](#)

VPC Lattice concepts

Throughout this guide, we'll frequently refer to the following [VPC Lattice terms](#):

- **Service** – A service is any software application that can run on an instance, container, or within a serverless function. This topic focuses only on services built using Lambda functions.
- **Service network** – A service network is a logical boundary containing a network of services. This topic covers how to configure your Lambda functions as services within a VPC Lattice service network.
- **Target group** – A target group is a collection of compute type destinations that run a service. Target groups for Lambda can contain only one Lambda function as the target. You cannot have a target group with multiple functions as targets.
- **Listener** – A listener is a process that receives traffic and routes it to different target groups within the service network.
- **Listener rule** – A listener rule encompasses the priority, actions, and conditions that a listener uses to determine where to route traffic. Each listener has a default listener rule, and a listener can have multiple listener rules. A listener rule can contain the following:
 - **Protocol** – The protocol that the listener uses to send the request to the destination. Can be either HTTP or HTTPS.
 - **Port** – The port that the listener polls for incoming requests. Can be between 1 and 65535 inclusive.
 - **Path** – The path to the target resource. For the default listener rule, path is the default path of /. A listener rule can have 6 paths at maximum, including the default path.
 - **Priority** – A listener uses the priority of a path to determine which path to route traffic to. A lower number denotes higher priority. The default path has the lowest priority. If you add a new path, VPC Lattice assigns it at the second lowest priority by default. This is just above the priority of the default path, but at a lower priority than all other non-default paths.

In addition, we'll refer to the following AWS Identity and Access Management (IAM) entities:

- **Network owner** – The network owner is the IAM role that owns the VPC Lattice service network.

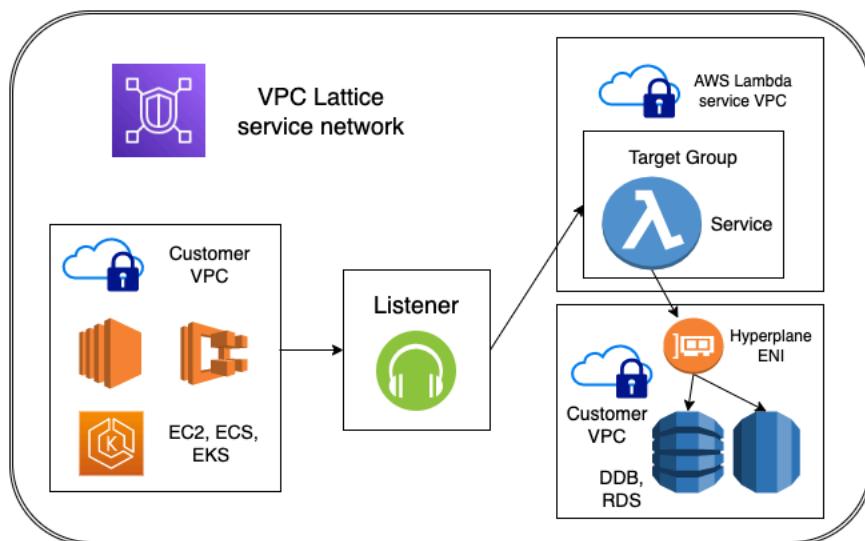
- **Service owner** – The service owner is the IAM role that owns the service built using a Lambda function. The service owner and the network owner do not have to be the same entity.

Using Lambda with VPC Lattice and VPC-enabled functions

By default, Lambda functions cannot access any private resources in a VPC. When you configure a Lambda function to connect to private subnets in a VPC, you're allowing the VPC-enabled function to access resources within that VPC. In other words, you're only focusing on the scope of a single VPC.

Lambda functions registered as services in a VPC Lattice network are not the same as VPC-enabled functions, but they can be complementary to each other. When you register a Lambda function as a service in a VPC Lattice network, you're creating an ingress path to your function from one or more VPCs. Additionally, your function may have an optional egress path to another VPC.

When registering a Lambda function as a service, you're focusing on the ingress scenario. This encompasses the specific listener rule configurations that listeners use to route traffic to your service. From there, your Lambda function may communicate with other AWS services within your VPC through a [Hyperplane ENI \(elastic network interface\)](#). The following diagram illustrates this, where the Lambda function is a service within the VPC Lattice network.



Prerequisites and permissions

This topic assumes that you have both a VPC Lattice service network and a Lambda function. If you don't already have a VPC Lattice service network, refer to the [VPC Lattice user guide to create one](#).

When you register your Lambda function as a target via the Lambda console or AWS Command Line Interface (AWS CLI), Lambda automatically adds the necessary permissions for you.

For Lambda to automatically add permissions for you, so that you can successfully create a Lambda function as a target of a Lambda service, your role must have the following IAM permissions:

- [**AddPermission**](#)
- [**CreateListener**](#)
- [**CreateService**](#)
- [**CreateServiceNetworkServiceAssociation**](#)
- [**CreateTargetGroup**](#)

- **ListServiceNetworks** (required for console workflow only)
- **RegisterTargets**

To update an existing service in a VPC Lattice network to point to a Lambda function, your role must have the following IAM permissions:

- **CreateService**
- **ListListeners** (required for console workflow only)
- **ListServices** (required for console workflow only)
- **RegisterTargets**

You can also manually add the permission using the following AWS CLI command:

```
aws lambda add-permission \
--function-name my-function \
--action lambda:InvokeFunction \
--statement-id allow-vpc-lattice \
--principal vpc-lattice.amazonaws.com
--source-arn target-group-arn
```

Limitations

When using Lambda functions with VPC Lattice networks, keep in mind the following limitations:

- The Lambda function and the target group must be in the same account and in the same Region.
- The maximum size of the request body that you can forward to a service created using a Lambda function is 6 MB.
- The maximum size of the response JSON that the Lambda function can return is 6 MB.
- You can choose HTTP and HTTPS protocols only.

Registering your Lambda function with a VPC Lattice network

You can register any existing Lambda function with a VPC Lattice network using the AWS console or AWS CLI. After registering the function as a service, it can immediately start receiving requests.

To register a Lambda function with a VPC Lattice network (console)

1. Open the [Function page](#) of the Lambda console.
2. Choose the name of the function you want to register.
3. Under **Function overview**, choose **Add trigger**.
4. In the dropdown menu, select **VPC Lattice Application Network**.
5. For **Intent**, choose **Create new**.
6. For **Service name**, enter a name for your service.
7. For **VPC Lattice network**, select the service network that you want to associate this Lambda function with. You can also enter the full Amazon Resource Name (ARN) of the service network. If you don't have an existing service network, you can choose the link in the description, which takes you to the Amazon VPC console where you can create a VPC Lattice network.

Note

You don't need to select a VPC Lattice network to finish setting up the trigger. However, if you create the trigger without selecting a network, clients cannot access your service until you associate it with a VPC Lattice network.

8. For **Listener**, configure the following settings:
 - **Listener name** – Enter a name for your listener
 - **Protocol** – The protocol that the listener uses to send the request to the destination. Can be either HTTP or HTTPS.
 - **Port** – The port that the listener polls for incoming requests. Can be between 1 and 65535 inclusive.
 - Choose **Add Listener**. VPC Lattice will create a default routing with the default path /. After service creation, you cannot change its name, protocol, and port settings, but you can still define new routing paths.
9. Choose **Add**.

To register a Lambda function with a VPC Lattice network (AWS CLI)

1. Create a service using the `create-service` command. Note down the service's ARN in the response. You'll need it in the next steps.

```
aws vpc-lattice create-service --name my-vpc-lattice-service
```

2. Create a target group using the `create-target-group` command. Note down the target group's ARN in the response. You'll need it in the next steps.

```
aws vpc-lattice create-target-group \
--name my-vpc-lattice-target-group \
--type LAMBDA
```

3. Create a listener within the service network using the `create-listener` command. New listeners automatically use the default path of /. Replace the value of the `service` parameter with your service's ARN from step 1. Replace the value of `TargetGroupArn` in the `default-action` parameter with your target group ARN from step 2.

```
aws vpc-lattice create-listener \
--name https \
--service-identifier svc-0e2f2665e1cebb720 \
--protocol HTTPS \
--default-action \
forward='{targetGroups={targetGroupIdentifier=tg-0e2f2665e1cebb720}}'
```

4. Register your Lambda function as a target using the `register-targets` command. Replace the value of the `target-group-arn` parameter with your target group ARN from step 2. Replace the value of `Id` in the `targets` parameter with the ARN of your Lambda function.

```
aws vpc-lattice register-targets \
--target-group-identifier arn:aws:vpc-lattice:us-west-2:123456789012:targetgroup/tg-0e2f2665e1cebb720 \
--targets id=arn:aws:lambda:us-west-2:123456789012:function:my-function
```

Note

In the previous `register-targets` command, if your Lambda function doesn't already explicitly allow VPC Lattice to invoke it, VPC Lattice attaches the necessary permissions to

your function's execution role automatically. To allow VPC Lattice to automatically attach permissions, your role needs to have the [AddPermission](#) permission.

5. Associate the service with the service network using the `create-service-network-service-association` command. Replace the value of the `service` parameter with your service ARN from step 1. Replace the value of the `service-network` parameter with the ARN of your VPC Lattice service network.

```
aws vpc-lattice create-service-network-service-association \
  --service-identifier arn:aws:vpc-lattice:us-west-2:123456789012:service/
  svc-0b9b89d907bc8668c \
  --service-network-identifier arn:aws:vpc-lattice:us-
  west-2:123456789012:servicenetwork/03d622a31e5154247
```

Updating the target of a service in a VPC Lattice network

You can update any existing service within a VPC Lattice network to point to a Lambda function target. You can do this by adding a new routing path using the AWS console or AWS CLI. When you add a new path, VPC Lattice assigns that path the second lowest priority, just above the default path (lowest priority).

To update the target of a service to point to a Lambda function (console)

1. Open the [Function page](#) of the Lambda console.
2. Choose the name of the function you want to register.
3. Under **Function overview**, choose **Add trigger**.
4. In the dropdown menu, select **VPC Lattice Application Network**.
5. For **Intent**, choose **Select existing**.
6. For **Service name**, choose an existing service.
7. For **Listener**, choose an existing listener.
8. For **Rule name**, enter a name for the new rule.
9. For **Path**, define a new routing path for the listener.
10. Choose **Add**.

For the following steps, you'll need the ARN of your service, as well as the ARN of the listener that you want to add a new rule for.

1. Create a new target group using the `create-target-group` command. Note down the target group's ARN in the response, as you'll need it for future steps.

```
aws vpc-lattice create-target-group \
  --name my-vpc-lattice-target-group \
  --type LAMBDA
```

2. Create a new rule for your existing listener using the `create-rule` command. This command assumes that your conditions are in a file called `conditions-pattern.json` in your current directory. Replace the value of the `listener-arn` parameter with the ARN of your listener. Replace the value of `TargetGroupArn` in the `actions` parameter with your target group ARN from step 1.

```
aws vpc-lattice create-rule \
  --name my-rule
  --priority 1 \
```

```
--listener-identifier listener-0e9af499f72e5251b \
--service-identifier svc-01755f67d3a427803
--match httpMatch='{pathMatch={match={prefix="/test"}}}'
--default action
forward='{targetGroups=[{targetGroupIdentifier=tg-042d5b70f1e743940}]}'
```

3. Register your Lambda function as a target using the `register-targets` command. Replace the value of the `target-group-arn` parameter with your target group ARN from step 2. Replace the value of `Id` in the `targets` parameter with the ARN of your Lambda function.

```
aws vpc-lattice register-targets \
  --target-group-identifier arn:aws:vpc-lattice:us-west-2:123456789012:targetgroup/
tg-0e2f2665e1cebb720 \
  --targets id=arn:aws:lambda:us-west-2:123456789012:function:my-new-function
```

Deregistering a Lambda function target

To deregister a Lambda function target in a VPC Lattice network, use the VPC console. For more information, see the [VPC Lattice user guide](#).

Alternatively, you can use the following AWS CLI command:

```
aws vpc-lattice deregister-targets \
  --target-group-identifier arn:aws:vpc-lattice:us-west-2:123456789012:targetgroup/
tg-0e2f2665e1cebb720 \
  --targets id=arn:aws:lambda:us-west-2:123456789012:function:my-new-function
```

You cannot deregister a service built using Lambda functions from the Lambda console.

Cross-account networking

Services within your VPC Lattice service network don't have to all be in the same AWS account. In addition, the VPC Lattice service network itself can reside in a different account. This means that you can associate a service built using Lambda functions with a service network in a different AWS account. You'll need specific permissions from the network owner to create these service associations. For more information about permissions required, see [Prepare to call the VPC Lattice API](#) in the Amazon VPC Lattice user guide.

You can create an association between a Lambda function with a service network in a different account through the AWS console. To do this, instead of choosing the **VPC Lattice network** from the dropdown menu (this only displays networks in your account), paste the full ARN of the network.

In general, you can create an association between any service and any service network with the `create-service-network-service-association` AWS CLI command. This means you can manage your service networks in a central account and have services built using Lambda functions in other accounts across your AWS organization. In the following example, note that the service and service-network live in two different accounts:

```
aws vpc-lattice create-service-network-service-association \
  --service arn:aws:vpc-lattice:us-west-2:123456789012:service/svc-0b9b89d907bc8668c \
  --service-network arn:aws:vpc-lattice:us-
west-2:444455556666:servicenetwork/03d622a31e5154247
```

Receiving events from VPC Lattice

The VPC Lattice service routes Lambda invocation requests over HTTP and HTTPS. The following is an example event that your Lambda function might receive from VPC Lattice, in JSON format:

```
{  
    "body": "request_body",  
    "headers": {  
        "accept": "*/*",  
        "content-length": "9",  
        "user-agent": "FooClient/1.0",  
        "host": "foo.example.org",  
        "x-foo": "bar,baz",  
        "x-forwarded-for": "192.0.2.1"  
    },  
    "is_base64_encoded": false,  
    "method": "POST",  
    "query_string_parameters": {  
        "foo": "bar"  
    },  
    "raw_path": "/foo/bar/baz?foo=bar&foo=baz"  
}
```

If the content-encoding header is not present, Base64 encoding depends on the content type. For the content types, text/*, application/json, application/xml, and application/javascript, the service sends the body as is and sets isBase64Encoded to false.

Note

The request body can have a maximum size of 1023 KiB if sent in plaintext, or 767 KiB if base64 encoded. The list of request headers can have a maximum of 50 key-value pairs.

Sending responses back to VPC Lattice

When you send a response from your Lambda function back to VPC Lattice, the response must include the Base64 encoding status, status code, and relevant headers. The body is optional. The following is an example response in JSON format:

```
{  
    "isBase64Encoded": false,  
    "statusCode": 200,  
    "statusDescription": "200 OK",  
    "headers": {  
        "Set-Cookie": "cookies",  
        "Content-Type": "application/json"  
    },  
    "body": "Hello from Lambda (optional)"  
}
```

To include binary content in the response body, you must Base64 encode the content and set isBase64Encoded to true. This tells VPC Lattice to decode the content before sending the response to the client.

VPC Lattice doesn't honor hop-by-hop headers such as Connection or Transfer-Encoding. Also, you can omit the Content-Length header because VPC Lattice automatically computes it before sending responses to clients.

Note

The response body can have a maximum size of 1023 KiB if sent in plaintext, or 767 KiB if base64 encoded. The list of response headers can have a maximum of 50 key-value pairs.

Monitoring a service in a VPC Lattice network

To monitor services built using Lambda functions in a VPC Lattice network, VPC Lattice provides Amazon CloudWatch metrics, AWS CloudTrail logs, and access logs. These tools can help you track key performance metrics such as the total number of requests to your service and the number of connection timeouts.

By default, Lambda automatically emits metrics to CloudWatch and event history logs to CloudTrail. Access logs are optional, and Lambda deactivates them by default. For more information on monitoring, see [Monitoring Amazon VPC Lattice](#) in the VPC Lattice user guide.

Using AWS Lambda with AWS X-Ray

You can use AWS X-Ray to visualize the components of your application, identify performance bottlenecks, and troubleshoot requests that resulted in an error. Your Lambda functions send trace data to X-Ray, and X-Ray processes the data to generate a service map and searchable trace summaries.

If you've enabled X-Ray tracing in a service that invokes your function, Lambda sends traces to X-Ray automatically. The upstream service, such as Amazon API Gateway, or an application hosted on Amazon EC2 that is instrumented with the X-Ray SDK, samples incoming requests and adds a tracing header that tells Lambda to send traces or not. Traces from upstream message producers, such as Amazon SQS, are automatically linked to traces from downstream Lambda functions, creating an end-to-end view of the entire application. For more information, see [Tracing event-driven applications](#) in the [AWS X-Ray Developer Guide](#).

Note

X-Ray tracing is currently not supported for Lambda functions with Amazon Managed Streaming for Apache Kafka (Amazon MSK), self-managed Apache Kafka, or Amazon MQ with ActiveMQ and RabbitMQ event source mappings.

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring and operations tools**.
4. Choose **Edit**.
5. Under **X-Ray**, toggle on **Active tracing**.
6. Choose **Save**.

Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see [AWS X-Ray pricing](#).

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's [execution role \(p. 816\)](#). Otherwise, add the [AWSXRayDaemonWriteAccess](#) policy to the execution role.

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests.

Note

You cannot configure the X-Ray sampling rate for your functions.

In X-Ray, a *trace* records information about a request that is processed by one or more services. Services record *segments* that contain layers of *subsegments*. Lambda records a segment for the Lambda service that handles the invocation request, and one for the work done by the function. The function segment comes with subsegments for Initialization, Invocation and Overhead. For more information see [Lambda execution environment lifecycle \(p. 14\)](#).

Note

X-Ray treats unhandled exceptions in your Lambda function as Error statuses. X-Ray records Fault statuses only when Lambda experiences internal server errors. For more information, see [Errors, faults, and exceptions](#) in the X-Ray Developer Guide.

The Initialization subsegment represents the init phase of the Lambda execution environment lifecycle. During this phase, Lambda creates or unfreezes an execution environment with the resources you have configured, downloads the function code and all layers, initializes extensions, initializes the runtime, and runs the function's initialization code.

The Invocation subsegment represents the invoke phase where Lambda invokes the function handler. This begins with runtime and extension registration and it ends when the runtime is ready to send the response.

The Overhead subsegment represents the phase that occurs between the time when the runtime sends the response and the signal for the next invoke. During this time, the runtime finishes all tasks related to an invoke and prepares to freeze the sandbox.

Note

Occasionally, you may notice a large gap between the function initialization and invocation phases in your X-Ray traces. For functions using [provisioned concurrency \(p. 213\)](#), this is because Lambda initializes your function instances well in advance of invocation. For functions using [unreserved \(on-demand\) concurrency \(p. 197\)](#), Lambda may proactively initialize a function instance, even if there's no invocation. Visually, both of these cases show up as a time gap between the initialization and invocation phases.

Important

In Lambda, you can use the X-Ray SDK to extend the Invocation subsegment with additional subsegments for downstream calls, annotations, and metadata. You can't access the function segment directly or record work done outside of the handler invocation scope.

See the following topics for a language-specific introduction to tracing in Lambda:

- [Instrumenting Node.js code in AWS Lambda \(p. 282\)](#)
- [Instrumenting Python code in AWS Lambda \(p. 350\)](#)
- [Instrumenting Ruby code in AWS Lambda \(p. 381\)](#)
- [Instrumenting Java code in AWS Lambda \(p. 434\)](#)
- [Instrumenting Go code in AWS Lambda \(p. 482\)](#)
- [Instrumenting C# code in AWS Lambda \(p. 519\)](#)

For a full list of services that support active instrumentation, see [Supported AWS services](#) in the AWS X-Ray Developer Guide.

Sections

- [Execution role permissions \(p. 808\)](#)
- [The AWS X-Ray daemon \(p. 809\)](#)
- [Enabling active tracing with the Lambda API \(p. 809\)](#)
- [Enabling active tracing with AWS CloudFormation \(p. 809\)](#)

Execution role permissions

Lambda needs the following permissions to send trace data to X-Ray. Add them to your function's [execution role \(p. 816\)](#).

- [xray:PutTraceSegments](#)
- [xray:PutTelemetryRecords](#)

These permissions are included in the [AWSXRayDaemonWriteAccess](#) managed policy.

The AWS X-Ray daemon

Instead of sending trace data directly to the X-Ray API, the X-Ray SDK uses a daemon process. The AWS X-Ray daemon is an application that runs in the Lambda environment and listens for UDP traffic that contains segments and subsegments. It buffers incoming data and writes it to X-Ray in batches, reducing the processing and memory overhead required to trace invocations.

The Lambda runtime allows the daemon to up to 3 percent of your function's configured memory or 16 MB, whichever is greater. If your function runs out of memory during invocation, the runtime terminates the daemon process first to free up memory.

The daemon process is fully managed by Lambda and cannot be configured by the user. All segments generated by function invocations are recorded in the same account as the Lambda function. The daemon cannot be configured to redirect them to any other account.

For more information, see [The X-Ray daemon](#) in the X-Ray Developer Guide.

Enabling active tracing with the Lambda API

To manage tracing configuration with the AWS CLI or AWS SDK, use the following API operations:

- [UpdateFunctionConfiguration \(p. 1377\)](#)
- [GetFunctionConfiguration \(p. 1229\)](#)
- [CreateFunction \(p. 1165\)](#)

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Enabling active tracing with AWS CloudFormation

To activate tracing on an AWS::Lambda::Function resource in an AWS CloudFormation template, use the **TracingConfig** property.

Example [function-inline.yml](#) – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...

```

For an AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function resource, use the **Tracing** property.

Example [template.yml](#) – Tracing configuration

```
Resources:
```

```
function:  
  Type: AWS::Serverless::Function  
Properties:  
  Tracing: Active  
  ...
```

Best practices for working with AWS Lambda functions

The following are recommended best practices for using AWS Lambda:

Topics

- [Function code \(p. 811\)](#)
- [Function configuration \(p. 812\)](#)
- [Metrics and alarms \(p. 813\)](#)
- [Working with streams \(p. 813\)](#)
- [Security best practices \(p. 814\)](#)

For more information about best practices for Lambda applications, see [Application design](#) in the *Lambda operator guide*.

Function code

- **Separate the Lambda handler from your core logic.** This allows you to make a more unit-testable function. In Node.js this may look like:

```
exports.myHandler = function(event, context, callback) {
  var foo = event.foo;
  var bar = event.bar;
  var result = MyLambdaFunction (foo, bar);

  callback(null, result);
}

function MyLambdaFunction (foo, bar) {
  // MyLambdaFunction logic here
}
```

- **Take advantage of execution environment reuse to improve the performance of your function.** Initialize SDK clients and database connections outside of the function handler, and cache static assets locally in the /tmp directory. Subsequent invocations processed by the same instance of your function can reuse these resources. This saves cost by reducing function run time.

To avoid potential data leaks across invocations, don't use the execution environment to store user data, events, or other information with security implications. If your function relies on a mutable state that can't be stored in memory within the handler, consider creating a separate function or separate versions of a function for each user.

- **Use a keep-alive directive to maintain persistent connections.** Lambda purges idle connections over time. Attempting to reuse an idle connection when invoking a function will result in a connection error. To maintain your persistent connection, use the keep-alive directive associated with your runtime. For an example, see [Reusing Connections with Keep-Alive in Node.js](#).
- **Use [environment variables \(p. 76\)](#) to pass operational parameters to your function.** For example, if you are writing to an Amazon S3 bucket, instead of hard-coding the bucket name you are writing to, configure the bucket name as an environment variable.

- **Control the dependencies in your function's deployment package.** The AWS Lambda execution environment contains a number of libraries such as the AWS SDK for the Node.js and Python runtimes (a full list can be found here: [Lambda runtimes \(p. 37\)](#)). To enable the latest set of features and security updates, Lambda will periodically update these libraries. These updates may introduce subtle changes to the behavior of your Lambda function. To have full control of the dependencies your function uses, package all of your dependencies with your deployment package.
- **Minimize your deployment package size to its runtime necessities.** This will reduce the amount of time that it takes for your deployment package to be downloaded and unpacked ahead of invocation. For functions authored in Java or .NET Core, avoid uploading the entire AWS SDK library as part of your deployment package. Instead, selectively depend on the modules which pick up components of the SDK you need (e.g. DynamoDB, Amazon S3 SDK modules and [Lambda core libraries](#)).
- **Reduce the time it takes Lambda to unpack deployment packages** authored in Java by putting your dependency .jar files in a separate /lib directory. This is faster than putting all your function's code in a single jar with a large number of .class files. See [Deploy Java Lambda functions with .zip or JAR file archives \(p. 393\)](#) for instructions.
- **Minimize the complexity of your dependencies.** Prefer simpler frameworks that load quickly on [execution environment \(p. 14\)](#) startup. For example, prefer simpler Java dependency injection (IoC) frameworks like [Dagger](#) or [Guice](#), over more complex ones like [Spring Framework](#).
- **Avoid using recursive code** in your Lambda function, wherein the function automatically calls itself until some arbitrary criteria is met. This could lead to unintended volume of function invocations and escalated costs. If you do accidentally do so, set the function reserved concurrency to 0 immediately to throttle all invocations to the function, while you update the code.
- **Do not use non-documented, non-public APIs** in your Lambda function code. For AWS Lambda managed runtimes, Lambda periodically applies security and functional updates to Lambda's internal APIs. These internal API updates may be backwards-incompatible, leading to unintended consequences such as invocation failures if your function has a dependency on these non-public APIs. See [the API reference \(p. 1135\)](#) for a list of publicly available APIs.
- **Write idempotent code.** Writing idempotent code for your functions ensures that duplicate events are handled the same way. Your code should properly validate events and gracefully handle duplicate events. For more information, see [How do I make my Lambda function idempotent?](#).

Function configuration

- **Performance testing your Lambda function** is a crucial part in ensuring you pick the optimum memory size configuration. Any increase in memory size triggers an equivalent increase in CPU available to your function. The memory usage for your function is determined per-invoke and can be viewed in [Amazon CloudWatch](#). On each invoke a REPORT: entry will be made, as shown below:

```
REPORT RequestId: 3604209a-e9a3-11e6-939a-754dd98c7be3 Duration: 12.34 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 18 MB
```

By analyzing the Max Memory Used: field, you can determine if your function needs more memory or if you over-provisioned your function's memory size.

To find the right memory configuration for your functions, we recommend using the open source AWS Lambda Power Tuning project. For more information, see [AWS Lambda Power Tuning](#) on GitHub.

To optimize function performance, we also recommend deploying libraries that can leverage [Advanced Vector Extensions 2 \(AVX2\)](#). This allows you to process demanding workloads, including machine learning inferencing, media processing, high performance computing (HPC), scientific simulations, and financial modeling. For more information, see [Creating faster AWS Lambda functions with AVX2](#).

- **Load test your Lambda function** to determine an optimum timeout value. It is important to analyze how long your function runs so that you can better determine any problems with a dependency

service that may increase the concurrency of the function beyond what you expect. This is especially important when your Lambda function makes network calls to resources that may not handle Lambda's scaling.

- **Use most-restrictive permissions when setting IAM policies.** Understand the resources and operations your Lambda function needs, and limit the execution role to these permissions. For more information, see [Lambda resource access permissions \(p. 815\)](#).
- **Be familiar with Lambda quotas (p. 1131).** Payload size, file descriptors and /tmp space are often overlooked when determining runtime resource limits.
- **Delete Lambda functions that you are no longer using.** By doing so, the unused functions won't needlessly count against your deployment package size limit.
- **If you are using Amazon Simple Queue Service as an event source,** make sure the value of the function's expected invocation time does not exceed the [Visibility Timeout](#) value on the queue. This applies both to [CreateFunction \(p. 1165\)](#) and [UpdateFunctionConfiguration \(p. 1377\)](#).
 - In the case of [CreateFunction](#), AWS Lambda will fail the function creation process.
 - In the case of [UpdateFunctionConfiguration](#), it could result in duplicate invocations of the function.

Metrics and alarms

- Use [Working with Lambda function metrics \(p. 870\)](#) and [CloudWatch Alarms](#) instead of creating or updating a metric from within your Lambda function code. It's a much more efficient way to track the health of your Lambda functions, allowing you to catch issues early in the development process. For instance, you can configure an alarm based on the expected duration of your Lambda function invocation in order to address any bottlenecks or latencies attributable to your function code.
- Leverage your logging library and [AWS Lambda Metrics and Dimensions](#) to catch app errors (e.g. ERR, ERROR, WARNING, etc.)
- Use [AWS Cost Anomaly Detection](#) to detect unusual activity on your account. Cost Anomaly Detection uses machine learning to continuously monitor your cost and usage while minimizing false positive alerts. Cost Anomaly Detection uses data from AWS Cost Explorer, which has a delay of up to 24 hours. As a result, it can take up to 24 hours to detect an anomaly after usage occurs. To get started with Cost Anomaly Detection, you must first [sign up for Cost Explorer](#). Then, [access Cost Anomaly Detection](#).

Working with streams

- **Test with different batch and record sizes** so that the polling frequency of each event source is tuned to how quickly your function is able to complete its task. The [CreateEventSourceMapping \(p. 1153\)](#) BatchSize parameter controls the maximum number of records that can be sent to your function with each invoke. A larger batch size can often more efficiently absorb the invoke overhead across a larger set of records, increasing your throughput.

By default, Lambda invokes your function as soon as records are available. If the batch that Lambda reads from the event source has only one record in it, Lambda sends only one record to the function. To avoid invoking the function with a small number of records, you can tell the event source to buffer records for up to 5 minutes by configuring a *batching window*. Before invoking the function, Lambda continues to read records from the event source until it has gathered a full batch, the batching window expires, or the batch reaches the payload limit of 6 MB. For more information, see [Batching behavior \(p. 132\)](#).

- **Increase Kinesis stream processing throughput by adding shards.** A Kinesis stream is composed of one or more shards. Lambda will poll each shard with at most one concurrent invocation. For example, if your stream has 100 active shards, there will be at most 100 Lambda function invocations running concurrently. Increasing the number of shards will directly increase the number of maximum concurrent Lambda function invocations and can increase your Kinesis stream processing throughput.

If you are increasing the number of shards in a Kinesis stream, make sure you have picked a good partition key (see [Partition Keys](#)) for your data, so that related records end up on the same shards and your data is well distributed.

- Use [Amazon CloudWatch](#) on IteratorAge to determine if your Kinesis stream is being processed. For example, configure a CloudWatch alarm with a maximum setting to 30000 (30 seconds).

Security best practices

- **Monitor your usage of AWS Lambda as it relates to security best practices by using AWS Security Hub.** Security Hub uses security controls to evaluate resource configurations and security standards to help you comply with various compliance frameworks. For more information about using Security Hub to evaluate Lambda resources, see [AWS Lambda controls](#) in the AWS Security Hub User Guide.

Lambda resource access permissions

You can use AWS Identity and Access Management (IAM) to manage access to the Lambda API and resources such as functions and layers. For users and applications in your account that use Lambda, you can create IAM policies that apply to users, groups, or roles.

Every Lambda function has an IAM role called an [execution role \(p. 816\)](#). In this role, you can attach a policy that defines the permissions that your function needs to access other AWS services and resources. At a minimum, your function needs access to Amazon CloudWatch Logs for log streaming. If your function calls other service APIs with the AWS SDK, you must include the necessary permissions in the execution role's policy. Lambda also uses the execution role to get permission to read from event sources when you use an [event source mapping \(p. 131\)](#) to invoke your function.

To give other accounts and AWS services permission to use your Lambda resources, use a [resource-based policy \(p. 832\)](#). Lambda resources include functions, versions, aliases, and layer versions. When a user tries to access a Lambda resource, Lambda considers both the user's [identity-based policies \(p. 823\)](#) and the resource's resource-based policy. When an AWS service such as Amazon Simple Storage Service (Amazon S3) calls your Lambda function, Lambda considers only the resource-based policy.

To manage permissions for users and applications in your account, we recommend using an [AWS managed policy \(p. 823\)](#). You can use these managed policies as-is, or as a starting point for writing your own more restrictive policies. Policies can restrict user permissions by the resource that an action affects, and by additional optional conditions. For more information, see [Resources and conditions for Lambda actions \(p. 838\)](#).

If your Lambda functions contain calls to other AWS resources, you might also want to restrict which functions can access those resources. To do this, include the `lambda:SourceFunctionArn` condition key in an IAM identity-based policy or service control policy (SCP) for the target resource. For more information, see [Working with Lambda execution environment credentials \(p. 820\)](#).

For more information about IAM, see the [IAM User Guide](#).

For more information about applying security principles to Lambda applications, see [Security](#) in the [AWS Lambda Operator Guide](#).

Topics

- [Lambda execution role \(p. 816\)](#)
- [Identity-based IAM policies for Lambda \(p. 823\)](#)
- [Attribute-based access control for Lambda \(p. 828\)](#)
- [Using resource-based policies for Lambda \(p. 832\)](#)
- [Resources and conditions for Lambda actions \(p. 838\)](#)
- [Using permissions boundaries for AWS Lambda applications \(p. 845\)](#)

Lambda execution role

A Lambda function's execution role is an AWS Identity and Access Management (IAM) role that grants the function permission to access AWS services and resources. For example, you might create an execution role that has permission to send logs to Amazon CloudWatch and upload trace data to AWS X-Ray. This page provides information on how to create, view, and manage a Lambda function's execution role.

You provide an execution role when you create a function. **When you invoke your function, Lambda automatically provides your function with temporary credentials by assuming this role.** You don't have to call `sts:AssumeRole` in your function code.

In order for Lambda to properly assume your execution role, the role's [trust policy \(p. 817\)](#) must specify the Lambda service principal (`lambda.amazonaws.com`) as a trusted service.

To view a function's execution role

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of a function.
3. Choose **Configuration**, and then choose **Permissions**.
4. Under **Resource summary**, review the services and resources that the function can access.
5. Choose a service from the dropdown list to see permissions related to that service.

You can add or remove permissions from a function's execution role at any time, or configure your function to use a different role. Add permissions for any services that your function calls with the AWS SDK, and for services that Lambda uses to enable optional features.

When you add permissions to your function, update its code or configuration as well. This forces running instances of your function, which have outdated credentials, to stop and be replaced.

Topics

- [Creating an execution role in the IAM console \(p. 816\)](#)
- [Grant least privilege access to your Lambda execution role \(p. 817\)](#)
- [Managing roles with the IAM API \(p. 817\)](#)
- [Session duration for temporary security credentials \(p. 818\)](#)
- [AWS managed policies for Lambda features \(p. 818\)](#)
- [Working with Lambda execution environment credentials \(p. 820\)](#)

Creating an execution role in the IAM console

By default, Lambda creates an execution role with minimal permissions when you [create a function in the Lambda console \(p. 4\)](#). You can also create an execution role in the IAM console.

To create an execution role in the IAM console

1. Open the [Roles page](#) in the IAM console.
2. Choose **Create role**.
3. Under **Use case**, choose **Lambda**.
4. Choose **Next**.
5. Select the AWS managed policies **AWSLambdaBasicExecutionRole** and **AWSXRayDaemonWriteAccess**.
6. Choose **Next**.
7. Enter a **Role name** and then choose **Create role**.

For detailed instructions, see [Creating a role for an AWS service \(console\)](#) in the *IAM User Guide*.

Grant least privilege access to your Lambda execution role

When you first create an IAM role for your Lambda function during the development phase, you might sometimes grant permissions beyond what is required. Before publishing your function in the production environment, as a best practice, adjust the policy to include only the required permissions. For more information, see [Apply least-privilege permissions](#) in the *IAM User Guide*.

Use IAM Access Analyzer to help identify the required permissions for the IAM execution role policy. IAM Access Analyzer reviews your AWS CloudTrail logs over the date range that you specify and generates a policy template with only the permissions that the function used during that time. You can use the template to create a managed policy with fine-grained permissions, and then attach it to the IAM role. That way, you grant only the permissions that the role needs to interact with AWS resources for your specific use case.

For more information, see [Generate policies based on access activity](#) in the *IAM User Guide*.

Managing roles with the IAM API

To create an execution role with the AWS Command Line Interface (AWS CLI), use the **create-role** command. When using this command, you can specify the [trust policy](#) inline. A role's trust policy gives the specified principals permission to assume the role. In the following example, you grant the Lambda service principal permission to assume your role. Note that requirements for escaping quotes in the JSON string may vary depending on your shell.

```
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"}]}'
```

You can also define the trust policy for the role using a separate JSON file. In the following example, `trust-policy.json` is a file in the current directory.

Example trust-policy.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
```

You should see the following output:

```
{  
    "Role": {  
        "Path": "/"},
```

```

"RoleName": "lambda-ex",
"RoleId": "AROAQFOXMP6TZ6ITKWND",
"Arn": "arn:aws:iam::123456789012:role/lambda-ex",
"CreateDate": "2020-01-17T23:19:12Z",
"AssumeRolePolicyDocument": {
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
}

```

Note

Lambda automatically assumes your execution role when you invoke your function. You should avoid calling `sts:AssumeRole` manually in your function code. If your use case requires that the role assumes itself, you must include the role itself as a trusted principal in your role's trust policy. For more information on how to modify a role trust policy, see [Modifying a role trust policy \(console\)](#) in the IAM User Guide.

To add permissions to the role, use the **attach-policy-to-role** command. Start by adding the `AWSLambdaBasicExecutionRole` managed policy.

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

Session duration for temporary security credentials

Lambda assumes the execution role associated with your function to fetch temporary security credentials which are then available as environment variables during a function's invocation. If you use these temporary credentials outside of Lambda, such as to create a presigned Amazon S3 URL, you can't control the session duration. The IAM maximum session duration setting doesn't apply to sessions that are assumed by AWS services such as Lambda. Use the [sts:AssumeRole](#) action if you need control over session duration.

AWS managed policies for Lambda features

The following AWS managed policies provide permissions that are required to use Lambda features.

Change	Description	Date
AWSLambdaMSKExecutionRole – Lambda added the kafka:DescribeClusterV2 permission to this policy.	AWSLambdaMSKExecutionRole grants permissions to read and access records from an Amazon Managed Streaming for Apache Kafka (Amazon MSK) cluster, manage elastic network interfaces (ENIs), and write to CloudWatch Logs.	June 17, 2022

Change	Description	Date
AWSLambdaBasicExecutionRole – Lambda started tracking changes to this policy.	AWSLambdaBasicExecutionRole grants permissions to upload logs to CloudWatch.	February 14, 2022
AWSLambdaDynamoDBExecutionRole – Lambda started tracking changes to this policy.	AWSLambdaDynamoDBExecutionRole grants permissions to read records from an Amazon DynamoDB stream and write to CloudWatch Logs.	February 14, 2022
AWSLambdaKinesisExecutionRole – Lambda started tracking changes to this policy.	AWSLambdaKinesisExecutionRole grants permissions to read events from an Amazon Kinesis data stream and write to CloudWatch Logs.	February 14, 2022
AWSLambdaMSKExecutionRole – Lambda started tracking changes to this policy.	AWSLambdaMSKExecutionRole grants permissions to read and access records from an Amazon Managed Streaming for Apache Kafka (Amazon MSK) cluster, manage elastic network interfaces (ENIs), and write to CloudWatch Logs.	February 14, 2022
AWSLambdaSQSQueueExecutionRole – Lambda started tracking changes to this policy.	AWSLambdaSQSQueueExecutionRole grants permissions to read a message from an Amazon Simple Queue Service (Amazon SQS) queue and write to CloudWatch Logs.	February 14, 2022
AWSLambdaVPCAccessExecutionRole – Lambda started tracking changes to this policy.	AWSLambdaVPCAccessExecutionRole grants permissions to manage ENIs within an Amazon VPC and write to CloudWatch Logs.	February 14, 2022
AWSXRayDaemonWriteAccess – Lambda started tracking changes to this policy.	AWSXRayDaemonWriteAccess grants permissions to upload trace data to X-Ray.	February 14, 2022
CloudWatchLambdaInsightsExecutionRolePolicy – Lambda started tracking changes to this policy.	CloudWatchLambdaInsightsExecutionRolePolicy grants permissions to write runtime metrics to CloudWatch Lambda Insights.	February 14, 2022
AmazonS3ObjectLambdaExecutionRolePolicy – Lambda started tracking changes to this policy.	AmazonS3ObjectLambdaExecutionRolePolicy grants permissions to interact with Amazon Simple Storage Service (Amazon S3) object Lambda and to write to CloudWatch Logs.	February 14, 2022

For some features, the Lambda console attempts to add missing permissions to your execution role in a customer managed policy. These policies can become numerous. To avoid creating extra policies, add the relevant AWS managed policies to your execution role before enabling features.

When you use an [event source mapping \(p. 131\)](#) to invoke your function, Lambda uses the execution role to read event data. For example, an event source mapping for Kinesis reads events from a data stream and sends them to your function in batches.

When a service assumes a role in your account, you can include the `aws:SourceAccount` and `aws:SourceArn` global condition context keys in your role trust policy to limit access to the role to only requests that are generated by expected resources. For more information, see [Cross-service confused deputy prevention for AWS Security Token Service](#).

You can use event source mappings with the following services:

Services that Lambda reads events from

- [Amazon DynamoDB \(p. 635\)](#)
- [Amazon Kinesis \(p. 684\)](#)
- [Amazon MQ \(p. 708\)](#)
- [Amazon Managed Streaming for Apache Kafka \(Amazon MSK\) \(p. 716\)](#)
- [Self-managed Apache Kafka \(p. 671\)](#)
- [Amazon Simple Queue Service \(Amazon SQS\) \(p. 778\)](#)
- [Amazon DocumentDB \(with MongoDB compatibility\) \(Amazon DocumentDB\) \(p. 610\)](#)

In addition to the AWS managed policies, the Lambda console provides templates for creating a custom policy with permissions for additional use cases. When you create a function in the Lambda console, you can choose to create a new execution role with permissions from one or more templates. These templates are also applied automatically when you create a function from a blueprint, or when you configure options that require access to other services. Example templates are available in this guide's [GitHub repository](#).

Working with Lambda execution environment credentials

It's common for your Lambda function code to make API requests to other AWS services. To make these requests, Lambda generates an ephemeral set of credentials by assuming your function's execution role. These credentials are available as environment variables during your function's invocation.

When working with AWS SDKs, you don't need to provide credentials for the SDK directly in code. By default, the credential provider chain sequentially checks each place where you can set credentials and selects the first one available—usually the environment variables (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_SESSION_TOKEN`).

Lambda injects the source function ARN into the credentials context if the request is an AWS API request that comes from within your execution environment. Lambda also injects the source function ARN for the following AWS API requests that Lambda makes on your behalf outside of your execution environment:

Service	Action	Reason
CloudWatch Logs	<code>CreateLogGroup</code> , <code>CreateLogStream</code> , <code>PutLogEvents</code>	To store logs into a CloudWatch Logs log group
X-Ray	<code>PutTraceSegments</code>	To send trace data to X-Ray

Service	Action	Reason
Amazon EFS	ClientMount	To connect your function to an Amazon Elastic File System (Amazon EFS) file system

Other AWS API calls that Lambda makes outside of your execution environment on your behalf using the same execution role don't contain the source function ARN. Examples of such API calls outside the execution environment include:

- Calls to AWS Key Management Service (AWS KMS) to automatically encrypt and decrypt your environment variables.
- Calls to Amazon Elastic Compute Cloud (Amazon EC2) to create elastic network interfaces (ENIs) for a VPC-enabled function.
- Calls to AWS services, such as Amazon Simple Queue Service (Amazon SQS), to read from an event source that's set up as an [event source mapping \(p. 131\)](#).

With the source function ARN in the credentials context, you can verify whether a call to your resource came from a specific Lambda function's code. To verify this, use the `lambda:SourceFunctionArn` condition key in an IAM identity-based policy or service control policy (SCP).

Note

You cannot use the `lambda:SourceFunctionArn` condition key in resource-based policies.

With this condition key in your identity-based policies or SCPs, you can implement security controls for the API actions that your function code makes to other AWS services. This has a few key security applications, such as helping you identify the source of a credential leak.

Note

The `lambda:SourceFunctionArn` condition key is different from the `lambda:FunctionArn` and `aws:SourceArn` condition keys. The `lambda:FunctionArn` condition key applies only to [event source mappings \(p. 131\)](#) and helps define which functions your event source can invoke.

The `aws:SourceArn` condition key applies only to policies where your Lambda function is the target resource, and helps define which other AWS services and resources can invoke that function. The `lambda:SourceFunctionArn` condition key can apply to any identity-based policy or SCP to define the specific Lambda functions that have permissions to make specific AWS API calls to other resources.

To use `lambda:SourceFunctionArn` in your policy, include it as a condition with any of the [ARN condition operators](#). The value of the key must be a valid ARN.

For example, suppose your Lambda function code makes an `s3:PutObject` call that targets a specific Amazon S3 bucket. You might want to allow only one specific Lambda function to have `s3:PutObject` access that bucket. In this case, your function's execution role should have a policy attached that looks like this:

Example policy granting a specific Lambda function access to an Amazon S3 resource

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ExampleSourceFunctionArn",
      "Effect": "Allow",
      "Action": "s3:PutObject",
      "Resource": "arn:aws:s3:::lambda_bucket/*",
      "Condition": {
        "StringEquals": {
          "aws:SourceArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction"
        }
      }
    }
  ]
}
```

```

        "ArnEquals": {
            "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
        }
    }
}

```

This policy allows only s3:PutObject access if the source is the Lambda function with ARN arn:aws:lambda:us-east-1:123456789012:function:source_lambda. This policy doesn't allow s3:PutObject access to any other calling identity. This is true even if a different function or entity makes an s3:PutObject call with the same execution role.

You can also use lambda:SourceFunctionArn in [service control policies](#). For example, suppose you want to restrict access to your bucket to either a single Lambda function's code or to calls from a specific Amazon Virtual Private Cloud (VPC). The following SCP illustrates this.

Example policy denying access to Amazon S3 under specific conditions

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "StringNotEqualsIfExists": {
                    "aws:SourceVpc": [
                        "vpc-12345678"
                    ]
                },
                "ArnNotEqualsIfExists": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
                }
            }
        ]
    }
}

```

This policy denies all S3 actions unless they come from a specific Lambda function with ARN arn:aws:lambda:123456789012:function:source_lambda, or unless they come from the specified VPC. The StringNotEqualsIfExists operator tells IAM to process this condition only if the aws:SourceVpc key is present in the request. Similarly, IAM considers the ArnNotEqualsIfExists operator only if the lambda:SourceFunctionArn exists.

Identity-based IAM policies for Lambda

You can use identity-based policies in AWS Identity and Access Management (IAM) to grant users in your account access to Lambda. Identity-based policies can apply to users directly, or to groups and roles that are associated with a user. You can also grant users in another account permission to assume a role in your account and access your Lambda resources. This page shows an example of how identity-based policies can be used for function development.

Lambda provides AWS managed policies that grant access to Lambda API actions and, in some cases, access to other AWS services used to develop and manage Lambda resources. Lambda updates these managed policies as needed to ensure that your users have access to new features when they're released.

- **AWSLambda_FullAccess** – Grants full access to Lambda actions and other AWS services used to develop and maintain Lambda resources. This policy was created by scoping down the previous policy **AWSLambdaFullAccess**.
 - **AWSLambda_ReadOnlyAccess** – Grants read-only access to Lambda resources. This policy was created by scoping down the previous policy **AWSLambdaReadOnlyAccess**.
 - **AWSLambdaRole** – Grants permissions to invoke Lambda functions.

AWS managed policies grant permission to API actions without restricting the Lambda functions or layers that a user can modify. For finer-grained control, you can create your own policies that limit the scope of a user's permissions.

Sections

- [Function development \(p. 823\)](#)
 - [Layer development and use \(p. 826\)](#)
 - [Cross-account roles \(p. 827\)](#)
 - [Condition keys for VPC settings \(p. 827\)](#)

Function development

Use identity-based policies to allow users to perform operations on Lambda functions.

Note

For a function defined as a container image, the user permission to access the image MUST be configured in the Amazon Elastic Container Registry. For an example, see [Amazon ECR permissions](#). (p. 112)

The following shows an example of a permissions policy with limited scope. It allows a user to create and manage Lambda functions named with a designated prefix (`intern-`), and configured with a designated execution role.

Example Function development policy

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "ReadOnlyPermissions",  
            "Effect": "Allow",  
            "Action": [  
                "lambda:GetAccountSettings",  
                "lambda:GetEventSourceMapping",  
                "lambda:GetFunction",  
                "lambda:GetFunctionConfiguration",  
                "lambda:ListFunctions",  
                "lambda:ListFunctionCodeSha256",  
                "lambda:ListTags",  
                "lambda:UpdateFunctionConfiguration",  
                "lambda:UpdateFunctionCode",  
                "lambda:UpdateFunctionCodeSha256",  
                "lambda:UpdateFunctionTags"  
            ]  
        }  
    ]  
}
```

```

        "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": {
        "StringLike": {
            "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-*"
}
]
}

```

The permissions in the policy are organized into statements based on the [resources and conditions \(p. 838\)](#) that they support.

- **ReadOnlyPermissions** – The Lambda console uses these permissions when you browse and view functions. They don't support resource patterns or conditions.

```

    "Action": [
        "lambda:GetAccountSettings",
        "lambda:GetEventSourceMapping",
        "lambda:GetFunction",
        "lambda:GetFunctionConfiguration",
        "lambda:GetFunctionCodeSigningConfig",
        "lambda:GetFunctionConcurrency",
        "lambda>ListEventSourceMappings",
        "lambda>ListFunctions",
        "lambda>ListTags",
        "iam>ListRoles"
    ],
    "Resource": "*"

```

- **DevelopFunctions** – Use any Lambda action that operates on functions prefixed with `intern-`, except `AddPermission` and `PutFunctionConcurrency`. `AddPermission` modifies the [resource-based policy \(p. 832\)](#) on the function and can have security implications. `PutFunctionConcurrency` reserves scaling capacity for a function and can take capacity away from other functions.

```

    "NotAction": [
        "lambda:AddPermission",
        "lambda:PutFunctionConcurrency"
    ],
    "Resource": "arn:aws:lambda:*.*:function:intern-*"

```

- **DevelopEventSourceMappings** – Manage event source mappings on functions that are prefixed with `intern-`. These actions operate on event source mappings, but you can restrict them by function with a *condition*.

```

    "Action": [
        "lambda>DeleteEventSourceMapping",
        "lambda:UpdateEventSourceMapping",
        "lambda>CreateEventSourceMapping"
    ],
    "Resource": "*",
    "Condition": {
        "StringLike": {
            "lambda:FunctionArn": "arn:aws:lambda:*.*:function:intern-*"
        }
    }

```

- **PassExecutionRole** – View and pass only a role named `intern-lambda-execution-role`, which must be created and managed by a user with IAM permissions. `PassRole` is used when you assign an execution role to a function.

```

    "Action": [
        "iam>ListRolePolicies",
        "iam>ListAttachedRolePolicies",
        "iam>GetRole",
        "iam>GetRolePolicy",
        "iam>PassRole",
        "iam>SimulatePrincipalPolicy"
    ],
    "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"

```

- **ViewLogs** – Use CloudWatch Logs to view logs for functions that are prefixed with `intern-`.

```

    "Action": [
        "logs:*log"
    ],
    "Resource": "arn:aws:logs:*::log-group:/aws/lambda/intern-*"

```

This policy allows a user to get started with Lambda, without putting other users' resources at risk. It doesn't allow a user to configure a function to be triggered by or call other AWS services, which requires broader IAM permissions. It also doesn't include permission to services that don't support limited-scope policies, like CloudWatch and X-Ray. Use the read-only policies for these services to give the user access to metrics and trace data.

When you configure triggers for your function, you need access to use the AWS service that invokes your function. For example, to configure an Amazon S3 trigger, you need permission to use the Amazon S3 actions that manage bucket notifications. Many of these permissions are included in the **AWSLambdaFullAccess** managed policy. Example policies are available in this guide's [GitHub repository](#).

Layer development and use

The following policy grants a user permission to create layers and use them with functions. The resource patterns allow the user to work in any AWS Region and with any layer version, as long as the name of the layer starts with test-.

Example layer development policy

```
{
    "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-*::*"
        }
    ]
}
```

You can also enforce layer use during function creation and configuration with the `lambda:Layer` condition. For example, you can prevent users from using layers published by other accounts. The following policy adds a condition to the `CreateFunction` and `UpdateFunctionConfiguration` actions to require that any layers specified come from account 123456789012.

```
{
    "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:*:*"
        ]
    }
}
}
```

To ensure that the condition applies, verify that no other statements grant the user permission to these actions.

Cross-account roles

You can apply any of the preceding policies and statements to a role, which you can then share with another account to give it access to your Lambda resources. Unlike a user, a role doesn't have credentials for authentication. Instead, it has a *trust policy* that specifies who can assume the role and use its permissions.

You can use cross-account roles to give accounts that you trust access to Lambda actions and resources. If you just want to grant permission to invoke a function or use a layer, use [resource-based policies \(p. 832\)](#) instead.

For more information, see [IAM roles](#) in the *IAM User Guide*.

Condition keys for VPC settings

You can use condition keys for VPC settings to provide additional permission controls for your Lambda functions. For example, you can enforce that all Lambda functions in your organization are connected to a VPC. You can also specify the subnets and security groups that the functions are allowed to use, or are denied from using.

For more information, see [the section called “Using IAM condition keys for VPC settings” \(p. 225\)](#).

Attribute-based access control for Lambda

With [attribute-based access control \(ABAC\)](#), you can use tags to control access to your Lambda functions. You can attach tags to a Lambda function, pass them in certain API requests, or attach them to the AWS Identity and Access Management (IAM) principal making the request. For more information about how AWS grants attribute-based access, see [Controlling access to AWS resources using tags](#) in the *IAM User Guide*.

You can use ABAC to [grant least privilege](#) without specifying an Amazon Resource Name (ARN) or ARN pattern in the IAM policy. Instead, you can specify a tag in the [condition element](#) of an IAM policy to control access. Scaling is easier with ABAC because you don't have to update your IAM policies when you create new functions. Instead, add tags to the new functions to control access.

In Lambda, tags work at the function level. Tags aren't supported for layers, code signing configurations, or event source mappings. When you tag a function, those tags apply to all versions and aliases associated with the function. For information about how to tag functions, see [Using tags on Lambda functions \(p. 250\)](#).

You can use the following condition keys to control function actions:

- [aws:ResourceTag/tag-key](#): Control access based on the tags that are attached to Lambda functions.
- [aws:RequestTag/tag-key](#): Require tags to be present in a request, such as when creating a new function.
- [aws:PrincipalTag/tag-key](#): Control what the IAM principal (the person making the request) is allowed to do based on the tags that are attached to their IAM [user](#) or [role](#).
- [aws:TagKeys](#): Control whether specific tag keys can be used in a request.

For a complete list of Lambda actions that support ABAC, see [Function actions \(p. 841\)](#) and check the **Condition** column in the table.

The following steps demonstrate one way to set up permissions using ABAC. In this example scenario, you'll create four IAM permissions policies. Then, you'll attach these policies to a new IAM role. Finally, you'll create an IAM user and give that user permission to assume the new role.

Prerequisites

Make sure that you have a [Lambda execution role \(p. 816\)](#). You'll use this role when you grant IAM permissions and when you create a Lambda function.

Step 1: Require tags on new functions

When using ABAC with Lambda, it's a best practice to require that all functions have tags. This helps ensure that your ABAC permissions policies work as expected.

Create an IAM policy similar to the following example. This policy uses the [aws:RequestTag/tag-key](#) and [aws:TagKeys](#) condition keys to require that new functions and the IAM principal creating the functions both have the project tag. The `ForAllValues` modifier ensures that `project` is the only allowed tag. If you don't include the `ForAllValues` modifier, users can add other tags to the function as long as they also pass `project`.

Example – Require tags on new functions

```
{  
  "Version": "2012-10-17",  
  "Statement": {
```

```

    "Effect": "Allow",
    "Action": [
        "lambda>CreateFunction",
        "lambda>TagResource"
    ],
    "Resource": "arn:aws:lambda:*:*:function:*",
    "Condition": {
        "StringEquals": {
            "aws:RequestTag/projectproject}"
        },
        "ForAllValues:StringEquals": {
            "aws:TagKeys": "project"
        }
    }
}
}

```

Step 2: Allow actions based on tags attached to a Lambda function and IAM principal

Create a second IAM policy using the [aws:ResourceTag/tag-key](#) condition key to require the principal's tag to match the tag that's attached to the function. The following example policy allows principals with the project tag to invoke functions with the project tag. If a function has any other tags, the action is denied.

Example – Require matching tags on function and IAM principal

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction",
                "lambda:GetFunction"
            ],
            "Resource": "arn:aws:lambda:*:*:function:*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/projectproject}"
                }
            }
        ]
    }
}
```

Step 3: Grant list permissions

Create a policy that allows the principal to list Lambda functions and IAM roles. This allows the principal to see all Lambda functions and IAM roles on the console and when calling the API actions.

Example – Grant Lambda and IAM list permissions

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllResourcesLambdaNoTags",
            "Effect": "Allow",

```

```
        "Action": [
            "lambda:GetAccountSettings",
            "lambda>ListFunctions",
            "iam>ListRoles"
        ],
        "Resource": "*"
    ]
}
```

Step 4: Grant IAM permissions

Create a policy that allows **iam:PassRole**. This permission is required when you assign an execution role to a function. In the following example policy, replace the example ARN with the ARN of your Lambda execution role.

Note

Do not use the `ResourceTag` condition key in a policy with the `iam:PassRole` action. You cannot use the tag on an IAM role to control access to who can pass that role. For more information about permissions required to pass a role to a service, see [Granting a user permissions to pass a role to an AWS service](#).

Example – Grant permission to pass the execution role

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": "arn:aws:iam::111122223333:role/lambda-ex"
        }
    ]
}
```

Step 5: Create the IAM role

It's a best practice to [use roles to delegate permissions](#). [Create an IAM role](#) called abac-project-role:

- On **Step 1: Select trusted entity**: Choose **AWS account** and then choose **This account**.
- On **Step 2: Add permissions**: Attach the four IAM policies that you created in the previous steps.
- On **Step 3: Name, review, and create**: Choose **Add tag**. For **Key**, enter project. Don't enter a **Value**.

Step 6: Create the IAM user

[Create an IAM user](#) called abac-test-user. In the **Set permissions** section, choose **Attach existing policies directly** and then choose **Create policy**. Enter the following policy definition. Replace **111122223333** with your [AWS account ID](#). This policy allows abac-test-user to assume abac-project-role.

Example – Allow IAM user to assume ABAC role

```
{
```

```
"Version": "2012-10-17",
"Statement": [
    "Effect": "Allow",
    "Action": "sts:AssumeRole",
    "Resource": "arn:aws:iam::111122223333:role/abac-project-role"
}
```

Step 7: Test the permissions

1. Sign in to the AWS console as abac-test-user. For more information, see [Sign in as an IAM user](#).
2. Switch to the abac-project-role role. For more information, see [Switching to a role \(console\)](#).
3. [Create a Lambda function \(p. 250\)](#):
 - Under **Permissions**, choose **Change default execution role**, and then for **Execution role**, choose **Use an existing role**. Choose the same execution role that you used in [Step 4: Grant IAM permissions \(p. 830\)](#).
 - Under **Advanced settings**, choose **Enable tags** and then choose **Add new tag**. For **Key**, enter **project**. Don't enter a **Value**.
4. [Test the function \(p. 163\)](#).
5. Create a second Lambda function and add a different tag, such as environment. This operation should fail because the ABAC policy that you created in [Step 1: Require tags on new functions \(p. 828\)](#) only allows the principal to create functions with the project tag.
6. Create a third function without tags. This operation should fail because the ABAC policy that you created in [Step 1: Require tags on new functions \(p. 828\)](#) doesn't allow the principal to create functions without tags.

This authorization strategy allows you to control access without creating new policies for each new user. To grant access to new users, simply give them permission to assume the role that corresponds to their assigned project.

Using resource-based policies for Lambda

Lambda supports resource-based permissions policies for Lambda functions and layers. Resource-based policies let you grant usage permission to other AWS accounts or organizations on a per-resource basis. You also use a resource-based policy to allow an AWS service to invoke your function on your behalf.

For Lambda functions, you can [grant an account permission \(p. 834\)](#) to invoke or manage a function. You can also use a single resource-based policy to grant permissions to an entire organization in AWS Organizations. You can also use resource-based policies to [grant invoke permission to an AWS service \(p. 833\)](#) that invokes a function in response to activity in your account.

To view a function's resource-based policy

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Permissions**.
4. Scroll down to **Resource-based policy** and then choose **View policy document**. The resource-based policy shows the permissions that are applied when another account or AWS service attempts to access the function. The following example shows a statement that allows Amazon S3 to invoke a function named my-function for a bucket named my-bucket in account 123456789012.

Example Resource-based policy

```
{  
    "Version": "2012-10-17",  
    "Id": "default",  
    "Statement": [  
        {  
            "Sid": "lambda-allow-s3-my-function",  
            "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"  
                }  
            }  
        }  
    ]  
}
```

For Lambda layers, you can only use a resource-based policy on a specific layer version, instead of the entire layer. In addition to policies that grant permission to a single account or multiple accounts, for layers, you can also grant permission to all accounts in an organization.

Note

You can only update resource-based policies for Lambda resources within the scope of the [AddPermission \(p. 1141\)](#) and [AddLayerVersionPermission \(p. 1137\)](#) API actions. Currently, you can't author policies for your Lambda resources in JSON, or use conditions that don't map to parameters for those actions.

Resource-based policies apply to a single function, version, alias, or layer version. They grant permission to one or more services and accounts. For trusted accounts that you want to have access to multiple resources, or to use API actions that resource-based policies don't support, you can use [cross-account roles \(p. 823\)](#).

Topics

- [Granting function access to AWS services \(p. 833\)](#)
- [Granting function access to an organization \(p. 834\)](#)
- [Granting function access to other accounts \(p. 834\)](#)
- [Granting layer access to other accounts \(p. 836\)](#)
- [Cleaning up resource-based policies \(p. 836\)](#)

Granting function access to AWS services

When you [use an AWS service to invoke your function \(p. 556\)](#), you grant permission in a statement on a resource-based policy. You can apply the statement to the entire function to be invoked or managed, or limit the statement to a single version or alias.

Note

When you add a trigger to your function with the Lambda console, the console updates the function's resource-based policy to allow the service to invoke it. To grant permissions to other accounts or services that aren't available in the Lambda console, you can use the AWS CLI.

Add a statement with the `add-permission` command. The simplest resource-based policy statement allows a service to invoke a function. The following command grants Amazon SNS permission to invoke a function named `my-function`.

```
aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --statement-id sns \
--principal sns.amazonaws.com --output text
```

You should see the following output:

```
{"Sid":"sns","Effect":"Allow","Principal": \
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:us- \
east-2:123456789012:function:my-function"}
```

This lets Amazon SNS call the `lambda:Invoke` API for the function, but it doesn't restrict the Amazon SNS topic that triggers the invocation. To ensure that your function is only invoked by a specific resource, specify the Amazon Resource Name (ARN) of the resource with the `source-arn` option. The following command only allows Amazon SNS to invoke the function for subscriptions to a topic named `my-topic`.

```
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
```

Some services can invoke functions in other accounts. If you specify a source ARN that has your account ID in it, that isn't an issue. For Amazon S3, however, the source is a bucket whose ARN doesn't have an account ID in it. It's possible that you could delete the bucket and another account could create a bucket with the same name. Use the `source-account` option with your account ID to ensure that only resources in your account can invoke the function.

```
aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --statement-id s3-account \
```

```
--principal s3.amazonaws.com --source-arn arn:aws:s3:::my-bucket-123456 --source-account 123456789012
```

Granting function access to an organization

To grant permissions to an organization in AWS Organizations, specify the organization ID as the `principal-org-id`. The following [AddPermission \(p. 1141\)](#) AWS CLI command grants invocation access to all users in organization o-a1b2c3d4e5f.

```
aws lambda add-permission --function-name example \
--statement-id PrincipalOrgIDExample --action lambda:InvokeFunction \
--principal * --principal-org-id o-a1b2c3d4e5f
```

Note

In this command, `Principal` is `*`. This means that all users in the organization o-a1b2c3d4e5f get function invocation permissions. If you specify an AWS account or role as the `Principal`, then only that principal gets function invocation permissions, but only if they are also part of the o-a1b2c3d4e5f organization.

This command creates a resource-based policy that looks like the following:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PrincipalOrgIDExample",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:example",
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalOrgID": "o-a1b2c3d4e5f"
                }
            }
        }
    ]
}
```

For more information, see [aws:PrincipalOrgID](#) in the AWS Identity and Access Management user guide.

Granting function access to other accounts

To grant permissions to another AWS account, specify the account ID as the `principal`. The following example grants account 111122223333 permission to invoke `my-function` with the `prod` alias.

```
aws lambda add-permission --function-name my-function:prod --statement-id xaccount --action lambda:InvokeFunction \
--principal 111122223333 --output text
```

You should see the following output:

```
{"Sid":"xaccount","Effect":"Allow","Principal": {"AWS":"arn:aws:iam::111122223333:root"}, "Action":"lambda:InvokeFunction", "Resource":"arn:aws:lambda:us-east-2:123456789012:function:my-function"}
```

The resource-based policy grants permission for the other account to access the function, but doesn't allow users in that account to exceed their permissions. Users in the other account must have the corresponding [user permissions \(p. 823\)](#) to use the Lambda API.

To limit access to a user or role in another account, specify the full ARN of the identity as the principal. For example, `arn:aws:iam::123456789012:user/developer`.

The [alias \(p. 85\)](#) limits which version the other account can invoke. It requires the other account to include the alias in the function ARN.

```
aws lambda invoke --function-name arn:aws:lambda:us-west-2:123456789012:function:my-function:prod out
```

You should see the following output:

```
{  
    "StatusCode": 200,  
    "ExecutedVersion": "1"  
}
```

The function owner can then update the alias to point to a new version without the caller needing to change the way they invoke your function. This ensures that the other account doesn't need to change its code to use the new version, and it only has permission to invoke the version of the function associated with the alias.

You can grant cross-account access for most API actions that [operate on an existing function \(p. 841\)](#). For example, you could grant access to `lambda>ListAliases` to let an account get a list of aliases, or `lambda:GetFunction` to let them download your function code. Add each permission separately, or use `lambda:*` to grant access to all actions for the specified function.

Cross-account APIs

Currently, Lambda doesn't support cross-account actions for all of its APIs via resource-based policies. The following APIs are supported:

- [Invoke \(p. 1260\)](#)
- [GetFunction \(p. 1220\)](#)
- [GetFunctionConfiguration \(p. 1229\)](#)
- [UpdateFunctionCode \(p. 1367\)](#)
- [DeleteFunction \(p. 1193\)](#)
- [PublishVersion \(p. 1315\)](#)
- [ListVersionsByFunction \(p. 1306\)](#)
- [CreateAlias \(p. 1146\)](#)
- [GetAlias \(p. 1209\)](#)
- [ListAliases \(p. 1274\)](#)
- [UpdateAlias \(p. 1349\)](#)
- [DeleteAlias \(p. 1182\)](#)
- [GetPolicy \(p. 1252\)](#)
- [PutFunctionConcurrency \(p. 1327\)](#)
- [DeleteFunctionConcurrency \(p. 1197\)](#)
- [ListTags \(p. 1304\)](#)
- [TagResource \(p. 1345\)](#)
- [UntagResource \(p. 1347\)](#)

To grant other accounts permission for multiple functions, or for actions that don't operate on a function, we recommend that you use [IAM roles \(p. 823\)](#).

Granting layer access to other accounts

To grant layer-usage permission to another account, add a statement to the layer version's permissions policy using the **add-layer-version-permission** command. In each statement, you can grant permission to a single account, all accounts, or an organization.

```
aws lambda add-layer-version-permission --layer-name xray-sdk-nodejs --statement-id xaccount \
--action lambda:GetLayerVersion --principal 111122223333 --version-number 1 --output text
```

You should see output similar to the following:

```
e210ffdc-e901-43b0-824b-5fc0dd26d16 {"Sid":"xaccount","Effect":"Allow","Principal": {"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:lambda:us-east-2:123456789012:layer:xray-sdk-nodejs:1"}
```

Permissions apply only to a single layer version. Repeat the process each time that you create a new layer version.

To grant permission to all accounts in an organization, use the `organization-id` option. The following example grants all accounts in an organization permission to use version 3 of a layer.

```
aws lambda add-layer-version-permission --layer-name my-layer \
--statement-id engineering-org --version-number 3 --principal '*' \
--action lambda:GetLayerVersion --organization-id o-t194hfs8cz --output text
```

You should see the following output:

```
b0cd9796-d4eb-4564-939f-de7fe0b42236 {"Sid":"engineering-org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:lambda:us-east-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}}
```

To grant permission to all AWS accounts, use `*` for the principal, and omit the organization ID. For multiple accounts or organizations, you need to add multiple statements.

Cleaning up resource-based policies

To view a function's resource-based policy, use the `get-policy` command.

```
aws lambda get-policy --function-name my-function --output text
```

You should see the following output:

```
{"Version":"2012-10-17","Id":"default","Statement": [{"Sid":"sns","Effect":"Allow","Principal": {"Service":"s3.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:lambda*"}}}]} 7c681fc9-b791-4e91-acdf-eb847fd8aa0f0
```

For versions and aliases, append the version number or alias to the function name.

```
aws lambda get-policy --function-name my-function:PROD
```

To remove permissions from your function, use `remove-permission`.

```
aws lambda remove-permission --function-name example --statement-id sns
```

Use the `get-layer-version-policy` command to view the permissions on a layer.

```
aws lambda get-layer-version-policy --layer-name my-layer --version-number 3 --output text
```

You should see the following output:

```
b0cd9796-d4eb-4564-939f-de7fe0b42236 {"Sid":"engineering-org", "Effect":"Allow", "Principal":"*", "Action":"lambda:GetLayerVersion", "Resource": "arn:aws:lambda:us-west-2:123456789012:layer:my-layer:3", "Condition":{"StringEquals":{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}
```

Use `remove-layer-version-permission` to remove statements from the policy.

```
aws lambda remove-layer-version-permission --layer-name my-layer --version-number 3 --statement-id engineering-org
```

Resources and conditions for Lambda actions

You can restrict the scope of a user's permissions by specifying resources and conditions in an AWS Identity and Access Management (IAM) policy. Each action in a policy supports a combination of resource and condition types that varies depending on the behavior of the action.

Every IAM policy statement grants permission to an action that's performed on a resource. When the action doesn't act on a named resource, or when you grant permission to perform the action on all resources, the value of the resource in the policy is a wildcard (*). For many actions, you can restrict the resources that a user can modify by specifying the Amazon Resource Name (ARN) of a resource, or an ARN pattern that matches multiple resources.

To restrict permissions by resource, specify the resource by ARN.

Lambda resource ARN format

- Function – arn:aws:lambda:*us-west-2:123456789012:function:my-function*
- Function version – arn:aws:lambda:*us-west-2:123456789012:function:my-function:1*
- Function alias – arn:aws:lambda:*us-west-2:123456789012:function:my-function:TEST*
- Event source mapping – arn:aws:lambda:*us-west-2:123456789012:event-source-mapping:fa123456-14a1-4fd2-9fec-83de64ad683de6d47*
- Layer – arn:aws:lambda:*us-west-2:123456789012:layer:my-layer*
- Layer version – arn:aws:lambda:*us-west-2:123456789012:layer:my-layer:1*

For example, the following policy allows a user in AWS account 123456789012 to invoke a function named my-function in the US West (Oregon) AWS Region.

Example invoke function policy

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "Invoke",  
            "Effect": "Allow",  
            "Action": [  
                "lambda:InvokeFunction"  
            ],  
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:my-function"  
        }  
    ]  
}
```

This is a special case where the action identifier (lambda:InvokeFunction) differs from the API operation ([Invoke \(p. 1260\)](#)). For other actions, the action identifier is the operation name prefixed by lambda:.

Sections

- [Policy conditions \(p. 839\)](#)
- [Function resource names \(p. 839\)](#)
- [Function actions \(p. 841\)](#)
- [Event source mapping actions \(p. 843\)](#)
- [Layer actions \(p. 843\)](#)

Policy conditions

Conditions are an optional policy element that applies additional logic to determine if an action is allowed. In addition to common [conditions](#) that all actions support, Lambda defines condition types that you can use to restrict the values of additional parameters on some actions.

For example, the `lambda:Principal` condition lets you restrict the service or account that a user can grant invocation access to on a function's [resource-based policy \(p. 832\)](#). The following policy lets a user grant permission to Amazon Simple Notification Service (Amazon SNS) topics to invoke a function named `test`.

Example manage function policy permissions

```
{
    "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"
                }
            }
        }
    ]
}
```

The condition requires that the principal is Amazon SNS and not another service or account. The resource pattern requires that the function name is `test` and includes a version number or alias. For example, `test:v1`.

For more information on resources and conditions for Lambda and other AWS services, see [Actions, resources, and condition keys for AWS services](#) in the *Service Authorization Reference*.

Function resource names

You reference a Lambda function in a policy statement using an Amazon Resource Name (ARN). The format of a function ARN depends on whether you are referencing the whole function (unqualified) or a function [version \(p. 83\)](#) or [alias \(p. 85\)](#) (qualified).

When making Lambda API calls, users can specify a version or alias by passing a version ARN or alias ARN in the [GetFunction \(p. 1220\)](#) `FunctionName` parameter, or by setting a value in the [GetFunction \(p. 1220\)](#) `Qualifier` parameter. Lambda makes authorization decisions by comparing the resource element in the IAM policy with both the `FunctionName` and `Qualifier` passed in API calls. If there is a mismatch, Lambda denies the request.

Whether you are allowing or denying an action on your function, you must use the correct function ARN types in your policy statement to achieve the results that you expect. For example, if your policy references the unqualified ARN, Lambda accepts requests that reference the unqualified ARN but denies requests that reference a qualified ARN.

Note

You can't use a wildcard character (*) to match the account ID. For more information on accepted syntax, see [IAM JSON policy reference](#) in the *IAM User Guide*.

Example allowing invocation of an unqualified ARN

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "lambda:InvokeFunction",  
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction"  
        }  
    ]  
}
```

If your policy references a specific qualified ARN, Lambda accepts requests that reference that ARN but denies requests that reference the unqualified ARN or a different qualified ARN, for example, myFunction:2.

Example allowing invocation of a specific qualified ARN

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "lambda:InvokeFunction",  
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction:1"  
        }  
    ]  
}
```

If your policy references any qualified ARN using :*, Lambda accepts any qualified ARN but denies requests that reference the unqualified ARN.

Example allowing invocation of any qualified ARN

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "lambda:InvokeFunction",  
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction:/*"  
        }  
    ]  
}
```

If your policy references any ARN using *, Lambda accepts any qualified or unqualified ARN.

Example allowing invocation of any qualified or unqualified ARN

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "lambda:InvokeFunction",  
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction*"  
        }  
    ]  
}
```

Function actions

Actions that operate on a function can be restricted to a specific function, version, or alias ARN, as described in the following table. Actions that don't support resource restrictions are granted for all resources (*).

Function actions

Action	Resource	Condition
AddPermission (p. 1141)	Function	lambda:Principal
RemovePermission (p. 1343)	Function version Function alias	aws:ResourceTag/\${TagKey} lambda:FunctionUrlAuthType
Invoke (p. 1260) Permission: lambda:InvokeFunction	Function Function version Function alias	aws:ResourceTag/\${TagKey}
CreateFunction (p. 1165)	Function	lambda:CodeSigningConfigArn lambda:Layer lambda:VpcIds lambda:SubnetIds lambda:SecurityGroupIds aws:ResourceTag/\${TagKey} aws:RequestTag/\${TagKey} aws:TagKeys
UpdateFunctionConfiguration (p. 1377)	Function	lambda:CodeSigningConfigArn lambda:Layer lambda:VpcIds lambda:SubnetIds lambda:SecurityGroupIds aws:ResourceTag/\${TagKey}
CreateAlias (p. 1146) DeleteAlias (p. 1182) DeleteFunction (p. 1193) DeleteFunctionCodeSigningConfig DeleteFunctionConcurrency (p. 1197) GetAlias (p. 1209)	Function	aws:ResourceTag/\${TagKey}

Action	Resource	Condition
GetFunction (p. 1220) GetFunctionCodeSigningConfig GetFunctionConcurrency GetFunctionConfiguration (p. 1229) GetPolicy (p. 1252) ListProvisionedConcurrencyConfigs ListAliases (p. 1274) ListTags (p. 1304) ListVersionsByFunction (p. 1306) PublishVersion (p. 1315) PutFunctionCodeSigningConfig PutFunctionConcurrency (p. 1327) UpdateAlias (p. 1349) UpdateFunctionCode (p. 1367)		
CreateFunctionUrlConfig DeleteFunctionUrlConfig GetFunctionUrlConfig UpdateFunctionUrlConfig	Function Function alias	lambda:FunctionUrlAuthType lambda:FunctionArn aws:ResourceTag/\${TagKey}
ListFunctionUrlConfigs	Function	lambda:FunctionUrlAuthType
DeleteFunctionEventInvokeConfig GetFunctionEventInvokeConfig ListFunctionEventInvokeConfigs PutFunctionEventInvokeConfig UpdateFunctionEventInvokeConfig	Function	aws:ResourceTag/\${TagKey}
DeleteProvisionedConcurrencyConfig GetProvisionedConcurrencyConfig PutProvisionedConcurrencyConfig	Function alias Function version	aws:ResourceTag/\${TagKey}
GetAccountSettings (p. 1207) ListFunctions (p. 1286)	*	None

Action	Resource	Condition
TagResource (p. 1345)	Function	aws:ResourceTag/\${TagKey} aws:RequestTag/\${TagKey} aws:TagKeys
UntagResource (p. 1347)	Function	aws:ResourceTag/\${TagKey} aws:TagKeys

Event source mapping actions

For [event source mappings \(p. 131\)](#), you can restrict delete and update permissions to a specific event source. The `lambda:FunctionArn` condition lets you restrict which functions a user can configure an event source to invoke.

For these actions, the resource is the event source mapping, so Lambda provides a condition that lets you restrict permission based on the function that the event source mapping invokes.

Event source mapping actions

Action	Resource	Condition
DeleteEventSourceMapping (p. 1186)	Event source mapping	<code>lambda:FunctionArn</code>
UpdateEventSourceMapping (p. 1356)		
CreateEventSourceMapping (p. 1153)	*	<code>lambda:FunctionArn</code>
GetEventSourceMapping (p. 1214)	*	None
ListEventSourceMappings (p. 1279)		

Layer actions

Layer actions let you restrict the layers that a user can manage or use with a function. Actions related to layer use and permissions act on a version of a layer, while `PublishLayerVersion` acts on a layer name. You can use either with wildcards to restrict the layers that a user can work with by name.

Note

The [GetLayerVersion \(p. 1243\)](#) action also covers [GetLayerVersionByArn \(p. 1247\)](#). Lambda does not support `GetLayerVersionByArn` as an IAM action.

Layer actions

Action	Resource	Condition
AddLayerVersionPermission (p. 1137)	Layer version	None
RemoveLayerVersionPermission (p. 1341)		
GetLayerVersion (p. 1243)		
GetLayerVersionPolicy (p. 1250)		

Action	Resource	Condition
DeleteLayerVersion (p. 1203)		
ListLayerVersions (p. 1298)	Layer	None
PublishLayerVersion (p. 1310)		
ListLayers (p. 1295)	*	None

Using permissions boundaries for AWS Lambda applications

When you [create an application \(p. 964\)](#) in the AWS Lambda console, Lambda applies a *permissions boundary* to the application's IAM roles. The permissions boundary limits the scope of the [execution role \(p. 816\)](#) that the application's template creates for each of its functions, and any roles that you add to the template. The permissions boundary prevents users with write access to the application's Git repository from escalating the application's permissions beyond the scope of its own resources.

The application templates in the Lambda console include a global property that applies a permissions boundary to all functions that they create.

```
Globals:  
  Function:  
    PermissionsBoundary: !Sub 'arn:${AWS::Partition}:iam::${AWS::AccountId}:policy/${AppId}-${AWS::Region}-PermissionsBoundary'
```

The boundary limits the permissions of the functions' roles. You can add permissions to a function's execution role in the template, but that permission is only effective if it's also allowed by the permissions boundary. The role that AWS CloudFormation assumes to deploy the application enforces the use of the permissions boundary. That role only has permission to create and pass roles that have the application's permissions boundary attached.

By default, an application's permissions boundary enables functions to perform actions on the resources in the application. For example, if the application includes an Amazon DynamoDB table, the boundary allows access to any API action that can be restricted to operate on specific tables with resource-level permissions. You can only use actions that don't support resource-level permissions if they're specifically permitted in the boundary. These include Amazon CloudWatch Logs and AWS X-Ray API actions for logging and tracing.

Example permissions boundary

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Action": [  
        "*"  
      ],  
      "Resource": [  
        "arn:aws:lambda:us-east-2:123456789012:function:my-app-getAllItemsFunction-*",  
        "arn:aws:lambda:us-east-2:123456789012:function:my-app getByIdFunction-*",  
        "arn:aws:lambda:us-east-2:123456789012:function:my-app-putItemFunction-*",  
        "arn:aws:dynamodb:us-east-1:123456789012:table/my-app-SampleTable-*"  
      ],  
      "Effect": "Allow",  
      "Sid": "StackResources"  
    },  
    {  
      "Action": [  
        "logs>CreateLogGroup",  
        "logs>CreateLogStream",  
        "logs>DescribeLogGroups",  
        "logs>PutLogEvents",  
        "xray:Put*"  
      ],  
      "Resource": "*"  
    }  
  ]  
}
```

```
        "Effect": "Allow",
        "Sid": "StaticPermissions"
    },
    ...
]
```

To access other resources or API actions, you or an administrator must expand the permissions boundary to include those resources. You might also need to update the execution role or deployment role of an application to allow the use of additional actions.

- **Permissions boundary** – Extend the application's permissions boundary when you add resources to your application, or the execution role needs access to more actions. In IAM, add resources to the boundary to allow the use of API actions that support resource-level permissions on that resource's type. For actions that don't support resource-level permissions, add them in a statement that isn't scoped to any resource.
- **Execution role** – Extend a function's execution role when it needs to use additional actions. In the application template, add policies to the execution role. The intersection of permissions in the boundary and execution role is granted to the function.
- **Deployment role** – Extend the application's deployment role when it needs additional permissions to create or configure resources. In IAM, add policies to the application's deployment role. The deployment role needs the same user permissions that you need to deploy or update an application in AWS CloudFormation.

For a tutorial that walks through adding resources to an application and extending its permissions, see [???](#) (p. 964).

For more information, see [Permissions boundaries for IAM entities](#) in the IAM User Guide.

Security in AWS Lambda

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The [shared responsibility model](#) describes this as security of the cloud and security *in the cloud*:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the [AWS compliance programs](#). To learn about the compliance programs that apply to AWS Lambda, see [AWS Services in Scope by Compliance Program](#).
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Lambda. The following topics show you how to configure Lambda to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Lambda resources.

For more information about applying security principles to Lambda applications, see [Security](#) in the *Lambda operator guide*.

Topics

- [Data protection in AWS Lambda \(p. 847\)](#)
- [Identity and access management for Lambda \(p. 849\)](#)
- [Compliance validation for AWS Lambda \(p. 857\)](#)
- [Resilience in AWS Lambda \(p. 857\)](#)
- [Infrastructure security in AWS Lambda \(p. 858\)](#)
- [Configuration and vulnerability analysis in AWS Lambda \(p. 858\)](#)

Data protection in AWS Lambda

The AWS [shared responsibility model](#) applies to data protection in AWS Lambda. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. This content includes the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the [Data Privacy FAQ](#). For information about data protection in Europe, see the [AWS Shared Responsibility Model and GDPR](#) blog post on the [AWS Security Blog](#).

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center (successor to AWS Single Sign-On) or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see [Federal Information Processing Standard \(FIPS\) 140-2](#).

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Lambda or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Sections

- [Encryption in transit \(p. 848\)](#)
- [Encryption at rest \(p. 848\)](#)

Encryption in transit

Lambda API endpoints only support secure connections over HTTPS. When you manage Lambda resources with the AWS Management Console, AWS SDK, or the Lambda API, all communication is encrypted with Transport Layer Security (TLS). For a full list of API endpoints, see [AWS Regions and endpoints](#) in the AWS General Reference.

When you [connect your function to a file system \(p. 236\)](#), Lambda uses encryption in transit for all connections. For more information, see [Data encryption in Amazon EFS](#) in the *Amazon Elastic File System User Guide*.

When you use [environment variables \(p. 76\)](#), you can enable console encryption helpers to use client-side encryption to protect the environment variables in transit. For more information, see [Securing environment variables \(p. 80\)](#).

Encryption at rest

You can use [environment variables \(p. 76\)](#) to store secrets securely for use with Lambda functions. Lambda always encrypts environment variables at rest. By default, Lambda uses an AWS KMS key that Lambda creates in your account to encrypt your environment variables. This AWS managed key is named aws/lambda.

On a per-function basis, you can optionally configure Lambda to use a customer managed key instead of the default AWS managed key to encrypt your environment variables. For more information, see [Securing environment variables \(p. 80\)](#).

Lambda always encrypts files that you upload to Lambda, including [deployment packages \(p. 881\)](#) and [layer archives \(p. 93\)](#).

Amazon CloudWatch Logs and AWS X-Ray also encrypt data by default, and can be configured to use a customer managed key. For details, see [Encrypt log data in CloudWatch Logs](#) and [Data protection in AWS X-Ray](#).

Identity and access management for Lambda

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Lambda resources. IAM is an AWS service that you can use with no additional charge.

Topics

- [Audience \(p. 849\)](#)
- [Authenticating with identities \(p. 849\)](#)
- [Managing access using policies \(p. 851\)](#)
- [How AWS Lambda works with IAM \(p. 853\)](#)
- [AWS Lambda identity-based policy examples \(p. 853\)](#)
- [Troubleshooting AWS Lambda identity and access \(p. 855\)](#)

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in Lambda.

Service user – If you use the Lambda service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Lambda features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in Lambda, see [Troubleshooting AWS Lambda identity and access \(p. 855\)](#).

Service administrator – If you're in charge of Lambda resources at your company, you probably have full access to Lambda. It's your job to determine which Lambda features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with Lambda, see [How AWS Lambda works with IAM \(p. 853\)](#).

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to Lambda. To view example Lambda identity-based policies that you can use in IAM, see [AWS Lambda identity-based policy examples \(p. 853\)](#).

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (successor to AWS Single Sign-On) (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see [How to sign in to your AWS account](#) in the [AWS Sign-In User Guide](#).

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests using your credentials. If you don't use AWS

tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see [Signature Version 4 signing process](#) in the *AWS General Reference*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see [Multi-factor authentication](#) in the *AWS IAM Identity Center (successor to AWS Single Sign-On) User Guide* and [Using multi-factor authentication \(MFA\) in AWS](#) in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the *AWS account root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see [Tasks that require root user credentials](#) in the *AWS Account Management Reference Guide*.

Users and groups

An [IAM user](#) is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see [Rotate access keys regularly for use cases that require long-term credentials](#) in the *IAM User Guide*.

An [IAM group](#) is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see [When to create an IAM user \(instead of a role\)](#) in the *IAM User Guide*.

IAM roles

An [IAM role](#) is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by [switching roles](#). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see [Using IAM roles](#) in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see [Creating a role for a third-party Identity Provider](#) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see [Permission sets](#) in the *AWS IAM Identity Center (successor to AWS Single Sign-On) User Guide*.

- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see [How IAM roles differ from resource-based policies](#) in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
- **Principal permissions** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. Policies grant permissions to a principal. When you use some services, you might perform an action that then triggers another action in a different service. In this case, you must have permissions to perform both actions. To see whether an action requires additional dependent actions in a policy, see [Actions, Resources, and Condition Keys for AWS Lambda](#) in the *Service Authorization Reference*.
- **Service role** – A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Creating a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.
- **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see [Using an IAM role to grant permissions to applications running on Amazon EC2 instances](#) in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see [When to create an IAM role \(instead of a user\)](#) in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see [Overview of JSON policies](#) in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see [Creating IAM policies](#) in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see [Choosing between managed policies and inline policies](#) in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must [specify a principal](#) in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see [Access control list \(ACL\) overview](#) in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the Principal field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see [Permissions boundaries for IAM entities](#) in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see [How SCPs work](#) in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies.

Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see [Session policies](#) in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see [Policy evaluation logic](#) in the *IAM User Guide*.

How AWS Lambda works with IAM

Before you use IAM to manage access to Lambda, you should understand what IAM features are available to use with Lambda. To get a high-level view of how Lambda and other AWS services work with IAM, see [AWS services that work with IAM](#) in the *IAM User Guide*.

For an overview of permissions, policies, and roles as they are used by Lambda, see [Lambda resource access permissions \(p. 815\)](#).

AWS Lambda identity-based policy examples

By default, users and roles don't have permission to create or modify Lambda resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An administrator must create IAM policies that grant users and roles permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the users or groups that require those permissions.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see [Creating policies on the JSON tab](#) in the *IAM User Guide*.

Topics

- [Policy best practices \(p. 853\)](#)
- [Using the Lambda console \(p. 854\)](#)
- [Allow users to view their own permissions \(p. 854\)](#)

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Lambda resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see [AWS managed policies](#) or [AWS managed policies for job functions](#) in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see [Policies and permissions in IAM](#) in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all

requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see [IAM JSON policy elements: Condition](#) in the *IAM User Guide*.

- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions**
– IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see [IAM Access Analyzer policy validation](#) in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see [Configuring MFA-protected API access](#) in the *IAM User Guide*.

For more information about best practices in IAM, see [Security best practices in IAM](#) in the *IAM User Guide*.

Using the Lambda console

To access the AWS Lambda console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Lambda resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

For an example policy that grants minimal access for function development, see [Function development \(p. 823\)](#). In addition to Lambda APIs, the Lambda console uses other services to display trigger configuration and let you add new triggers. If your users use Lambda with other services, they need access to those services as well. For details on configuring other services with Lambda, see [Using AWS Lambda with other services \(p. 556\)](#).

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{  
    "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 ListPolicy"  
            ]  
        }  
    ]  
}
```

```
        "iam:ListGroupPolicies",
        "iam:ListPolicyVersions",
        "iam:ListPolicies",
        "iam:ListUsers"
    ],
    "Resource": "*"
}
]
```

Troubleshooting AWS Lambda identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Lambda and IAM.

Topics

- [I am not authorized to perform an action in Lambda \(p. 855\)](#)
- [I am not authorized to perform iam:PassRole \(p. 855\)](#)
- [I'm an administrator and want to migrate from AWS managed policies for Lambda that will be deprecated \(p. 856\)](#)
- [I want to allow people outside of my AWS account to access my Lambda resources \(p. 856\)](#)

I am not authorized to perform an action in Lambda

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your sign-in credentials.

The following example error occurs when the mateojackson user tries to use the console to view details about a function but does not have lambda:GetFunction permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
lambda:GetFunction on resource: my-function
```

In this case, Mateo asks his administrator to update his policies to allow him to access the my-function resource using the lambda:GetFunction action.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the iam:PassRole action, your policies must be updated to allow you to pass a role to Lambda.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named marymajor tries to use the console to perform an action in Lambda. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform: iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the iam:PassRole action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I'm an administrator and want to migrate from AWS managed policies for Lambda that will be deprecated

After March 1, 2021, the AWS managed policies **AWSLambdaReadOnlyAccess** and **AWSLambdaFullAccess** will be deprecated and can no longer be attached to new users. For more information about policy deprecations, see [Deprecated AWS managed policies](#) in the *IAM User Guide*.

Lambda has introduced two new AWS managed policies:

- The **AWSLambda_ReadOnlyAccess** policy grants read-only access to Lambda, Lambda console features, and other related AWS services. This policy was created by scoping down the previous policy **AWSLambdaReadOnlyAccess**.
- The **AWSLambda_FullAccess** policy grants full access to Lambda, Lambda console features, and other related AWS services. This policy was created by scoping down the previous policy **AWSLambdaFullAccess**.

Using the AWS managed policies

We recommend using the newly launched managed policies to grant users, groups, and roles access to Lambda; however, review the permissions granted in the policies to ensure they meet your requirements.

- To review the permissions of the **AWSLambda_ReadOnlyAccess** policy, see the [AWSLambda_ReadOnlyAccess](#) policy page in the IAM console.
- To review the permissions of the **AWSLambda_FullAccess** policy, see the [AWSLambda_FullAccess](#) policy page in the IAM console.

After reviewing the permissions, you can attach the policies to an IAM identity (groups, users, or roles). For instructions about attaching an AWS managed policy, see [Adding and removing IAM identity permissions](#) in the *IAM User Guide*.

Using customer managed policies

If you need more fine-grained access control or would like to add permissions, you can create your own [customer managed policies](#). For more information, see [Creating policies on the JSON tab](#) in the *IAM User Guide*.

I want to allow people outside of my AWS account to access my Lambda resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Lambda supports these features, see [How AWS Lambda works with IAM \(p. 853\)](#).
- To learn how to provide access to your resources across AWS accounts that you own, see [Providing access to an IAM user in another AWS account that you own](#) in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see [Providing access to AWS accounts owned by third parties](#) in the *IAM User Guide*.
- To learn how to provide access through identity federation, see [Providing access to externally authenticated users \(identity federation\)](#) in the *IAM User Guide*.

- To learn the difference between using roles and resource-based policies for cross-account access, see [How IAM roles differ from resource-based policies](#) in the *IAM User Guide*.

Compliance validation for AWS Lambda

Third-party auditors assess the security and compliance of AWS Lambda as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others.

For a list of AWS services in scope of specific compliance programs, see [AWS services in scope by compliance program](#). For general information, see [AWS compliance programs](#).

You can download third-party audit reports using AWS Artifact. For more information, see [Downloading reports in AWS artifact](#).

Your compliance responsibility when using Lambda is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- [Security and compliance quick start guides](#) – These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- [Architecting for HIPAA Security and Compliance on Amazon Web Services](#) – This whitepaper describes how companies can use AWS to create HIPAA-compliant applications.
- [AWS compliance resources](#) – This collection of workbooks and guides might apply to your industry and location.
- [AWS Config](#) – This AWS service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- [AWS Security Hub](#) – This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

Resilience in AWS Lambda

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see [AWS global infrastructure](#).

In addition to the AWS global infrastructure, Lambda offers several features to help support your data resiliency and backup needs.

- **Versioning** – You can use versioning in Lambda to save your function's code and configuration as you develop it. Together with aliases, you can use versioning to perform blue/green and rolling deployments. For details, see [Lambda function versions \(p. 83\)](#).
- **Scaling** – When your function receives a request while it's processing a previous request, Lambda launches another instance of your function to handle the increased load. Lambda automatically scales to handle 1,000 concurrent executions per Region, a [quota \(p. 1131\)](#) that can be increased if needed. For details, see [Lambda function scaling \(p. 197\)](#).
- **High availability** – Lambda runs your function in multiple Availability Zones to ensure that it is available to process events in case of a service interruption in a single zone. If you configure your

function to connect to a virtual private cloud (VPC) in your account, specify subnets in multiple Availability Zones to ensure high availability. For details, see [Configuring a Lambda function to access resources in a VPC \(p. 222\)](#).

- **Reserved concurrency** – To make sure that your function can always scale to handle additional requests, you can reserve concurrency for it. Setting reserved concurrency for a function ensures that it can scale to, but not exceed, a specified number of concurrent invocations. This ensures that you don't lose requests due to other functions consuming all of the available concurrency. For details, see [Configuring reserved concurrency \(p. 210\)](#).
- **Retries** – For asynchronous invocations and a subset of invocations triggered by other services, Lambda automatically retries on error with delays between retries. Other clients and AWS services that invoke functions synchronously are responsible for performing retries. For details, see [Error handling and automatic retries in AWS Lambda \(p. 161\)](#).
- **Dead-letter queue** – For asynchronous invocations, you can configure Lambda to send requests to a dead-letter queue if all retries fail. A dead-letter queue is an Amazon SNS topic or Amazon SQS queue that receives events for troubleshooting or reprocessing. For details, see [Dead-letter queues \(p. 129\)](#).

Infrastructure security in AWS Lambda

As a managed service, AWS Lambda is protected by the AWS global network security procedures that are described in the [Best Practices for Security, Identity, and Compliance](#).

You use AWS published API calls to access Lambda through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the [AWS Security Token Service](#) (AWS STS) to generate temporary security credentials to sign requests.

Configuration and vulnerability analysis in AWS Lambda

AWS Lambda provides [runtimes \(p. 37\)](#) that run your function code in an Amazon Linux-based execution environment. Lambda is responsible for keeping software in the runtime and execution environment up to date, releasing new runtimes for new languages and frameworks, and deprecating runtimes when the underlying software is no longer supported.

If you use additional libraries with your function, you're responsible for updating the libraries. You can include additional libraries in the [deployment package \(p. 881\)](#), or in [layers \(p. 93\)](#) that you attach to your function. You can also build [custom runtimes \(p. 59\)](#) and use layers to share them with other accounts.

Lambda deprecates runtimes when the software on the runtime or its execution environment reaches end of life. When Lambda deprecates a runtime, you're responsible for migrating your functions to a supported runtime for the same language or framework. For details, see [Runtime deprecation policy \(p. 39\)](#).

Sections

- [Detect vulnerabilities in your Lambda functions With Amazon Inspector \(p. 859\)](#)

Detect vulnerabilities in your Lambda functions With Amazon Inspector

You can use Amazon Inspector to detect security vulnerabilities in your Lambda functions and layers. Amazon Inspector is an automated vulnerability scanning service that discovers and reports vulnerabilities based on its vulnerability intelligence database. The Amazon Inspector vulnerability intelligence database sources data from internal AWS security research teams, paid vendor feeds, and industry-standard security advisories.

Amazon Inspector automatically creates an inventory of your active Lambda functions and layers then continuously monitors them for software package vulnerabilities. When Amazon Inspector discovers a vulnerability, it generates a finding that contains details about the security issue, and how to remediate the issue. You can view Amazon Inspector findings in the Amazon Inspector console or process them through other AWS services.

For information on activating and configuring Amazon Inspector Lambda scanning, see [Scanning Lambda functions with Amazon Inspector](#).

Monitoring and troubleshooting Lambda functions

AWS Lambda integrates with other AWS services to help you monitor and troubleshoot your Lambda functions. Lambda automatically monitors Lambda functions on your behalf and reports metrics through Amazon CloudWatch. To help you monitor your code when it runs, Lambda automatically tracks the number of requests, the invocation duration per request, and the number of requests that result in an error.

You can use other AWS services to troubleshoot your Lambda functions. This section describes how to use these AWS services to monitor, trace, debug, and troubleshoot your Lambda functions and applications. For details about function logging and errors in each runtime, see individual runtime sections.

For more information about monitoring Lambda applications, see [Monitoring and observability](#) in the *Lambda operator guide*.

Sections

- [Monitoring functions on the Lambda console \(p. 861\)](#)
- [Using Lambda Insights in Amazon CloudWatch \(p. 864\)](#)
- [Working with Lambda function metrics \(p. 870\)](#)
- [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 873\)](#)
- [Using CodeGuru Profiler with your Lambda function \(p. 876\)](#)
- [Example workflows using other AWS services \(p. 878\)](#)

Monitoring functions on the Lambda console

Lambda monitors functions on your behalf and sends metrics to Amazon CloudWatch. The Lambda console creates monitoring graphs for these metrics and shows them on the **Monitoring** page for each Lambda function.

This page describes the basics of using the Lambda console to view function metrics, including total requests, duration, and error rates.

Pricing

CloudWatch has a perpetual free tier. Beyond the free tier threshold, CloudWatch charges for metrics, dashboards, alarms, logs, and insights. For more information, see [Amazon CloudWatch pricing](#).

Using the Lambda console

You can monitor your Lambda functions and applications on the Lambda console.

To monitor a function

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose the **Monitor** tab.

Types of monitoring graphs

The following section describes the monitoring graphs on the Lambda console.

Lambda monitoring graphs

- **Invocations** – The number of times that the function was invoked.
- **Duration** – The average, minimum, and maximum amount of time your function code spends processing an event.
- **Error count and success rate (%)** – The number of errors and the percentage of invocations that completed without error.
- **Throttles** – The number of times that an invocation failed due to concurrency limits.
- **IteratorAge** – For stream event sources, the age of the last item in the batch when Lambda received it and invoked the function.
- **Async delivery failures** – The number of errors that occurred when Lambda attempted to write to a destination or dead-letter queue.
- **Concurrent executions** – The number of function instances that are processing events.

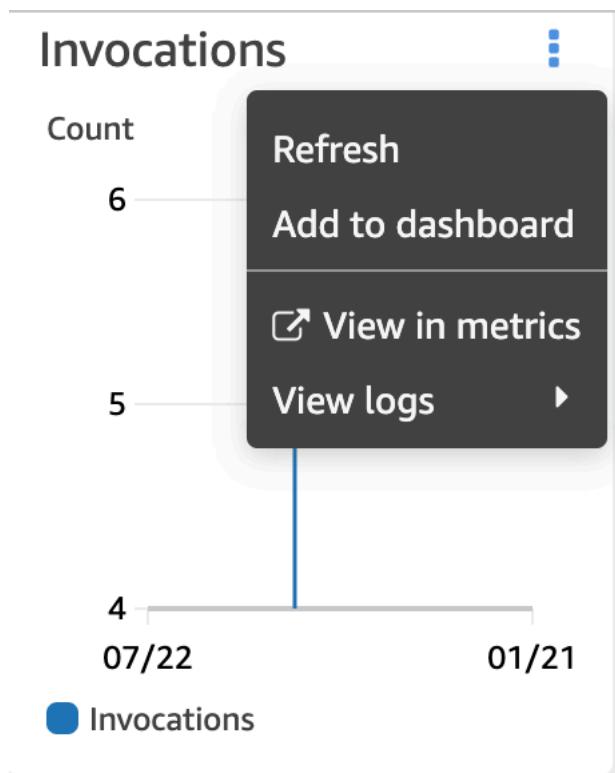
Viewing graphs on the Lambda console

The following section describes how to view CloudWatch monitoring graphs on the Lambda console, and open the CloudWatch metrics dashboard.

To view monitoring graphs for a function

1. Open the [Functions page](#) of the Lambda console.

2. Choose a function.
3. Choose the **Monitor** tab.
4. On the **Metrics, Logs, or Traces** tab, choose from the predefined time ranges, or choose a custom time range.
5. To see the definition of a graph in CloudWatch, choose the three vertical dots (**Widget actions**), and then choose **View in metrics** to open the **Metrics** dashboard on the CloudWatch console.



Viewing queries on the CloudWatch Logs console

The following section describes how to view and add reports from CloudWatch Logs Insights to a custom dashboard on the CloudWatch Logs console.

To view reports for a function

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose the **Monitor** tab.
4. Choose **View logs in CloudWatch**.
5. Choose **View in Logs Insights**.
6. Choose from the predefined time ranges, or choose a custom time range.
7. Choose **Run query**.
8. (Optional) Choose **Save**.

The screenshot shows the AWS CloudWatch Logs Insights interface. At the top, there is a dropdown menu labeled "Select log group(s)" containing the path "/aws/lambda/wear_heavy_coat". Below it is a "Clear" button and a search bar with the query text: "1 fields @timestamp, @message | sort @timestamp desc | limit 20". A date range selector shows "2020-05-01 (00:00:00) > 2020-12-31 (23:59:59)". Below the query area are three buttons: "Run query" (orange), "Save", and "History".

Below the query results, there are tabs for "Logs" (selected), "Visualization", "Export results", "Add to dashboard", and a gear icon. The "Logs" tab displays a histogram showing record counts by month from May to December. The histogram has a single prominent peak in October reaching a value of 30. Below the histogram is a table with columns "#", "@timestamp", and "@message". One row is shown: "# 1 2020-09-29T18:54:16.... {'Weather': 'FREEZING'}".

What's next?

- Learn about the metrics that Lambda records and sends to CloudWatch in [Working with Lambda function metrics \(p. 870\)](#).
- Learn how to use CloudWatch Lambda Insights to collect and aggregate Lambda function runtime performance metrics and logs in [Using Lambda Insights in Amazon CloudWatch \(p. 864\)](#).

Using Lambda Insights in Amazon CloudWatch

Amazon CloudWatch Lambda Insights collects and aggregates Lambda function runtime performance metrics and logs for your serverless applications. This page describes how to enable and use Lambda Insights to diagnose issues with your Lambda functions.

Sections

- [How Lambda Insights monitors serverless applications \(p. 864\)](#)
- [Pricing \(p. 864\)](#)
- [Supported runtimes \(p. 864\)](#)
- [Enabling Lambda Insights in the Lambda console \(p. 864\)](#)
- [Enabling Lambda Insights programmatically \(p. 865\)](#)
- [Using the Lambda Insights dashboard \(p. 865\)](#)
- [Example workflow to detect function anomalies \(p. 867\)](#)
- [Example workflow using queries to troubleshoot a function \(p. 868\)](#)
- [What's next? \(p. 863\)](#)

How Lambda Insights monitors serverless applications

CloudWatch Lambda Insights is a monitoring and troubleshooting solution for serverless applications running on AWS Lambda. The solution collects, aggregates, and summarizes system-level metrics including CPU time, memory, disk and network usage. It also collects, aggregates, and summarizes diagnostic information such as cold starts and Lambda worker shutdowns to help you isolate issues with your Lambda functions and resolve them quickly.

Lambda Insights uses a new CloudWatch Lambda Insights [extension](#), which is provided as a [Lambda layer \(p. 93\)](#). When you enable this extension on a Lambda function for a supported runtime, it collects system-level metrics and emits a single performance log event for every invocation of that Lambda function. CloudWatch uses embedded metric formatting to extract metrics from the log events. For more information, see [Using AWS Lambda extensions](#).

The Lambda Insights layer extends the `CreateLogStream` and `PutLogEvents` for the `/aws/lambda-insights/` log group.

Pricing

When you enable Lambda Insights for your Lambda function, Lambda Insights reports 8 metrics per function and every function invocation sends about 1KB of log data to CloudWatch. You only pay for the metrics and logs reported for your function by Lambda Insights. There are no minimum fees or mandatory service usage policies. You do not pay for Lambda Insights if the function is not invoked. For a pricing example, see [Amazon CloudWatch pricing](#).

Supported runtimes

You can use Lambda Insights with any of the runtimes that support [Lambda extensions \(p. 900\)](#).

Enabling Lambda Insights in the Lambda console

You can enable Lambda Insights enhanced monitoring on new and existing Lambda functions. When you enable Lambda Insights on a function in the Lambda console for a supported runtime, Lambda

adds the Lambda Insights [extension](#) as a layer to your function, and verifies or attempts to attach the [CloudWatchLambdaInsightsExecutionRolePolicy](#) policy to your function's [execution role](#).

To enable Lambda Insights in the Lambda console

1. Open the [Functions page](#) of the Lambda console.
2. Choose your function.
3. Choose the **Configuration** tab.
4. On the **Monitoring tools** pane, choose **Edit**.
5. Under **Lambda Insights**, turn on **Enhanced monitoring**.
6. Choose **Save**.

Enabling Lambda Insights programmatically

You can also enable Lambda Insights using the AWS Command Line Interface (AWS CLI), AWS Serverless Application Model (SAM) CLI, AWS CloudFormation, or the AWS Cloud Development Kit (AWS CDK). When you enable Lambda Insights programmatically on a function for a supported runtime, CloudWatch attaches the [CloudWatchLambdaInsightsExecutionRolePolicy](#) policy to your function's [execution role](#).

For more information, see [Getting started with Lambda Insights](#) in the *Amazon CloudWatch User Guide*.

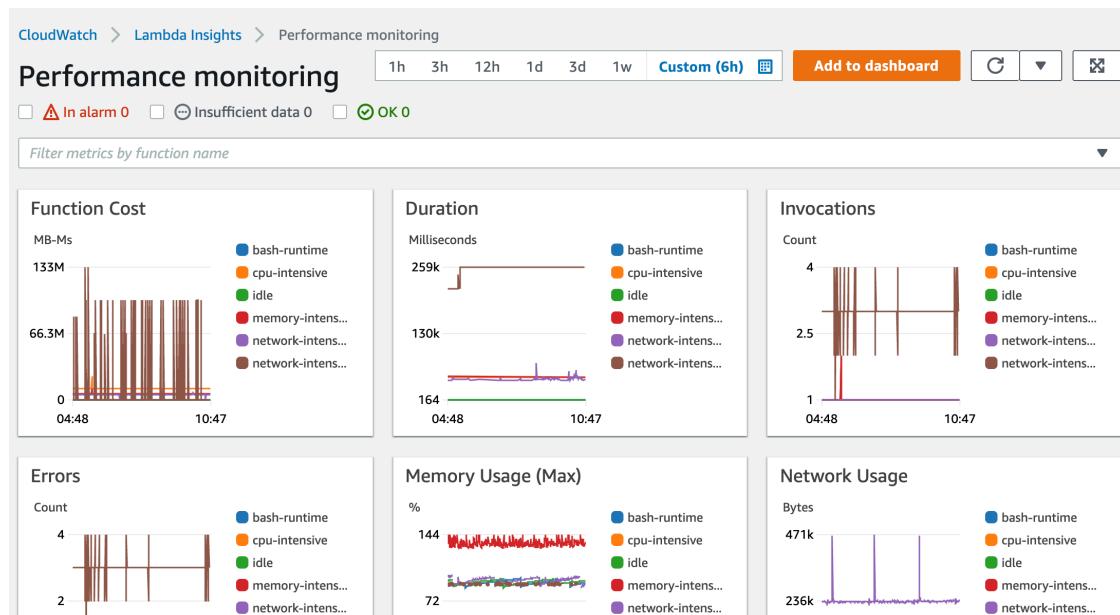
Using the Lambda Insights dashboard

The Lambda Insights dashboard has two views in the CloudWatch console: the multi-function overview and the single-function view. The multi-function overview aggregates the runtime metrics for the Lambda functions in the current AWS account and Region. The single-function view shows the available runtime metrics for a single Lambda function.

You can use the Lambda Insights dashboard multi-function overview in the CloudWatch console to identify over- and under-utilized Lambda functions. You can use the Lambda Insights dashboard single-function view in the CloudWatch console to troubleshoot individual requests.

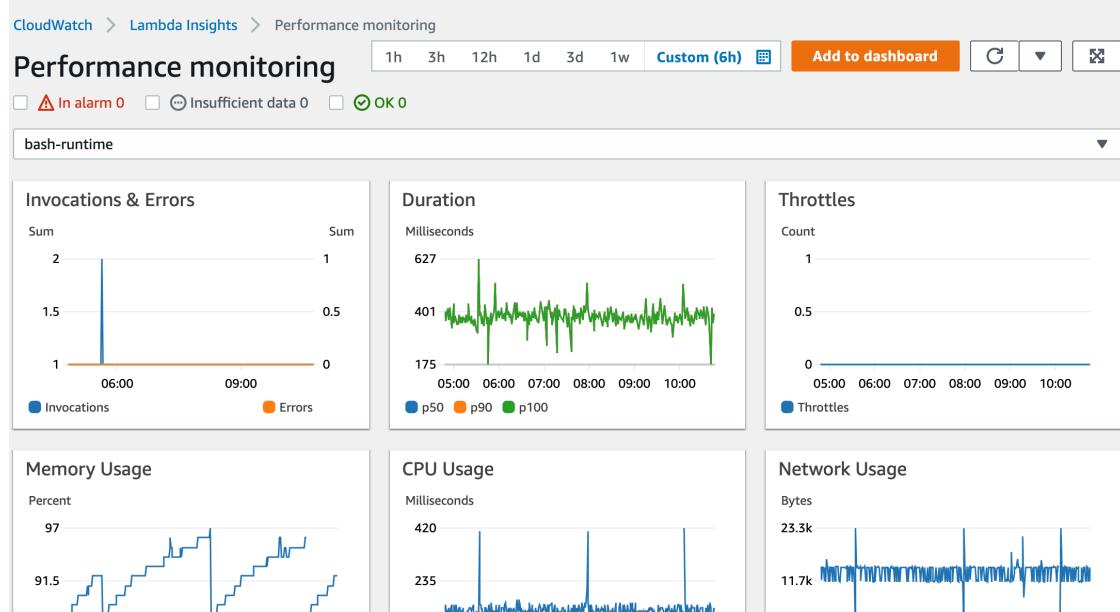
To view the runtime metrics for all functions

1. Open the [Multi-function](#) page in the CloudWatch console.
2. Choose from the predefined time ranges, or choose a custom time range.
3. (Optional) Choose **Add to dashboard** to add the widgets to your CloudWatch dashboard.



To view the runtime metrics of a single function

1. Open the [Single-function](#) page in the CloudWatch console.
2. Choose from the predefined time ranges, or choose a custom time range.
3. (Optional) Choose **Add to dashboard** to add the widgets to your CloudWatch dashboard.



For more information, see [Creating and working with widgets on CloudWatch dashboards](#).

Example workflow to detect function anomalies

You can use the multi-function overview on the Lambda Insights dashboard to identify and detect compute memory anomalies with your function. For example, if the multi-function overview indicates that a function is using a large amount of memory, you can view detailed memory utilization metrics in the **Memory Usage** pane. You can then go to the Metrics dashboard to enable anomaly detection or create an alarm.

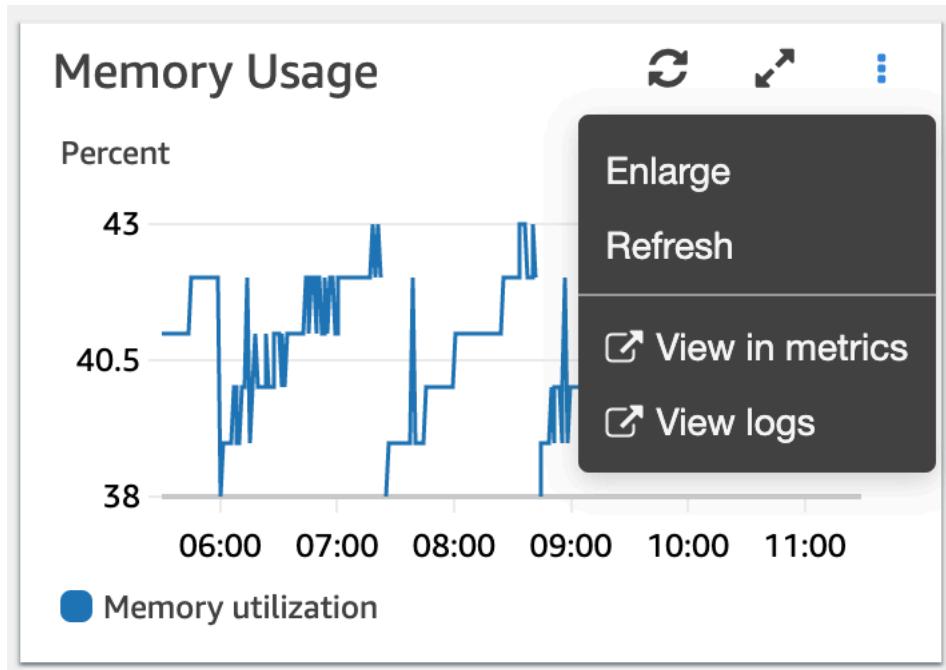
To enable anomaly detection for a function

1. Open the [Multi-function](#) page in the CloudWatch console.
2. Under **Function summary**, choose your function's name.

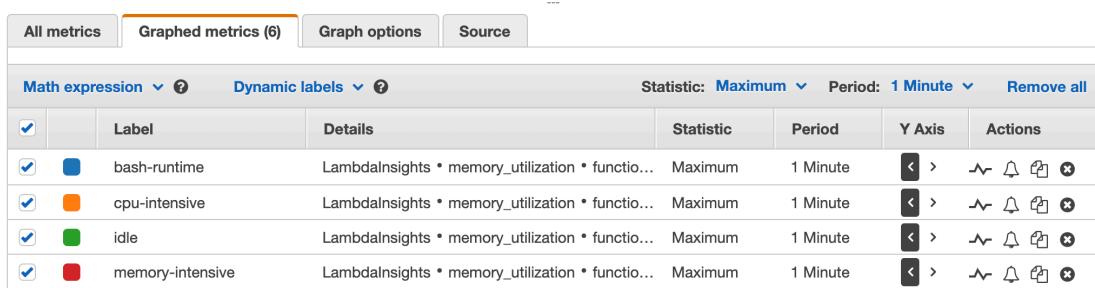
The single-function view opens with the function runtime metrics.

Function summary (6)							Actions					
	Function name	▲	Invocations	▼	CPU time	▼	Network IO	▼	Max. memory	▼	Cold starts	▼
<input type="checkbox"/>	bash-runtime		360		132.9167ms		4770 kB		<div style="width: 97%;">97%</div>		3	
<input type="checkbox"/>	cpu-intensive		359		6714.2897ms		4780 kB		<div style="width: 43%;">43%</div>		4	
<input type="checkbox"/>	idle		359		120.2507ms		4746 kB		<div style="width: 96%;">96%</div>		3	
<input type="checkbox"/>	memory-intensive		358		2385.9497ms		4794 kB		<div style="width: 144%;">144%</div>		4	
<input type="checkbox"/>	network-intensive		359		781.0585ms		82008 kB		<div style="width: 99%;">99%</div>		3	
<input type="checkbox"/>	network-intensive-vpc		43		2730.6977ms		95 kB		<div style="width: 91%;">91%</div>		43	

3. On the **Memory Usage** pane, choose the three vertical dots, and then choose **View in metrics** to open the Metrics dashboard.



4. On the **Graphed metrics** tab, in the **Actions** column, choose the first icon to enable anomaly detection for the function.



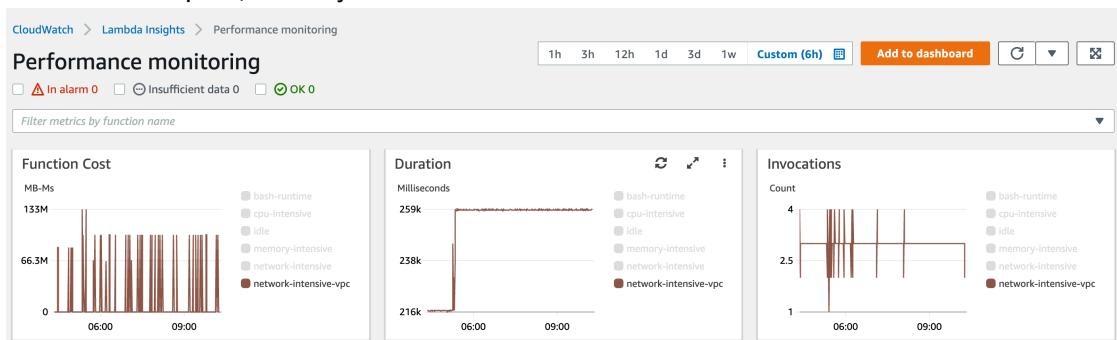
For more information, see [Using CloudWatch Anomaly Detection](#).

Example workflow using queries to troubleshoot a function

You can use the single-function view on the Lambda Insights dashboard to identify the root cause of a spike in function duration. For example, if the multi-function overview indicates a large increase in function duration, you can pause on or choose each function in the **Duration** pane to determine which function is causing the increase. You can then go to the single-function view and review the **Application logs** to determine the root cause.

To run queries on a function

1. Open the [Multi-function](#) page in the CloudWatch console.
 2. In the **Duration** pane, choose your function to filter the duration metrics.



3. Open the [Single-function](#) page.
 4. Choose the **Filter metrics by function name** dropdown list, and then choose your function.
 5. To view the **Most recent 1000 application logs**, choose the **Application logs** tab.
 6. Review the **Timestamp** and **Message** to identify the invocation request that you want to troubleshoot.

Most recent 1000 application logs (1000)		View logs
Timestamp	Message	
2020-09-30T16:24:36.121-06	0 0 0 0 0 0 0 --:--- 0:03:06 ---:--- 0	
2020-09-30T16:24:34.917-06	0 0 0 0 0 0 0 --:--- 0:04:15 ---:--- 0	
2020-09-30T16:24:34.120-06	0 0 0 0 0 0 0 --:--- 0:03:04 ---:--- 0	
2020-09-30T16:24:33.033-06	0 0 0 0 0 0 0 --:--- 0:01:26 ---:--- 0	

7. To show the **Most recent 1000 invocations**, choose the **Invocations** tab.
8. Select the **Timestamp** or **Message** for the invocation request that you want to troubleshoot.

	Timestamp	Request ID	Trace	Memory %	Network IO	CPU time	Cold start
<input checked="" type="checkbox"/>	2020-09-30 16:22:34 (UTC-06:00)	247e6369-3a2b...	-	<div style="width: 91%;">91%</div>	2 kB	2550ms	Yes
<input type="checkbox"/>	2020-09-30 16:13:39 (UTC-06:00)	311fb438-fa9d-4...	-	<div style="width: 90%;">90%</div>	2 kB	2340ms	Yes

9. Choose the **View logs** dropdown list, and then choose **View performance logs**.

An autogenerated query for your function opens in the **Logs Insights** dashboard.

10. Choose **Run query** to generate a **Logs** message for the invocation request.

Select log group(s) /aws/lambda-insights X Clear

```
1 fields @timestamp, @message
2 | filter function_name = "network-intensive-vpc"
3 | filter request_id = "247e6369-3a2b-4ccf-9e95-fb80c6ba711f"
4 | sort @timestamp desc
```

Run query Save History

Logs Visualization Export results Add to dashboard ⚙️

Showing 1 of 1 records matched ⓘ
1,856 records (2.0 MB) scanned in 4.0s @ 467 records/s (521.7 kB/s)

Hide histogram

#	@timestamp	@message
▶ 1	2020-09-30T16:22:34...	{"cpu_system_time":1520,"shutdown":1,"cpu_user_time":1030,"agent_memory_avg":7487349,"used_memory...}

What's next?

- Learn how to create a CloudWatch Logs dashboard in [Create a Dashboard](#) in the *Amazon CloudWatch User Guide*.
- Learn how to add queries to a CloudWatch Logs dashboard in [Add Query to Dashboard or Export Query Results](#) in the *Amazon CloudWatch User Guide*.

Working with Lambda function metrics

When your AWS Lambda function finishes processing an event, Lambda sends metrics about the invocation to Amazon CloudWatch. There is no charge for these metrics.

On the CloudWatch console, you can build graphs and dashboards with these metrics. You can set alarms to respond to changes in utilization, performance, or error rates. Lambda sends metric data to CloudWatch in 1-minute intervals. For more immediate insight into your Lambda function, you can create high-resolution [custom metrics](#) as described in the *AWS Lambda Operator Guide*. Charges apply for custom metrics and CloudWatch alarms. For more information, see [Amazon CloudWatch Pricing](#).

This page describes the Lambda function invocation, performance, and concurrency metrics available on the CloudWatch console.

Sections

- [Viewing metrics on the CloudWatch console \(p. 870\)](#)
- [Types of metrics \(p. 870\)](#)

Viewing metrics on the CloudWatch console

You can use the CloudWatch console to filter and sort function metrics by function name, alias, or version.

To view metrics on the CloudWatch console

1. Open the [Metrics page](#) (AWS/Lambda namespace) of the CloudWatch console.
2. On the **Browse** tab, under **Metrics**, choose any of the following dimensions:
 - **By Function Name** (`FunctionName`) – View aggregate metrics for all versions and aliases of a function.
 - **By Resource** (`Resource`) – View metrics for a version or alias of a function.
 - **By Executed Version** (`ExecutedVersion`) – View metrics for a combination of alias and version. Use the `ExecutedVersion` dimension to compare error rates for two versions of a function that are both targets of a [weighted alias \(p. 85\)](#).
 - **Across All Functions** (none) – View aggregate metrics for all functions in the current AWS Region.
3. Choose a metric, then choose **Add to graph** or another graphing option.

By default, graphs use the `Sum` statistic for all metrics. To choose a different statistic and customize the graph, use the options on the **Graphed metrics** tab.

Note

The timestamp on a metric reflects when the function was invoked. Depending on the duration of the invocation, this can be several minutes before the metric is emitted. For example, if your function has a 10-minute timeout, then look more than 10 minutes in the past for accurate metrics.

For more information about CloudWatch, see the [Amazon CloudWatch User Guide](#).

Types of metrics

The following section describes the types of Lambda metrics available on the CloudWatch console.

Invocation metrics

Invocation metrics are binary indicators of the outcome of a Lambda function invocation. For example, if the function returns an error, then Lambda sends the **Errors** metric with a value of 1. To get a count of the number of function errors that occurred each minute, view the **Sum** of the **Errors** metric with a period of 1 minute.

Note

View the following invocation metrics with the **Sum** statistic.

- **Invocations** – The number of times that your function code is invoked, including successful invocations and invocations that result in a function error. Invocations aren't recorded if the invocation request is throttled or otherwise results in an invocation error. The value of **Invocations** equals the number of requests billed.
- **Errors** – The number of invocations that result in a function error. Function errors include exceptions that your code throws and exceptions that the Lambda runtime throws. The runtime returns errors for issues such as timeouts and configuration errors. To calculate the error rate, divide the value of **Errors** by the value of **Invocations**. Note that the timestamp on an error metric reflects when the function was invoked, not when the error occurred.
- **DeadLetterErrors** – For [asynchronous invocation \(p. 123\)](#), the number of times that Lambda attempts to send an event to a dead-letter queue (DLQ) but fails. Dead-letter errors can occur due to misconfigured resources or size limits.
- **DestinationDeliveryFailures** – For asynchronous invocation and supported [event source mappings](#), the number of times that Lambda attempts to send an event to a [destination \(p. 34\)](#) but fails. For event source mappings, Lambda supports destinations for stream sources (DynamoDB and Kinesis). Delivery errors can occur due to permissions errors, misconfigured resources, or size limits. Errors can also occur if the destination you have configured is an unsupported type such as an Amazon SQS FIFO queue or an Amazon SNS FIFO topic.
- **Throttles** – The number of invocation requests that are throttled. When all function instances are processing requests and no concurrency is available to scale up, Lambda rejects additional requests with a `TooManyRequestsException` error. Throttled requests and other invocation errors don't count as either **Invocations** or **Errors**.
- **ProvisionedConcurrencyInvocations** – The number of times that your function code is invoked using [provisioned concurrency \(p. 213\)](#).
- **ProvisionedConcurrencySpilloverInvocations** – The number of times that your function code is invoked using standard concurrency when all provisioned concurrency is in use.

Performance metrics

Performance metrics provide performance details about a single function invocation. For example, the **Duration** metric indicates the amount of time in milliseconds that your function spends processing an event. To get a sense of how fast your function processes events, view these metrics with the **Average** or **Max** statistic.

- **Duration** – The amount of time that your function code spends processing an event. The billed duration for an invocation is the value of **Duration** rounded up to the nearest millisecond.
- **PostRuntimeExtensionsDuration** – The cumulative amount of time that the runtime spends running code for extensions after the function code has completed.
- **IteratorAge** – For [event source mappings \(p. 131\)](#) that read from streams, the age of the last record in the event. This metric measures the time between when a stream receives the record and when the event source mapping sends the event to the function.
- **OffsetLag** – For self-managed Apache Kafka and Amazon Managed Streaming for Apache Kafka (Amazon MSK) event sources, the difference in offset between the last record written to a topic and

the last record that your function's consumer group processed. Though a Kafka topic can have multiple partitions, this metric measures the offset lag at the topic level.

Duration also supports percentile (p) statistics. Use percentiles to exclude outlier values that skew Average and Maximum statistics. For example, the p95 statistic shows the maximum duration of 95 percent of invocations, excluding the slowest 5 percent. For more information, see [Percentiles](#) in the [Amazon CloudWatch User Guide](#).

Concurrency metrics

Lambda reports concurrency metrics as an aggregate count of the number of instances processing events across a function, version, alias, or AWS Region. To see how close you are to hitting [concurrency limits \(p. 207\)](#), view these metrics with the Max statistic.

- ConcurrentExecutions – The number of function instances that are processing events. If this number reaches your [concurrent executions quota \(p. 1131\)](#) for the Region, or the [reserved concurrency \(p. 210\)](#) limit on the function, then Lambda throttles additional invocation requests.
- ProvisionedConcurrentExecutions – The number of function instances that are processing events using [provisioned concurrency \(p. 213\)](#). For each invocation of an alias or version with provisioned concurrency, Lambda emits the current count.
- ProvisionedConcurrencyUtilization – For a version or alias, the value of ProvisionedConcurrentExecutions divided by the total amount of provisioned concurrency allocated. For example, .5 indicates that 50 percent of allocated provisioned concurrency is in use.
- UnreservedConcurrentExecutions – For a Region, the number of events that functions without reserved concurrency are processing.

Asynchronous invocation metrics

Asynchronous invocation metrics provide details about asynchronous invocations from event sources and direct invocations. You can set thresholds and alarms to notify you of certain changes. For example, when there's an undesired increase in the number of events queued for processing (`AsyncEventsReceived`). Or, when an event has been waiting a long time to be processed (`AsyncEventAge`).

- AsyncEventsReceived – The number of events that Lambda successfully queues for processing. This metric provides insight into the number of events that a Lambda function receives. Monitor this metric and set alarms for thresholds to check for issues. For example, to detect an undesirable number of events sent to Lambda, and to quickly diagnose issues resulting from incorrect trigger or function configurations. Mismatches between `AsyncEventsReceived` and `Invocations` can indicate a disparity in processing, events being dropped, or a potential queue backlog.
- AsyncEventAge – The time between when Lambda successfully queues the event and when the function is invoked. The value of this metric increases when events are being retried due to invocation failures or throttling. Monitor this metric and set alarms for thresholds on different statistics for when a queue buildup occurs. To troubleshoot an increase in this metric, look at the `Errors` metric to identify function errors and the `Throttles` metric to identify concurrency issues.
- AsyncEventsDropped – The number of events that are dropped without successfully executing the function. If you configure a dead-letter queue (DLQ) or `OnFailure` destination, then events are sent there before they're dropped. Events are dropped for various reasons. For example, events can exceed the maximum event age or exhaust the maximum retry attempts, or reserved concurrency might be set to 0. To troubleshoot why events are dropped, look at the `Errors` metric to identify function errors and the `Throttles` metric to identify concurrency issues.

Accessing Amazon CloudWatch logs for AWS Lambda

AWS Lambda automatically monitors Lambda functions on your behalf, pushing logs to Amazon CloudWatch. To help you troubleshoot failures in a function, after you set up permissions, Lambda logs all requests handled by your function and also automatically stores logs generated by your code through Amazon CloudWatch Logs.

You can insert logging statements into your code to help you validate that your code is working as expected. Lambda automatically integrates with CloudWatch Logs and pushes all logs from your code to a CloudWatch Logs group associated with a Lambda function, which is named `/aws/lambda/<function name>`.

You can view logs for Lambda functions using the Lambda console, the CloudWatch console, the AWS Command Line Interface (AWS CLI), or the CloudWatch API.

Note

It may take 5 to 10 minutes for logs to show up after a function invocation.

Section

- [Prerequisites \(p. 873\)](#)
- [Pricing \(p. 873\)](#)
- [Using the Lambda console \(p. 873\)](#)
- [Using the AWS Command Line Interface \(p. 874\)](#)
- [Runtime function logging \(p. 876\)](#)
- [What's next? \(p. 876\)](#)

Prerequisites

Your [execution role \(p. 816\)](#) needs permission to upload logs to CloudWatch Logs. You can add CloudWatch Logs permissions using the `AWSLambdaBasicExecutionRole` AWS managed policy provided by Lambda. To add this policy to your role, run the following command:

```
aws iam attach-role-policy --role-name your-role --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

For more information, see [AWS managed policies for Lambda features \(p. 818\)](#).

Pricing

There is no additional charge for using Lambda logs; however, standard CloudWatch Logs charges apply. For more information, see [CloudWatch pricing](#).

Using the Lambda console

To view logs using the Lambda console

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose **Monitor**.
4. Choose **View logs in CloudWatch**.

Using the AWS Command Line Interface

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- [AWS Command Line Interface \(AWS CLI\) version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

You can use the [AWS CLI](#) to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{  
    "StatusCode": 200,  
    "LogResult":  
        "U1RBU1QgUmVxdWVzdElk0iA4N2QwNDRi0C1mMTU0LTExZTgt0GNkYS0y0Tc0YzVlNGZiMjEgVmVyc2lvb...",  
    "ExecutedVersion": "$LATEST"  
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \  
--query 'LogResult' --output text | base64 -d
```

You should see the following output:

```
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
```

The `base64` utility is available on Linux, macOS, and [Ubuntu on Windows](#). macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The `cli-binary-format` option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

```
#!/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 and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version: $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf \tINFO\tENVIRONMENT VARIABLES\r{\r\t\t\"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\", \r\t\t...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf \tINFO\tEVENT\r{\r\t\t\"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"
}
```

Runtime function logging

To debug and validate that your code is working as expected, you can output logs with the standard logging functionality for your programming language. The Lambda runtime uploads your function's log output to CloudWatch Logs. For language-specific instructions, see the following topics:

- [AWS Lambda function logging in Node.js \(p. 273\)](#)
- [AWS Lambda function logging in Python \(p. 334\)](#)
- [AWS Lambda function logging in Ruby \(p. 372\)](#)
- [AWS Lambda function logging in Java \(p. 410\)](#)
- [AWS Lambda function logging in Go \(p. 473\)](#)
- [Lambda function logging in C# \(p. 506\)](#)
- [AWS Lambda function logging in PowerShell \(p. 534\)](#)

What's next?

- Learn more about log groups and accessing them through the CloudWatch console in [Monitoring system, application, and custom log files](#) in the *Amazon CloudWatch User Guide*.

Using CodeGuru Profiler with your Lambda function

You can use Amazon CodeGuru Profiler to gain insights into runtime performance of your Lambda functions. This page describes how to activate CodeGuru Profiler from the Lambda console.

Sections

- [Supported runtimes \(p. 876\)](#)
- [Activating CodeGuru Profiler from the Lambda console \(p. 877\)](#)
- [What happens when you activate CodeGuru Profiler from the Lambda console? \(p. 877\)](#)
- [What's next? \(p. 877\)](#)

Supported runtimes

You can activate CodeGuru Profiler from the Lambda console if your function's runtime is Python3.8, Python3.9, Java 8 with Amazon Linux 2, or Java 11. For additional runtime versions, you can activate CodeGuru Profiler manually.

- For Java runtimes, see [Profiling your Java applications that run on AWS Lambda](#).
- For Python runtimes, see [Profiling your Python applications that run on AWS Lambda](#).

Activating CodeGuru Profiler from the Lambda console

This section describes how to activate CodeGuru Profiler from the Lambda console.

To activate CodeGuru Profiler from the Lambda console

1. Open the [Functions page](#) of the Lambda console.
2. Choose your function.
3. Choose the **Configuration** tab.
4. On the **Monitoring and operations tools** pane, choose **Edit**.
5. Under **Amazon CodeGuru Profiler**, turn on **Code profiling**.
6. Choose **Save**.

After activation, CodeGuru automatically creates a profiler group with the name `aws-lambda-<your-function-name>`. You can change the name from the CodeGuru console.

What happens when you activate CodeGuru Profiler from the Lambda console?

When you activate CodeGuru Profiler from the console, Lambda automatically does the following on your behalf:

- Lambda adds a CodeGuru Profiler layer to your function. For more details, see [Use AWS Lambda layers](#) in the *Amazon CodeGuru Profiler User Guide*.
- Lambda also adds environment variables to your function. The exact value varies based on the runtime.

Environment variables

Runtimes	Key	Value
java8.al2, java11	JAVA_TOOL_OPTIONS	<code>-javaagent:/opt/codeguru-profiler-java-agent-standalone.jar</code>
python3.8, python3.9	AWS_LAMBDA_EXEC_WRAPPER	<code>/opt/codeguru_profiler_lambda_exec</code>

- Lambda adds the `AmazonCodeGuruProfilerAgentAccess` policy to your function's execution role.

Note

When you deactivate CodeGuru Profiler from the console, Lambda automatically removes the CodeGuru Profiler layer from your function. However, Lambda does not remove the environment variables or the `AmazonCodeGuruProfilerAgentAccess` policy from your execution role.

What's next?

- Learn more about the data collected by your profiler group in [Working with visualizations](#) in the *Amazon CodeGuru Profiler User Guide*.

Example workflows using other AWS services

AWS Lambda integrates with other AWS services to help you monitor, trace, debug, and troubleshoot your Lambda functions. This page shows workflows you can use with AWS X-Ray, AWS Trusted Advisor and CloudWatch ServiceLens to trace and troubleshoot your Lambda functions.

Sections

- [Prerequisites \(p. 878\)](#)
- [Pricing \(p. 879\)](#)
- [Example AWS X-Ray workflow to view a service map \(p. 879\)](#)
- [Example AWS X-Ray workflow to view trace details \(p. 879\)](#)
- [Example AWS Trusted Advisor workflow to view recommendations \(p. 880\)](#)
- [What's next? \(p. 880\)](#)

Prerequisites

The following section describes the steps to using AWS X-Ray and Trusted Advisor to troubleshoot your Lambda functions.

Using AWS X-Ray

AWS X-Ray needs to be enabled on the Lambda console to complete the AWS X-Ray workflows on this page. If your execution role does not have the required permissions, the Lambda console will attempt to add them to your execution role.

To enable AWS X-Ray on the Lambda console

1. Open the [Functions page](#) of the Lambda console.
2. Choose your function.
3. Choose the **Configuration** tab.
4. On the **Monitoring tools** pane, choose **Edit**.
5. Under **AWS X-Ray**, turn on **Active tracing**.
6. Choose **Save**.

Using AWS Trusted Advisor

AWS Trusted Advisor inspects your AWS environment and makes recommendations on ways you can save money, improve system availability and performance, and help close security gaps. You can use Trusted Advisor checks to evaluate the Lambda functions and applications in your AWS account. The checks provide recommended steps to take and resources for more information.

- For more information on AWS support plans for Trusted Advisor checks, see [Support plans](#).
- For more information about the checks for Lambda, see [AWS Trusted Advisor best practice checklist](#).
- For more information on how to use the Trusted Advisor console, see [Get started with AWS Trusted Advisor](#).
- For instructions on how to allow and deny console access to Trusted Advisor, see [IAM policy examples](#).

Pricing

- With AWS X-Ray you pay only for what you use, based on the number of traces recorded, retrieved, and scanned. For more information, see [AWS X-Ray Pricing](#).
- Trusted Advisor cost optimization checks are included with AWS Business and Enterprise support subscriptions. For more information, see [AWS Trusted Advisor Pricing](#).

Example AWS X-Ray workflow to view a service map

If you've enabled AWS X-Ray, you can view a ServiceLens service map on the CloudWatch console. A service map displays your service endpoints and resources as nodes and highlights the traffic, latency, and errors for each node and its connections.

You can choose a node to see detailed insights about the correlated metrics, logs, and traces associated with that part of the service. This enables you to investigate problems and their effect on an application.

To view service map and traces using the CloudWatch console

- Open the [Functions page](#) of the Lambda console.
- Choose a function.
- Choose **Monitoring**.
- Choose **View traces in X-Ray**.
- Choose **Service map**.
- Choose from the predefined time ranges, or choose a custom time range.
- To troubleshoot requests, choose a filter.

Example AWS X-Ray workflow to view trace details

If you've enabled AWS X-Ray, you can use the single-function view on the CloudWatch Lambda Insights dashboard to show the distributed trace data of a function invocation error. For example, if the application logs message shows an error, you can open the ServiceLens traces view to see the distributed trace data and the other services handling the transaction.

To view trace details of a function

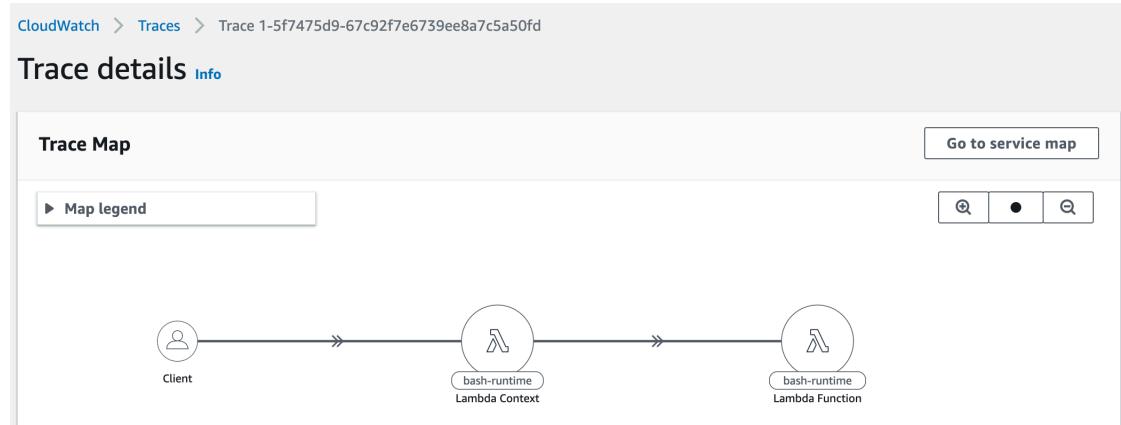
- Open the [single-function view](#) in the CloudWatch console.
- Choose the **Application logs** tab.
- Use the **Timestamp** or **Message** to identify the invocation request that you want to troubleshoot.
- To show the **Most recent 1000 invocations**, choose the **Invocations** tab.

The screenshot shows a table titled "Most recent 1000 invocations (359)". The table has columns: Request ID, Trace, Memory %, and Network IO. The first three rows of data are as follows:

Request ID	Trace	Memory %	Network IO
00c99bab-92f7-46cc-af28-ca71ad43f894	View	91%	14 kB
01fd5427-f3cd-4689-a39e-19f59c3eb7a2	View	91%	11 kB

5. Choose the **Request ID** column to sort entries in ascending alphabetical order.
6. In the **Trace** column, choose **View**.

The **Trace details** page opens in the ServiceLens traces view.



Example AWS Trusted Advisor workflow to view recommendations

Trusted Advisor checks Lambda functions in all AWS Regions to identify functions with the highest potential cost savings, and deliver actionable recommendations for optimization. It analyzes your Lambda usage data such as function execution time, billed duration, memory used, memory configured, timeout configuration and errors.

For example, the *Lambda Functions with High Error Rate* check recommends that you use AWS X-Ray or CloudWatch to detect errors with your Lambda functions.

To check for functions with high error rates

1. Open the [Trusted Advisor](#) console.
2. Choose the **Cost Optimization** category.
3. Scroll down to **AWS Lambda Functions with High Error Rates**. Expand the section to see the results and the recommended actions.

What's next?

- Learn more about how to integrate traces, metrics, logs, and alarms in [Using ServiceLens to Monitor the Health of Your Applications](#).
- Learn more about how to get a list of Trusted Advisor checks in [Using Trusted Advisor as a web service](#).

Creating Lambda container images

AWS provides a set of open-source [base images \(p. 882\)](#) that you can use to create your container image. These base images include a [runtime interface client \(p. 883\)](#) to manage the interaction between Lambda and your function code.

For example applications, including a Node.js example and a Python example, see [Container image support for Lambda](#) on the AWS Blog.

To reduce the time it takes for Lambda container functions to become active, see [Use multi-stage builds](#) in the Docker documentation. To build efficient container images, follow the [Best practices for writing Dockerfiles](#).

Note

Container images aren't supported for Lambda functions in the Middle East (UAE) Region.

Topics

- [Base images for Lambda \(p. 882\)](#)
- [Testing Lambda container images locally \(p. 884\)](#)
- [Prerequisites \(p. 887\)](#)
- [Image types \(p. 887\)](#)
- [Container tools \(p. 887\)](#)
- [Container image settings \(p. 888\)](#)
- [Creating images from AWS base images \(p. 888\)](#)
- [Creating images from alternative base images \(p. 890\)](#)
- [Upload the image to the Amazon ECR repository \(p. 891\)](#)
- [Create an image using the AWS SAM toolkit \(p. 893\)](#)

Base images for Lambda

AWS provides a set of open-source base images that you can use. You can also use a preferred community or private base image. Lambda provides client software that you add to your preferred base image to make it compatible with the Lambda service.

Note

Each base image is compatible with one or more of the instruction set architectures that Lambda supports. You need to build the function image for only one architecture. Lambda does not support multi-architecture images.

Topics

- [AWS base images for Lambda \(p. 882\)](#)
- [Base images for custom runtimes \(p. 882\)](#)
- [Runtime interface clients \(p. 883\)](#)
- [Runtime interface emulator \(p. 883\)](#)

AWS base images for Lambda

You can use one of the AWS base images for Lambda to build the container image for your function code. The base images are preloaded with a language runtime and other components required to run a container image on Lambda. You add your function code and dependencies to the base image and then package it as a container image.

AWS will maintain and regularly update these images. In addition, AWS will release AWS base images when any new managed runtime becomes available.

Lambda provides base images for the following runtimes:

- [Node.js \(p. 266\)](#)
- [Python \(p. 329\)](#)
- [Java \(p. 400\)](#)
- [.NET \(p. 502\)](#)
- [Go \(p. 465\)](#)
- [Ruby \(p. 368\)](#)

Base images for custom runtimes

AWS provides base images that contain the required Lambda components and the Amazon Linux or Amazon Linux2 operating system. You can add your preferred runtime, dependencies and code to these images.

Tags	Runtime	Operating system
al2	provided.al2	Amazon Linux 2
alami	provided	Amazon Linux

Amazon ECR Public Gallery: [gallery.ecr.aws/lambda/provided](#)

Runtime interface clients

The runtime interface client in your container image manages the interaction between Lambda and your function code. The [Runtime API \(p. 54\)](#), along with the [Extensions API \(p. 900\)](#), defines a simple HTTP interface for runtimes to receive invocation events from Lambda and respond with success or failure indications.

Each of the AWS base images for Lambda include a runtime interface client. If you choose one of the base images for custom runtimes or an alternative base image, you need to add the appropriate runtime interface client.

For your convenience, Lambda provides an open source runtime interface client for each of the supported Lambda runtimes:

- [Node.js \(p. 270\)](#)
- [Python \(p. 331\)](#)
- [Java \(p. 406\)](#)
- [.NET \(p. 504\)](#)
- [Go \(p. 466\)](#)
- [Ruby \(p. 369\)](#)

Runtime interface emulator

Lambda provides a runtime interface emulator (RIE) for you to test your function locally. The AWS base images for Lambda and base images for custom runtimes include the RIE. For other base images, you can download the [Runtime interface emulator](#) from the AWS GitHub repository.

Testing Lambda container images locally

The AWS Lambda Runtime Interface Emulator (RIE) is a proxy for the Lambda Runtime API that allows you to locally test your Lambda function packaged as a container image. The emulator is a lightweight web server that converts HTTP requests into JSON events to pass to the Lambda function in the container image.

The AWS base images for Lambda include the RIE component. If you use an alternate base image, you can test your image without adding RIE to the image. You can also build the RIE component into your base image. AWS provides an open-sourced RIE component on the AWS GitHub repository. Note that there are separate RIE components for the x86-64 architecture and the arm64 architecture.

You can use the emulator to test whether your function code is compatible with the Lambda environment. Also use the emulator to test that your Lambda function runs to completion successfully and provides the expected output. If you build extensions and agents into your container image, you can use the emulator to test that the extensions and agents work correctly with the Lambda Extensions API.

For examples of how to use the RIE, see [Container image support for Lambda](#) on the AWS Blog.

Topics

- [Guidelines for using the RIE \(p. 884\)](#)
- [Environment variables \(p. 884\)](#)
- [Test an image with RIE included in the image \(p. 885\)](#)
- [Build RIE into your base image \(p. 885\)](#)
- [Test an image without adding RIE to the image \(p. 886\)](#)

Guidelines for using the RIE

Note the following guidelines when using the Runtime Interface Emulator:

- The RIE does not emulate Lambda's security and authentication configurations, or Lambda orchestration.
- Lambda provides an emulator for each of the instruction set architectures.
- The emulator does not support AWS X-Ray tracing or other Lambda integrations.

Environment variables

The runtime interface emulator supports a subset of [environment variables \(p. 76\)](#) for the Lambda function in the local running image.

If your function uses security credentials, you can configure the credentials by setting the following environment variables:

- AWS_ACCESS_KEY_ID
- AWS_SECRET_ACCESS_KEY
- AWS_SESSION_TOKEN
- AWS_REGION

To set the function timeout, configure AWS_LAMBDA_FUNCTION_TIMEOUT. Enter the maximum number of seconds that you want to allow the function to run.

The emulator does not populate the following Lambda environment variables. However, you can set them to match the values that you expect when the function runs in the Lambda service:

- AWS_LAMBDA_FUNCTION_VERSION
- AWS_LAMBDA_FUNCTION_NAME
- AWS_LAMBDA_FUNCTION_MEMORY_SIZE

Test an image with RIE included in the image

The AWS base images for Lambda include the runtime interface emulator. You can also follow these steps after you build the RIE into your alternative base image.

To test your Lambda function with the emulator

1. Build your image locally using the `docker build` command.

```
docker build -t myfunction:latest .
```

2. Run your container image locally using the `docker run` command.

```
docker run -p 9000:8080 myfunction:latest
```

This command runs the image as a container and starts up an endpoint locally at `localhost:9000/2015-03-31/functions/function/invocations`.

3. From a new terminal window, post an event to the following endpoint using a `curl` command:

```
curl -XPOST "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the Lambda function running in the container image and returns a response.

Build RIE into your base image

You can build RIE into a base image. The following steps show how to download the RIE from GitHub to your local machine and update your Dockerfile to install RIE.

To build the emulator into your image

1. Create a script and save it in your project directory. Set execution permissions for the script file.

The script checks for the presence of the `AWS_LAMBDA_RUNTIME_API` environment variable, which indicates the presence of the runtime API. If the runtime API is present, the script runs the [runtime interface client \(p. 883\)](#). Otherwise, the script runs the runtime interface emulator.

The following example shows a typical script for a Node.js function.

```
#!/bin/sh
if [ -z "${AWS_LAMBDA_RUNTIME_API}" ]; then
  exec /usr/local/bin/aws-lambda-rie /usr/local/bin/npx aws-lambda-ric $@
else
  exec /usr/local/bin/npx aws-lambda-ric $@
fi
```

The following example shows a typical script for a Python function.

```
#!/bin/sh
if [ -z "${AWS_LAMBDA_RUNTIME_API}" ]; then
    exec /usr/local/bin/aws-lambda-rie /usr/local/bin/python -m awslambdaric $@
else
    exec /usr/local/bin/python -m awslambdaric $@
fi
```

2. Download the runtime interface emulator for your target architecture from GitHub into your project directory. Lambda provides an emulator for each of the instruction set architectures.
 - x86_64 – Download [aws-lambda-rie](#)
 - arm64 – Download [aws-lambda-rie-arm64](#)
3. Copy the script, install the emulator package, and change ENTRYPPOINT to run the new script by adding the following lines to your Dockerfile.

To use the default x86-64 architecture:

```
COPY ./entry_script.sh /entry_script.sh
ADD aws-lambda-rie-x86_64 /usr/local/bin/aws-lambda-rie
ENTRYPOINT [ "/entry_script.sh" ]
```

To use the arm64 architecture:

```
COPY ./entry_script.sh /entry_script.sh
ADD aws-lambda-rie-arm64 /usr/local/bin/aws-lambda-rie
ENTRYPOINT [ "/entry_script.sh" ]
```

4. Build your image locally using the docker build command.

```
docker build -t myfunction:latest .
```

Test an image without adding RIE to the image

You install the runtime interface emulator to your local machine. When you run the container image, you set the entry point to be the emulator.

To test an image without adding RIE to the image

1. From your project directory, run the following command to download the RIE (x86-64 architecture) from GitHub and install it on your local machine.

```
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
```

To download the RIE for arm64 architecture, use the previous command with a different GitHub download url.

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/
aws-lambda-rie-arm64 \
```

2. Run your Lambda function using the docker run command.

```
docker run -d -v ~/.aws-lambda-rie:/aws-lambda -p 9000:8080 \
--entrypoint /aws-lambda/aws-lambda-rie hello-world:latest <image entrypoint> \
<(optional) image command>
```

This runs the image as a container and starts up an endpoint locally at `localhost:9000/2015-03-31/functions/function/invocations`.

3. Post an event to the following endpoint using a curl command:

```
curl -XPOST "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function running in the container image and returns a response.

Prerequisites

Install the [AWS Command Line Interface \(AWS CLI\) version 2](#) and the [Docker CLI](#). Additionally, note the following requirements:

- The container image must implement the Lambda [Runtime API \(p. 54\)](#). The AWS open-source [runtime interface clients \(p. 883\)](#) implement the API. You can add a runtime interface client to your preferred base image to make it compatible with Lambda.
- The container image must be able to run on a read-only file system. Your function code can access a writable `/tmp` directory with between 512 MB and 10,240 MB, in 1-MB increments, of storage.
- The default Lambda user must be able to read all the files required to run your function code. Lambda follows security best practices by defining a default Linux user with least-privileged permissions. Verify that your application code does not rely on files that other Linux users are restricted from running.
- Lambda supports only Linux-based container images.
- Lambda provides multi-architecture base images. However, the image you build for your function must target only one of the architectures. Lambda does not support functions that use multi-architecture container images.

Image types

You can use an AWS provided base image or an alternative base image, such as Alpine or Debian. Lambda supports any image that conforms to one of the following image manifest formats:

- Docker image manifest V2, schema 2 (used with Docker version 1.10 and newer)
- Open Container Initiative (OCI) Specifications (v1.0.0 and up)

Lambda supports a maximum uncompressed image size of 10 GB, including all layers.

Container tools

To create your container image, you can use any development tool that supports one of the following container image manifest formats:

- Docker image manifest V2, schema 2 (used with Docker version 1.10 and newer)
- OCI Specifications (v1.0.0 and up)

For example, you can use the Docker CLI to build, test, and deploy your container images.

Container image settings

Lambda supports the following container image settings in the Dockerfile:

- **ENTRYPOINT** – Specifies the absolute path to the entry point of the application.
- **CMD** – Specifies parameters that you want to pass in with ENTRYPPOINT.
- **WORKDIR** – Specifies the absolute path to the working directory.
- **ENV** – Specifies an environment variable for the Lambda function.

Note

Lambda ignores the values of any unsupported container image settings in the Dockerfile.

For more information about how Docker uses the container image settings, see [ENTRYPOINT](#) in the Dockerfile reference on the Docker Docs website. For more information about using ENTRYPPOINT and CMD, see [Demystifying ENTRYPPOINT and CMD in Docker](#) on the AWS Open Source Blog.

You can specify the container image settings in the Dockerfile when you build your image. You can also override these configurations using the Lambda console or Lambda API. This allows you to deploy multiple functions that deploy the same container image but with different runtime configurations.

Warning

When you specify ENTRYPPOINT or CMD in the Dockerfile or as an override, make sure that you enter the absolute path. Also, do not use symlinks as the entry point to the container.

Creating images from AWS base images

To build a container image for a new Lambda function, you can start with an AWS base image for Lambda. Lambda provides two types of base images:

- Multi-architecture base image

Specify one of the main image tags (such as `python:3.9` or `java:11`) to choose this type of image.

- Architecture-specific base image

Specify an image tag with an architecture suffix. For example, specify `3.9-arm64` to choose the arm64 base image for Python 3.9.

You can also use an [alternative base image from another container registry \(p. 890\)](#). Lambda provides open-source runtime interface clients that you add to an alternative base image to make it compatible with Lambda.

Note

AWS periodically provides updates to the AWS base images for Lambda. If your Dockerfile includes the image name in the FROM property, your Docker client pulls the latest version of the image from the Amazon ECR repository. To use the updated base image, you must rebuild your container image and [update the function code \(p. 115\)](#).

To create an image from an AWS base image for Lambda

1. On your local machine, create a project directory for your new function.

2. Create a directory named **app** in the project directory, and then add your function handler code to the app directory.
3. Use a text editor to create a new Dockerfile.

The AWS base images provide the following environment variables:

- LAMBDA_TASK_ROOT=/var/task
- LAMBDA_RUNTIME_DIR=/var/runtime

Install any dependencies under the \${LAMBDA_TASK_ROOT} directory alongside the function handler to ensure that the Lambda runtime can locate them when the function is invoked.

The following shows an example Dockerfile for Node.js, Python, and Ruby:

Node.js 18

```
FROM public.ecr.aws/lambda/nodejs:18

# Assumes your function is named "app.js", and there is a package.json file in the
# app directory
COPY app.js package.json ${LAMBDA_TASK_ROOT}/

# Install NPM dependencies for function
RUN npm install

# Set the CMD to your handler (could also be done as a parameter override outside
# of the Dockerfile)
CMD [ "app.handler" ]
```

Python 3.8

```
FROM public.ecr.aws/lambda/python:3.8

# Install the function's dependencies using file requirements.txt
# from your project folder.

COPY requirements.txt .
RUN pip3 install -r requirements.txt --target "${LAMBDA_TASK_ROOT}"

# Copy function code
COPY app.py ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside
# of the Dockerfile)
CMD [ "app.handler" ]
```

Ruby 2.7

```
FROM public.ecr.aws/lambda/ruby:2.7

# Copy dependency management file
COPY Gemfile ${LAMBDA_TASK_ROOT}

# Install dependencies under LAMBDA_TASK_ROOT
ENV GEM_HOME=${LAMBDA_TASK_ROOT}
RUN bundle install

# Copy function code
COPY app.rb ${LAMBDA_TASK_ROOT}
```

```
# Set the CMD to your handler (could also be done as a parameter override outside
# of the Dockerfile)
CMD [ "app.LambdaFunction::Handler.process" ]
```

4. Build your Docker image with the `docker build` command. Enter a name for the image. The following example names the image `hello-world`.

```
docker build -t hello-world .
```

5. Start the Docker image with the `docker run` command. For this example, enter `hello-world` as the image name.

```
docker run -p 9000:8080 hello-world
```

6. (Optional) Test your application locally using the [runtime interface emulator \(p. 884\)](#). From a new terminal window, post an event to the following endpoint using a curl command:

```
curl -XPOST "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function running in the container image and returns a response.

Creating images from alternative base images

Prerequisites

- The AWS CLI
- Docker Desktop
- Your function code

To create an image using an alternative base image

1. Choose a base image. Lambda supports all Linux distributions, such as Alpine, Debian, and Ubuntu.
2. On your local machine, create a project directory for your new function.
3. Create a directory named `app` in the project directory, and then add your function handler code to the `app` directory.
4. Use a text editor to create a new Dockerfile with the following configuration:
 - Set the `FROM` property to the URI of the base image.
 - Add instructions to install the runtime interface client.
 - Set the `ENTRYPOINT` property to invoke the runtime interface client.
 - Set the `CMD` argument to specify the Lambda function handler.

The following example shows a Dockerfile for Python:

```
# Define function directory
ARG FUNCTION_DIR="/function"

FROM python:buster as build-image

# Install aws-lambda-cpp build dependencies
RUN apt-get update && \
    apt-get install -y \
    g++ \
```

```
make \
cmake \
unzip \
libcurl4-openssl-dev

# Include global arg in this stage of the build
ARG FUNCTION_DIR
# Create function directory
RUN mkdir -p ${FUNCTION_DIR}

# Copy function code
COPY app/* ${FUNCTION_DIR}

# Install the runtime interface client
RUN pip install \
    --target ${FUNCTION_DIR} \
    awslambdaric

# Multi-stage build: grab a fresh copy of the base image
FROM python:buster

# 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 build image dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

ENTRYPOINT [ "/usr/local/bin/python", "-m", "awslambdaric" ]
CMD [ "app.handler" ]
```

5. Build your Docker image with the `docker build` command. Enter a name for the image. The following example names the image `hello-world`.

```
docker build -t hello-world .
```

6. (Optional) Test your application locally using the [Runtime interface emulator \(p. 884\)](#).

Upload the image to the Amazon ECR repository

To upload the image to Amazon ECR and create the Lambda function

1. Run the [get-login-password](#) command to authenticate the Docker CLI to your Amazon ECR registry.
 - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
 - Replace `111122223333` with your AWS account 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. Create a repository in Amazon ECR using the [create-repository](#) command.

```
aws ecr create-repository --repository-name hello-world --image-scanning-configuration
scanOnPush=true --image-tag-mutability MUTABLE
```

If successful, you see a response like this:

```
{  
    "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. Copy the `repositoryUri` from the output in the previous step.
4. Run the [docker tag](#) command to tag your local image into your Amazon ECR repository as the latest version. In this command:
 - Replace `docker-image:test` with the name and [tag](#) of your Docker image.
 - Replace the Amazon ECR repository URI with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the [docker push](#) command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.
6. [Create an execution role \(p. 191\)](#) for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.
7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
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
```

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{  
    "ExecutedVersion": "$LATEST",  
    "StatusCode": 200  
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must create a new image version and store the image in the Amazon ECR repository. For more information, see [Updating function code \(p. 115\)](#).

Create an image using the AWS SAM toolkit

You can use the AWS Serverless Application Model (AWS SAM) toolkit to create and deploy a function defined as a container image. For a new project, you can use the AWS SAM CLI `init` command to set up the scaffolding for your project in your preferred runtime.

In your AWS SAM template, you set the Runtime type to Image and provide the URI of the base image.

For more information, see [Building applications](#) in the *AWS Serverless Application Model Developer Guide*.

Lambda extensions

You can use Lambda extensions to augment your Lambda functions. For example, use Lambda extensions to integrate functions with your preferred monitoring, observability, security, and governance tools. You can choose from a broad set of tools that [AWS Lambda Partners](#) provides, or you can [create your own Lambda extensions \(p. 900\)](#).

Lambda supports external and internal extensions. An external extension runs as an independent process in the execution environment and continues to run after the function invocation is fully processed. Because extensions run as separate processes, you can write them in a different language than the function. All [Lambda runtimes \(p. 37\)](#) support extensions.

An internal extension runs as part of the runtime process. Your function accesses internal extensions by using wrapper scripts or in-process mechanisms such as JAVA_TOOL_OPTIONS. For more information, see [Modifying the runtime environment \(p. 49\)](#).

You can add extensions to a function using the Lambda console, the AWS Command Line Interface (AWS CLI), or infrastructure as code (IaC) services and tools such as AWS CloudFormation, AWS Serverless Application Model (AWS SAM), and Terraform.

You are charged for the execution time that the extension consumes (in 1 ms increments). For more pricing information for extensions, see [AWS Lambda Pricing](#). For pricing information for partner extensions, see those partners' websites. There is no cost to install your own extensions.

Topics

- [Execution environment \(p. 894\)](#)
- [Impact on performance and resources \(p. 895\)](#)
- [Permissions \(p. 895\)](#)
- [Configuring Lambda extensions \(p. 896\)](#)
- [AWS Lambda extensions partners \(p. 898\)](#)
- [Lambda Extensions API \(p. 900\)](#)
- [Lambda Telemetry API \(p. 911\)](#)

Execution environment

Lambda invokes your function in an [execution environment \(p. 14\)](#), which provides a secure and isolated runtime environment. The execution environment manages the resources required to run your function and provides lifecycle support for the function's runtime and extensions.

The lifecycle of the execution environment includes the following phases:

- **Init:** In this phase, Lambda creates or unfreezes an execution environment with the configured resources, downloads the code for the function and all layers, initializes any extensions, initializes the runtime, and then runs the function's initialization code (the code outside the main handler). The Init phase happens either during the first invocation, or in advance of function invocations if you have enabled [provisioned concurrency \(p. 213\)](#).

The Init phase is split into three sub-phases: Extension init, Runtime init, and Function init. These sub-phases ensure that all extensions and the runtime complete their setup tasks before the function code runs.

When [Lambda SnapStart \(p. 992\)](#) is activated, the Init phase happens when you publish a function version. Lambda saves a snapshot of the memory and disk state of the initialized execution environment, persists the encrypted snapshot, and caches it for low-latency access. If you have a beforeCheckpoint [runtime hook \(p. 1001\)](#), then the code runs at the end of Init phase.

- **Restore** (SnapStart only): When you first invoke a [SnapStart \(p. 992\)](#) function and as the function scales up, Lambda resumes new execution environments from the persisted snapshot instead of initializing the function from scratch. If you have an afterRestore() [runtime hook \(p. 1001\)](#), the code runs at the end of the Restore phase. You are charged for the duration of afterRestore() runtime hooks. The runtime (JVM) must load and afterRestore() runtime hooks must complete within the timeout limit (10 seconds). Otherwise, you'll get a SnapStartTimeoutException. When the Restore phase completes, Lambda invokes the function handler (the Invoke phase).
- **Invoke**: In this phase, Lambda invokes the function handler. After the function runs to completion, Lambda prepares to handle another function invocation.
- **Shutdown**: This phase is triggered if the Lambda function does not receive any invocations for a period of time. In the Shutdown phase, Lambda shuts down the runtime, alerts the extensions to let them stop cleanly, and then removes the environment. Lambda sends a Shutdown event to each extension, which tells the extension that the environment is about to be shut down.

During the Init phase, Lambda extracts layers containing extensions into the /opt directory in the execution environment. Lambda looks for extensions in the /opt/extensions/ directory, interprets each file as an executable bootstrap for launching the extension, and starts all extensions in parallel.

Impact on performance and resources

The size of your function's extensions counts towards the deployment package size limit. For a .zip file archive, the total unzipped size of the function and all extensions cannot exceed the unzipped deployment package size limit of 250 MB.

Extensions can impact the performance of your function because they share function resources such as CPU, memory, and storage. For example, if an extension performs compute-intensive operations, you may see your function's execution duration increase.

Each extension must complete its initialization before Lambda invokes the function. Therefore, an extension that consumes significant initialization time can increase the latency of the function invocation.

To measure the extra time that the extension takes after the function execution, you can use the PostRuntimeExtensionsDuration [function metric \(p. 870\)](#). To measure the increase in memory used, you can use the MaxMemoryUsed metric. To understand the impact of a specific extension, you can run different versions of your functions side by side.

Permissions

Extensions have access to the same resources as functions. Because extensions are executed within the same environment as the function, permissions are shared between the function and the extension.

For a .zip file archive, you can create an AWS CloudFormation template to simplify the task of attaching the same extension configuration—including AWS Identity and Access Management (IAM) permissions—to multiple functions.

Configuring Lambda extensions

Configuring extensions (.zip file archive)

You can add an extension to your function as a [Lambda layer \(p. 93\)](#). Using layers enables you to share extensions across your organization or to the entire community of Lambda developers. You can add one or more extensions to a layer. You can register up to 10 extensions for a function.

You add the extension to your function using the same method as you would for any layer. For more information, see [Using layers with your Lambda function \(p. 245\)](#).

Add an extension to your function (console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose a function.
3. Choose the **Code** tab if it is not already selected.
4. Under **Layers**, choose **Edit**.
5. For **Choose a layer**, choose **Specify an ARN**.
6. For **Specify an ARN**, enter the Amazon Resource Name (ARN) of an extension layer.
7. Choose **Add**.

Using extensions in container images

You can add extensions to your [container image \(p. 881\)](#). The ENTRYPPOINT container image setting specifies the main process for the function. Configure the ENTRYPPOINT setting in the Dockerfile, or as an override in the function configuration.

You can run multiple processes within a container. Lambda manages the lifecycle of the main process and any additional processes. Lambda uses the [Extensions API \(p. 900\)](#) to manage the extension lifecycle.

Example: Adding an external extension

An external extension runs in a separate process from the Lambda function. Lambda starts a process for each extension in the /opt/extensions/ directory. Lambda uses the Extensions API to manage the extension lifecycle. After the function has run to completion, Lambda sends a Shutdown event to each external extension.

Example of adding an external extension to a Python base image

```
FROM public.ecr.aws/lambda/python:3.8

# 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
ADD my-extension.zip /opt
CMD python ./my-function.py
```

Next steps

To learn more about extensions, we recommend the following resources:

- For a basic working example, see [Building Extensions for AWS Lambda](#) on the AWS Compute Blog.
- For information about extensions that AWS Lambda Partners provides, see [Introducing AWS Lambda Extensions](#) on the AWS Compute Blog.
- To view available example extensions and wrapper scripts, see [AWS Lambda Extensions](#) on the AWS Samples GitHub repository.

AWS Lambda extensions partners

AWS Lambda has partnered with several third party entities to provide extensions to integrate with your Lambda functions. The following list details third party extensions that are ready for you to use at any time.

- [AppDynamics](#) – Provides automatic instrumentation of Node.js or Python Lambda functions, providing visibility and alerting on function performance.
- [Check Point CloudGuard](#) – An extension-based runtime solution that offers full lifecycle security for serverless applications.
- [Datadog](#) – Provides comprehensive, real-time visibility to your serverless applications through the use of metrics, traces, and logs.
- [Dynatrace](#) – Provides visibility into traces and metrics, and leverages AI for automated error detection and root cause analysis across the entire application stack.
- [Elastic](#) – Provides Application Performance Monitoring (APM) to identify and resolve root cause issues using correlated traces, metrics, and logs.
- [Epsagon](#) – Listens to invocation events, stores traces, and sends them in parallel to Lambda function executions.
- [Fastly](#) – Protects your Lambda functions from suspicious activity, such as injection-style attacks, account takeover via credential stuffing, malicious bots, and API abuse.
- [HashiCorp Vault](#) – Manages secrets and makes them available for developers to use within function code, without making functions Vault aware.
- [Honeycomb](#) – Observability tool for debugging your app stack.
- [Lumigo](#) – Profiles Lambda function invocations and collects metrics for troubleshooting issues in serverless and microservice environments.
- [New Relic](#) – Runs alongside Lambda functions, automatically collecting, enhancing, and transporting telemetry to New Relic's unified observability platform.
- [Sedai](#) – An autonomous cloud management platform, powered by AI/ML, that delivers continuous optimization for cloud operations teams to maximize cloud cost savings, performance, and availability at scale.
- [Sentry](#) – Diagnose, fix, and optimize performance of Lambda functions.
- [Site24x7](#) – Achieve real-time observability into your Lambda environments
- [Splunk](#) – Collects high-resolution, low-latency metrics for efficient and effective monitoring of Lambda functions.
- [Sumo Logic](#) – Provides visibility into the health and performance of serverless applications.
- [Thundra](#) – Provides asynchronous telemetry reporting, such as traces, metrics, and logs.

AWS managed extensions

AWS provides its own managed extensions, including:

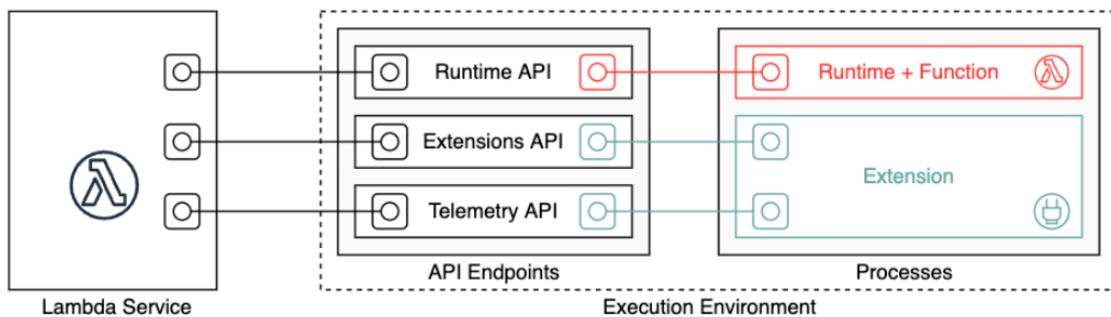
- [AWS AppConfig](#) – Use feature flags and dynamic data to update your Lambda functions. You can also use this extension to update other dynamic configuration, such as ops throttling and tuning.
- [Amazon CodeGuru Profiler](#) – Improves application performance and reduces cost by pinpointing an application's most expensive line of code and providing recommendations for improving code.
- [CloudWatch Lambda Insights](#) – Monitor, troubleshoot, and optimize the performance of your Lambda functions through automated dashboards.
- [AWS Distro for OpenTelemetry \(ADOT\)](#) – Enables functions to send trace data to AWS monitoring services such as AWS X-Ray, and to destinations that support OpenTelemetry such as Honeycomb and Lightstep.

- **AWS Parameters and Secrets** – Enables customers to securely retrieve parameters from [AWS Systems Manager Parameter Store](#) and secrets from [AWS Secrets Manager](#).

For additional extensions samples and demo projects, see [AWS Lambda Extensions](#).

Lambda Extensions API

Lambda function authors use extensions to integrate Lambda with their preferred tools for monitoring, observability, security, and governance. Function authors can use extensions from AWS, [AWS Partners \(p. 898\)](#), and open-source projects. For more information on using extensions, see [Introducing AWS Lambda Extensions](#) on the AWS Compute Blog. This section describes how to use the Lambda Extensions API, the Lambda execution environment lifecycle, and the Lambda Extensions API reference.



As an extension author, you can use the Lambda Extensions API to integrate deeply into the Lambda [execution environment \(p. 14\)](#). Your extension can register for function and execution environment lifecycle events. In response to these events, you can start new processes, run logic, and control and participate in all phases of the Lambda lifecycle: initialization, invocation, and shutdown. In addition, you can use the [Runtime Logs API \(p. 939\)](#) to receive a stream of logs.

An extension runs as an independent process in the execution environment and can continue to run after the function invocation is fully processed. Because extensions run as processes, you can write them in a different language than the function. We recommend that you implement extensions using a compiled language. In this case, the extension is a self-contained binary that is compatible with supported runtimes. All [Lambda runtimes \(p. 37\)](#) support extensions. If you use a non-compiled language, ensure that you include a compatible runtime in the extension.

Lambda also supports *internal extensions*. An internal extension runs as a separate thread in the runtime process. The runtime starts and stops the internal extension. An alternative way to integrate with the Lambda environment is to use language-specific [environment variables and wrapper scripts \(p. 49\)](#). You can use these to configure the runtime environment and modify the startup behavior of the runtime process.

You can add extensions to a function in two ways. For a function deployed as a [zip file archive \(p. 18\)](#), you deploy your extension as a [layer \(p. 93\)](#). For a function defined as a container image, you add [the extensions \(p. 896\)](#) to your container image.

Note

For example extensions and wrapper scripts, see [AWS Lambda Extensions](#) on the AWS Samples GitHub repository.

Topics

- [Lambda execution environment lifecycle \(p. 900\)](#)
- [Extensions API reference \(p. 906\)](#)

Lambda execution environment lifecycle

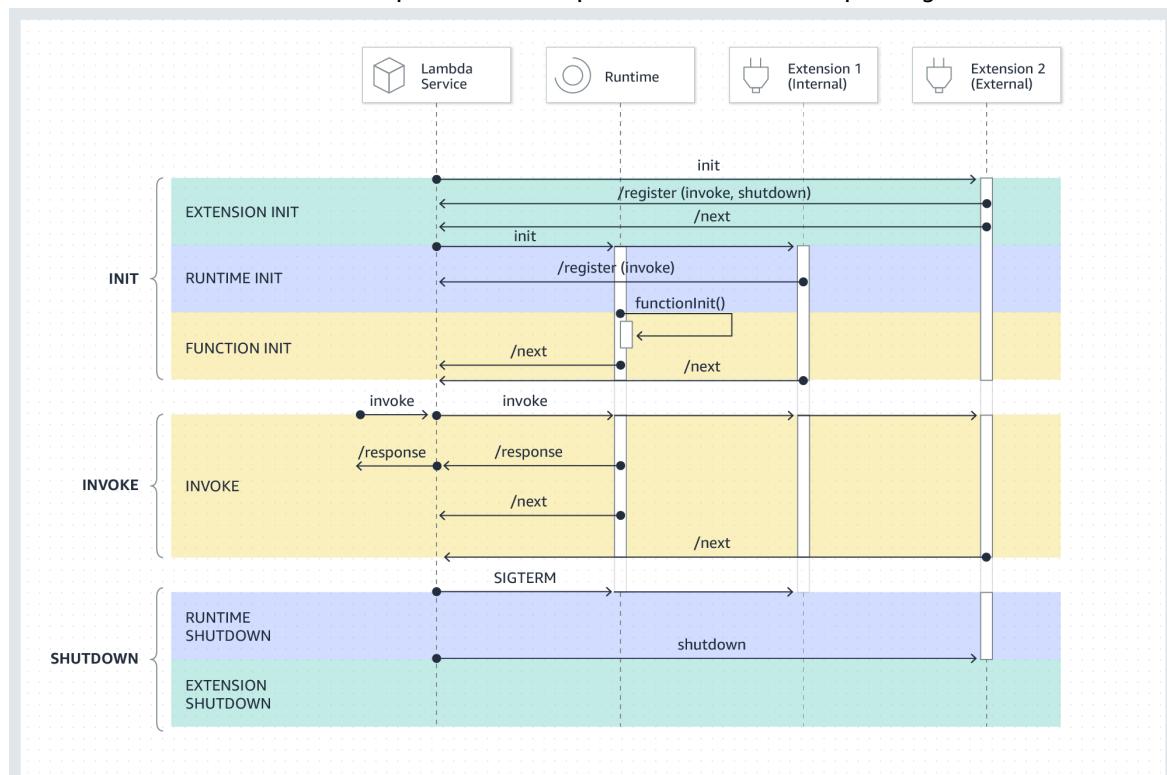
The lifecycle of the execution environment includes the following phases:

- **Init:** In this phase, Lambda creates or unfreezes an execution environment with the configured resources, downloads the code for the function and all layers, initializes any extensions, initializes the runtime, and then runs the function's initialization code (the code outside the main handler). The Init phase happens either during the first invocation, or in advance of function invocations if you have enabled [provisioned concurrency \(p. 213\)](#).

The Init phase is split into three sub-phases: Extension init, Runtime init, and Function init. These sub-phases ensure that all extensions and the runtime complete their setup tasks before the function code runs.

- **Invoke:** In this phase, Lambda invokes the function handler. After the function runs to completion, Lambda prepares to handle another function invocation.
- **Shutdown:** This phase is triggered if the Lambda function does not receive any invocations for a period of time. In the Shutdown phase, Lambda shuts down the runtime, alerts the extensions to let them stop cleanly, and then removes the environment. Lambda sends a Shutdown event to each extension, which tells the extension that the environment is about to be shut down.

Each phase starts with an event from Lambda to the runtime and to all registered extensions. The runtime and each extension signal completion by sending a Next API request. Lambda freezes the execution environment when each process has completed and there are no pending events.



Topics

- [Init phase \(p. 902\)](#)
- [Invoke phase \(p. 15\)](#)
- [Shutdown phase \(p. 16\)](#)
- [Permissions and configuration \(p. 905\)](#)
- [Failure handling \(p. 906\)](#)
- [Troubleshooting extensions \(p. 906\)](#)

Init phase

During the Extension `init` phase, each extension needs to register with Lambda to receive events. Lambda uses the full file name of the extension to validate that the extension has completed the bootstrap sequence. Therefore, each `Register` API call must include the `Lambda-Extension-Name` header with the full file name of the extension.

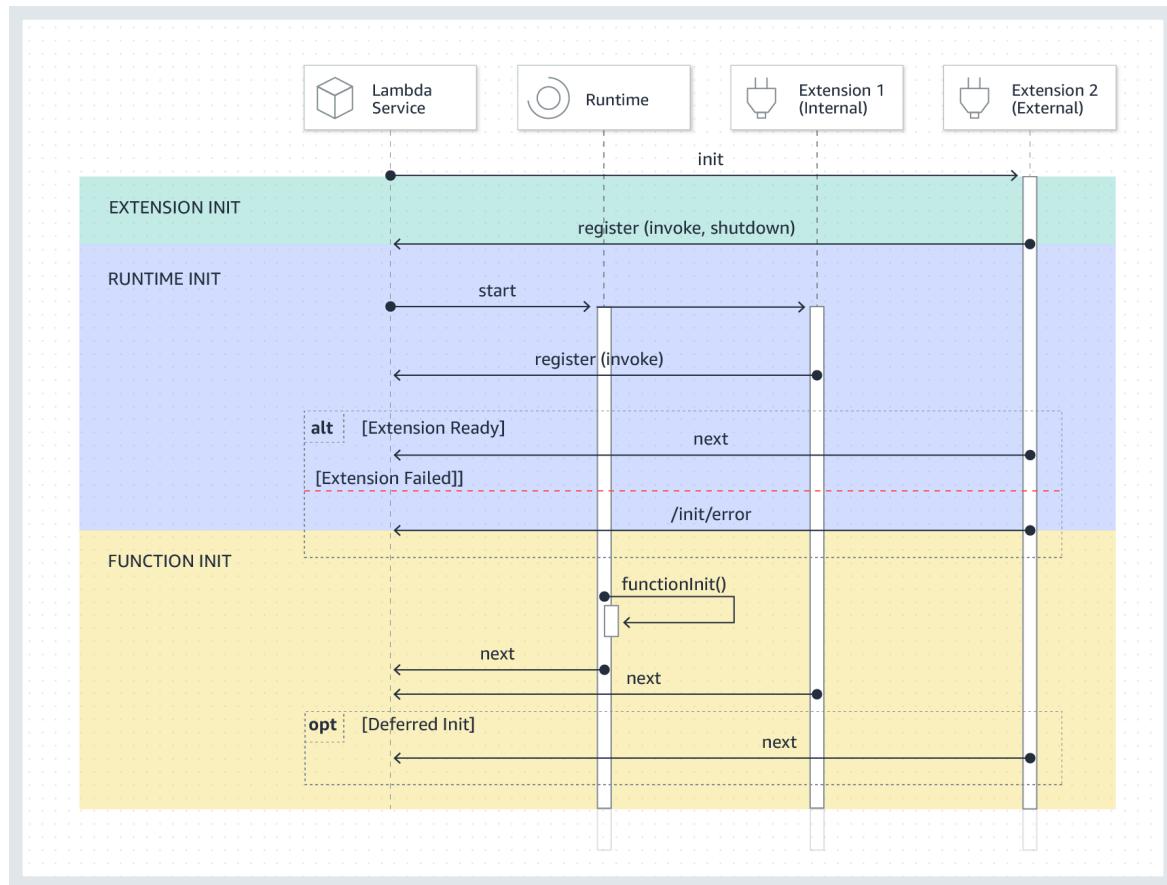
You can register up to 10 extensions for a function. This limit is enforced through the `Register` API call.

After each extension registers, Lambda starts the Runtime `init` phase. The runtime process calls `functionInit` to start the Function `init` phase.

The Init phase completes after the runtime and each registered extension indicate completion by sending a Next API request.

Note

Extensions can complete their initialization at any point in the Init phase.



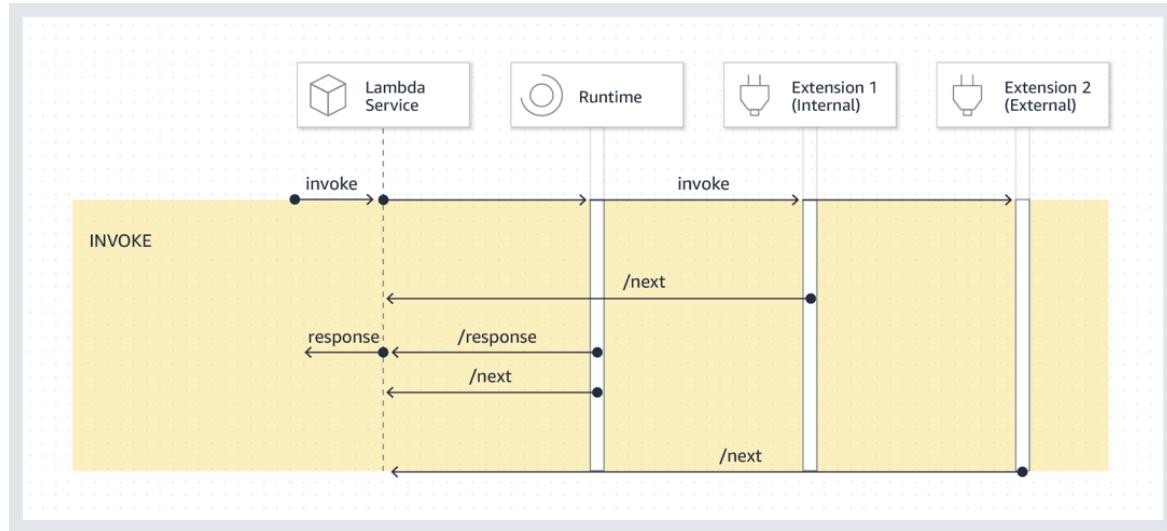
Invoke phase

When a Lambda function is invoked in response to a Next API request, Lambda sends an `Invoke` event to the runtime and to each extension that is registered for the `Invoke` event.

During the invocation, external extensions run in parallel with the function. They also continue running after the function has completed. This enables you to capture diagnostic information or to send logs, metrics, and traces to a location of your choice.

After receiving the function response from the runtime, Lambda returns the response to the client, even if extensions are still running.

The Invoke phase ends after the runtime and all extensions signal that they are done by sending a Next API request.



Event payload: The event sent to the runtime (and the Lambda function) carries the entire request, headers (such as RequestId), and payload. The event sent to each extension contains metadata that describes the event content. This lifecycle event includes the type of the event, the time that the function times out (deadlineMs), the requestId, the invoked function's Amazon Resource Name (ARN), and tracing headers.

Extensions that want to access the function event body can use an in-runtime SDK that communicates with the extension. Function developers use the in-runtime SDK to send the payload to the extension when the function is invoked.

Here is an example payload:

```
{
  "eventType": "INVOCATE",
  "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"
  }
}
```

Duration limit: The function's timeout setting limits the duration of the entire Invoke phase. For example, if you set the function timeout as 360 seconds, the function and all extensions need to complete within 360 seconds. Note that there is no independent post-invoke phase. The duration is the sum of all invocation time (runtime + extensions) and is not calculated until the function and all extensions have finished running.

Performance impact and extension overhead: Extensions can impact function performance. As an extension author, you have control over the performance impact of your extension. For example, if your extension performs compute-intensive operations, the function's duration increases because the extension and the function code share the same CPU resources. In addition, if your extension performs

extensive operations after the function invocation completes, the function duration increases because the Invoke phase continues until all extensions signal that they are completed.

Note

Lambda allocates CPU power in proportion to the function's memory setting. You might see increased execution and initialization duration at lower memory settings because the function and extension processes are competing for the same CPU resources. To reduce the execution and initialization duration, try increasing the memory setting.

To help identify the performance impact introduced by extensions on the Invoke phase, Lambda outputs the PostRuntimeExtensionsDuration metric. This metric measures the cumulative time spent between the runtime Next API request and the last extension Next API request. To measure the increase in memory used, use the MaxMemoryUsed metric. For more information about function metrics, see [Working with Lambda function metrics \(p. 870\)](#).

Function developers can run different versions of their functions side by side to understand the impact of a specific extension. We recommend that extension authors publish expected resource consumption to make it easier for function developers to choose a suitable extension.

Shutdown phase

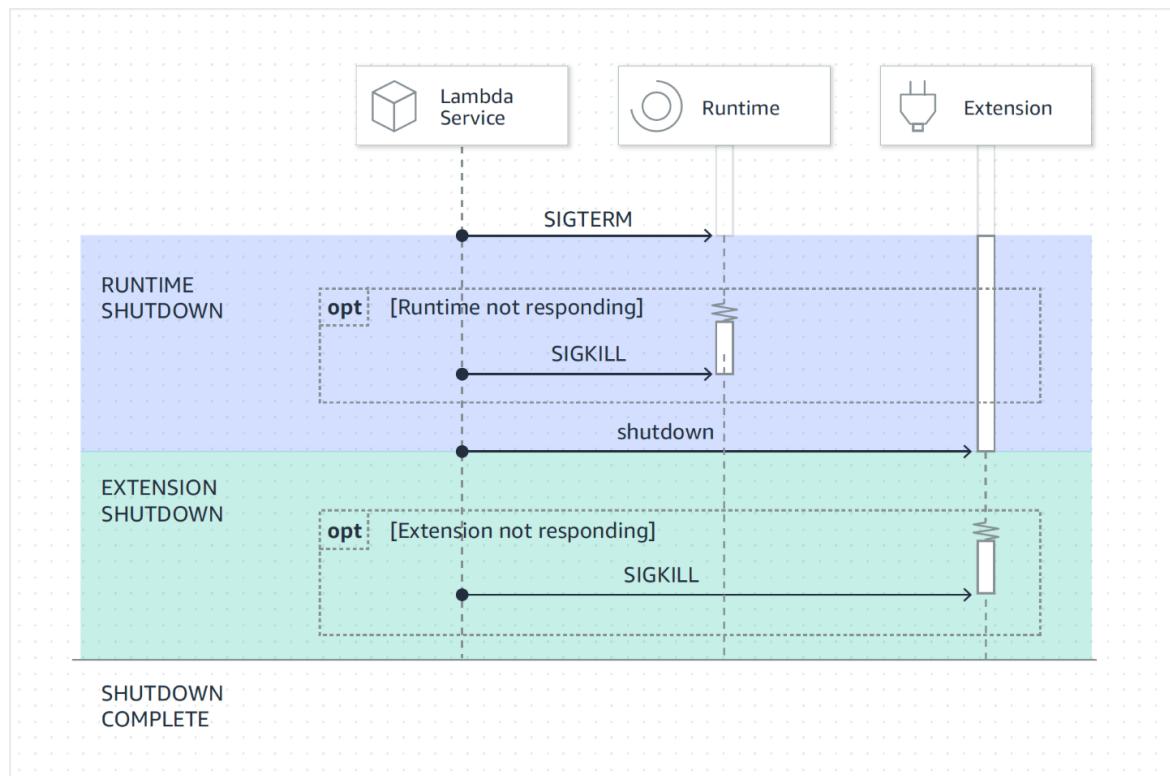
When Lambda is about to shut down the runtime, it sends a Shutdown to each registered external extension. Extensions can use this time for final cleanup tasks. The Shutdown event is sent in response to a Next API request.

Duration limit: The maximum duration of the Shutdown phase depends on the configuration of registered extensions:

- 0 ms – A function with no registered extensions
- 500 ms – A function with a registered internal extension
- 2,000 ms – A function with one or more registered external extensions

For a function with external extensions, Lambda reserves up to 300 ms (500 ms for a runtime with an internal extension) for the runtime process to perform a graceful shutdown. Lambda allocates the remainder of the 2,000 ms limit for external extensions to shut down.

If the runtime or an extension does not respond to the Shutdown event within the limit, Lambda ends the process using a SIGKILL signal.



Event payload: The Shutdown event contains the reason for the shutdown and the time remaining in milliseconds.

The shutdownReason includes the following values:

- SPINDOWN – Normal shutdown
- TIMEOUT – Duration limit timed out
- FAILURE – Error condition, such as an out-of-memory event

```
{
  "eventType": "SHUTDOWN",
  "shutdownReason": "reason for shutdown",
  "deadlineMs": "the time and date that the function times out in Unix time milliseconds"
}
```

Permissions and configuration

Extensions run in the same execution environment as the Lambda function. Extensions also share resources with the function, such as CPU, memory, and /tmp disk storage. In addition, extensions use the same AWS Identity and Access Management (IAM) role and security context as the function.

File system and network access permissions: Extensions run in the same file system and network name namespace as the function runtime. This means that extensions need to be compatible with the associated operating system. If an extension requires any additional outbound network traffic rules, you must apply these rules to the function configuration.

Note

Because the function code directory is read-only, extensions cannot modify the function code.

Environment variables: Extensions can access the function's [environment variables \(p. 76\)](#), except for the following variables that are specific to the runtime process:

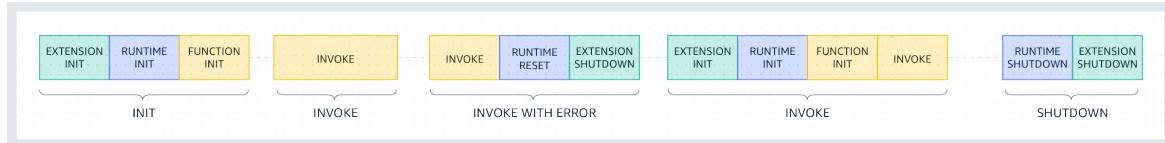
- AWS_EXECUTION_ENV
- AWS_LAMBDA_LOG_GROUP_NAME
- AWS_LAMBDA_LOG_STREAM_NAME
- AWS_XRAY_CONTEXT_MISSING
- AWS_XRAY_DAEMON_ADDRESS
- LAMBDA_RUNTIME_DIR
- LAMBDA_TASK_ROOT
- _AWS_XRAY_DAEMON_ADDRESS
- _AWS_XRAY_DAEMON_PORT
- _HANDLER

Failure handling

Initialization failures: If an extension fails, Lambda restarts the execution environment to enforce consistent behavior and to encourage fail fast for extensions. Also, for some customers, the extensions must meet mission-critical needs such as logging, security, governance, and telemetry collection.

Invoke failures (such as out of memory, function timeout): Because extensions share resources with the runtime, memory exhaustion affects them. When the runtime fails, all extensions and the runtime itself participate in the Shutdown phase. In addition, the runtime is restarted—either automatically as part of the current invocation, or via a deferred re-initialization mechanism.

If there is a failure (such as a function timeout or runtime error) during Invoke, the Lambda service performs a reset. The reset behaves like a Shutdown event. First, Lambda shuts down the runtime, then it sends a Shutdown event to each registered external extension. The event includes the reason for the shutdown. If this environment is used for a new invocation, the extension and runtime are re-initialized as part of the next invocation.



For a more detailed explanation of the previous diagram, see [Failures during the invoke phase \(p. 16\)](#).

Extension logs: Lambda sends the log output of extensions to CloudWatch Logs. Lambda also generates an additional log event for each extension during Init. The log event records the name and registration preference (event, config) on success, or the failure reason on failure.

Troubleshooting extensions

- If a Register request fails, make sure that the Lambda-Extension-Name header in the Register API call contains the full file name of the extension.
- If the Register request fails for an internal extension, make sure that the request does not register for the Shutdown event.

Extensions API reference

The OpenAPI specification for the extensions API version **2020-01-01** is available here: [extensions-api.zip](#)

You can retrieve the value of the API endpoint from the AWS_LAMBDA_RUNTIME_API environment variable. To send a Register request, use the prefix 2020-01-01/ before each API path. For example:

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
```

API methods

- [Register \(p. 907\)](#)
- [Next \(p. 908\)](#)
- [Init error \(p. 909\)](#)
- [Exit error \(p. 910\)](#)

Register

During Extension init, all extensions need to register with Lambda to receive events. Lambda uses the full file name of the extension to validate that the extension has completed the bootstrap sequence. Therefore, each Register API call must include the Lambda-Extension-Name header with the full file name of the extension.

Internal extensions are started and stopped by the runtime process, so they are not permitted to register for the Shutdown event.

Path – /extension/register

Method – POST

Request headers

- Lambda-Extension-Name – The full file name of the extension. Required: yes. Type: string.
- Lambda-Extension-Accept-Feature – Use this to specify optional Extensions features during registration. Required: no. Type: comma separated string. Features available to specify using this setting:
 - accountId – If specified, the Extension registration response will contain the account ID associated with the Lambda function that you're registering the Extension for.

Request body parameters

- events – Array of the events to register for. Required: no. Type: array of strings. Valid strings: INVOKE, SHUTDOWN.

Response headers

- Lambda-Extension-Identifier – Generated unique agent identifier (UUID string) that is required for all subsequent requests.

Response codes

- 200 – Response body contains the function name, function version, and handler name.
- 400 – Bad Request
- 403 – Forbidden
- 500 – Container error. Non-recoverable state. Extension should exit promptly.

Example Example request body

```
{  
    'events': [ 'INVOKE', 'SHUTDOWN' ]  
}
```

Example Example response body

```
{  
    "functionName": "helloWorld",  
    "functionVersion": "$LATEST",  
    "handler": "lambda_function.lambda_handler"  
}
```

Example Example response body with optional accountId feature

```
{  
    "functionName": "helloWorld",  
    "functionVersion": "$LATEST",  
    "handler": "lambda_function.lambda_handler",  
    "accountId": "123456789012"  
}
```

Next

Extensions send a Next API request to receive the next event, which can be an Invoke event or a Shutdown event. The response body contains the payload, which is a JSON document that contains event data.

The extension sends a Next API request to signal that it is ready to receive new events. This is a blocking call.

Do not set a timeout on the GET call, as the extension can be suspended for a period of time until there is an event to return.

Path – /extension/event/next

Method – GET

Request headers

- **Lambda-Extension-Identifier** – Unique identifier for extension (UUID string). Required: yes.
Type: UUID string.

Response headers

- **Lambda-Extension-Event-Identifier** – Unique identifier for the event (UUID string).

Response codes

- 200 – Response contains information about the next event (EventInvoke or EventShutdown).
- 403 – Forbidden
- 500 – Container error. Non-recoverable state. Extension should exit promptly.

Init error

The extension uses this method to report an initialization error to Lambda. Call it when the extension fails to initialize after it has registered. After Lambda receives the error, no further API calls succeed. The extension should exit after it receives the response from Lambda.

Path – /extension/init/error

Method – POST

Request headers

- **Lambda-Extension-Identifier** – Unique identifier for extension. Required: yes. Type: UUID string.
- **Lambda-Extension-Function-Error-Type** – Error type that the extension encountered. Required: yes. This header consists of a string value. Lambda accepts any string, but we recommend a format of <category.reason>. For example:
 - Extension.NoSuchHandler
 - Extension.APIKeyNotFound
 - Extension.ConfigInvalid
 - Extension.UnknownReason

Request body parameters

- **ErrorRequest** – Information about the error. Required: no.

This field is a JSON object with the following structure:

```
{  
    errorMessage: string (text description of the error),  
    errorType: string,  
    stackTrace: array of strings  
}
```

Note that Lambda accepts any value for `errorType`.

The following example shows a Lambda function error message in which the function could not parse the event data provided in the invocation.

Example Function error

```
{  
    "errorMessage" : "Error parsing event data.",  
    "errorType" : "InvalidEventDataException",  
    "stackTrace": [ ]  
}
```

Response codes

- 202 – Accepted
- 400 – Bad Request
- 403 – Forbidden
- 500 – Container error. Non-recoverable state. Extension should exit promptly.

Exit error

The extension uses this method to report an error to Lambda before exiting. Call it when you encounter an unexpected failure. After Lambda receives the error, no further API calls succeed. The extension should exit after it receives the response from Lambda.

Path – /extension/exit/error

Method – POST

Request headers

- **Lambda-Extension-Identifier** – Unique identifier for extension. Required: yes. Type: UUID string.
- **Lambda-Extension-Function-Error-Type** – Error type that the extension encountered. Required: yes. This header consists of a string value. Lambda accepts any string, but we recommend a format of <category.reason>. For example:
 - Extension.NoSuchHandler
 - Extension.APIKeyNotFound
 - Extension.ConfigInvalid
 - Extension.UnknownReason

Request body parameters

- **ErrorRequest** – Information about the error. Required: no.

This field is a JSON object with the following structure:

```
{  
    errorMessage: string (text description of the error),  
    errorType: string,  
    stackTrace: array of strings  
}
```

Note that Lambda accepts any value for **errorType**.

The following example shows a Lambda function error message in which the function could not parse the event data provided in the invocation.

Example Function error

```
{  
    "errorMessage" : "Error parsing event data.",  
    "errorType" : "InvalidEventDataException",  
    "stackTrace": [ ]  
}
```

Response codes

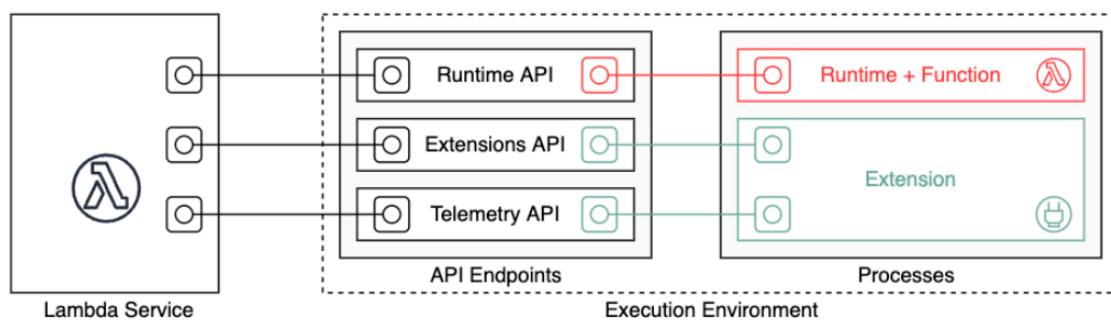
- 202 – Accepted
- 400 – Bad Request
- 403 – Forbidden
- 500 – Container error. Non-recoverable state. Extension should exit promptly.

Lambda Telemetry API

Using the Lambda Telemetry API, your extensions can directly receive telemetry data from Lambda. During function initialization and invocation, Lambda automatically captures telemetry, such as logs, platform metrics, and platform traces. With Telemetry API, extensions can get this telemetry data directly from Lambda in near real time.

You can subscribe your Lambda extensions to telemetry streams directly from within the Lambda execution environment. After subscribing, Lambda automatically streams all telemetry data to your extensions. You can then process, filter, and deliver that data to your preferred destination, such as an Amazon Simple Storage Service (Amazon S3) bucket or a third-party observability tools provider.

The following diagram shows how the Extensions API and Telemetry API connect extensions to Lambda from within the execution environment. The Runtime API also connects your runtime and function to Lambda.



Important

The Lambda Telemetry API supersedes the Lambda Logs API. **While the Logs API remains fully functional, we recommend using only the Telemetry API going forward.** You can subscribe your extension to a telemetry stream using either the Telemetry API or the Logs API. After subscribing using one of these APIs, any attempt to subscribe using the other API returns an error.

Extensions can use the Telemetry API to subscribe to three different telemetry streams:

- **Platform telemetry** – Logs, metrics, and traces, which describe events and errors related to the execution environment runtime lifecycle, extension lifecycle, and function invocations.
- **Function logs** – Custom logs that the Lambda function code generates.
- **Extension logs** – Custom logs that the Lambda extension code generates.

Note

Lambda sends logs and metrics to CloudWatch, and traces to X-Ray (if you've activated tracing), even if an extension subscribes to telemetry streams.

Sections

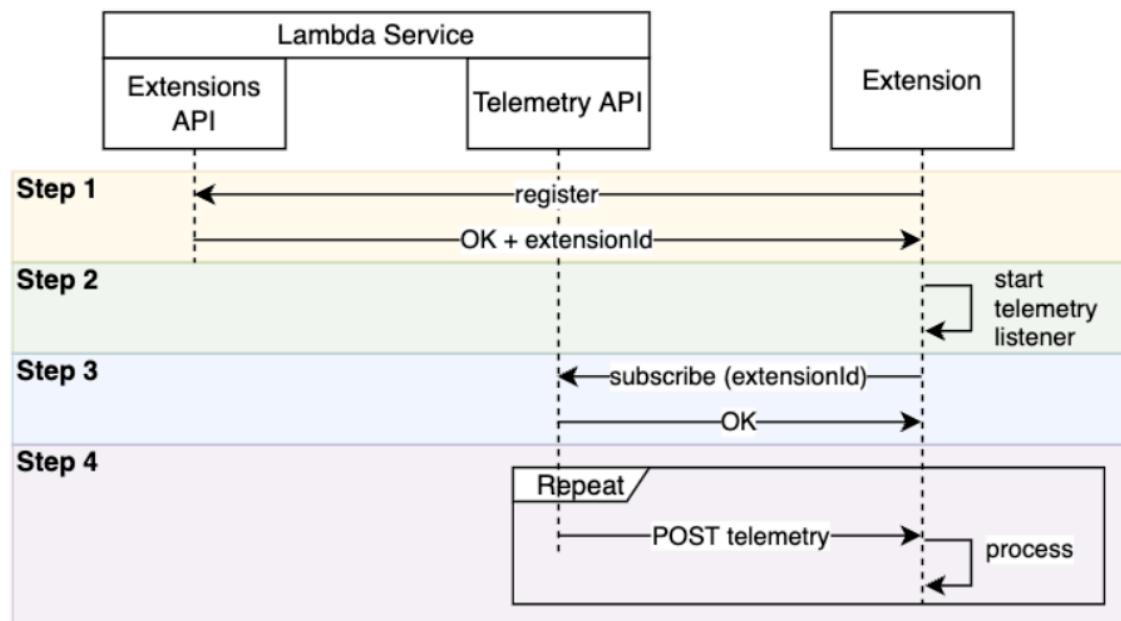
- [Creating extensions using the Telemetry API \(p. 912\)](#)
- [Registering your extension \(p. 913\)](#)
- [Creating a telemetry listener \(p. 913\)](#)
- [Specifying a destination protocol \(p. 914\)](#)
- [Configuring memory usage and buffering \(p. 915\)](#)
- [Sending a subscription request to the Telemetry API \(p. 916\)](#)
- [Inbound Telemetry API messages \(p. 916\)](#)

- [Lambda Telemetry API reference \(p. 919\)](#)
- [Lambda Telemetry API Event schema reference \(p. 922\)](#)
- [Converting Lambda Telemetry API Event objects to OpenTelemetry Spans \(p. 935\)](#)
- [Lambda Logs API \(p. 939\)](#)

Creating extensions using the Telemetry API

Lambda extensions run as independent processes in the execution environment, and can continue to run after the function invocation completes. Because extensions are separate processes, you can write them in a language different from the function code. We recommend implementing extensions using a compiled language such as Golang or Rust. This way, the extension is a self-contained binary that can be compatible with any supported runtime.

The following diagram illustrates a four-step process to create an extension that receives and processes telemetry data using the Telemetry API.



Here is each step in more detail:

1. Register your extension using the [the section called “Extensions API” \(p. 900\)](#). This provides you with a Lambda-Extension-Identifier, which you'll need in the following steps. For more information about how to register your extension, see [the section called “Registering your extension” \(p. 913\)](#).
2. Create a telemetry listener. This can be a basic HTTP or TCP server. Lambda uses the URI of the telemetry listener to send telemetry data to your extension. For more information, see [the section called “Creating a telemetry listener” \(p. 913\)](#).
3. Using the Subscribe API in the Telemetry API, subscribe your extension to your desired telemetry streams. You'll need the URI of your telemetry listener for this step. For more information, see [the section called “Sending a subscription request to the Telemetry API” \(p. 916\)](#).
4. Get telemetry data from Lambda via the telemetry listener. You can do any custom processing of this data, such as dispatching the data to Amazon S3 or to an external observability service.

Note

A Lambda function's execution environment can start and stop multiple times as part of its [lifecycle \(p. 900\)](#). In general, your extension code runs during function invocations, and also up to 2 seconds during the shutdown phase. We recommend batching the telemetry as it arrives to your listener, and using the Invoke and Shutdown lifecycle events to dispatch each batch to their desired destinations.

Registering your extension

Before you can subscribe to receive telemetry data, you must register your Lambda extension. Registration occurs during the [extension initialization phase \(p. 902\)](#). The following example shows an HTTP request to register an extension.

```
POST http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
Lambda-Extension-Name: lambda_extension_name
{
  'events': [ 'INVOKE', 'SHUTDOWN' ]
}
```

If the request succeeds, the subscriber receives an HTTP 200 success response. The response header contains the `Lambda-Extension-Identifier`. The response body contains other properties of the function.

```
HTTP/1.1 200 OK
Lambda-Extension-Identifier: a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
{
  "functionName": "lambda_function",
  "functionVersion": "$LATEST",
  "handler": "lambda_handler",
  "accountId": "123456789012"
}
```

For more information, see the [the section called “Extensions API reference” \(p. 906\)](#).

Creating a telemetry listener

Your Lambda extension must have a listener that handles incoming requests from the Telemetry API. The following code shows an example telemetry listener implementation 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
}
```

Specifying a destination protocol

When you subscribe to receive telemetry using the Telemetry API, you can specify a destination protocol in addition to the destination URI:

```
{
    "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
    }
}
```

Lambda accepts two protocols for receiving telemetry:

- **HTTP (recommended)** – Lambda delivers telemetry to a local HTTP endpoint (`http://sandbox.localdomain:${PORT}/${PATH}`) as an array of records in JSON format. The `$PATH` parameter is optional. Lambda supports only HTTP, not HTTPS. Lambda delivers telemetry through POST requests.
- **TCP** – Lambda delivers telemetry to a TCP port in [Newline delimited JSON \(NDJSON\) format](#).

Note

We strongly recommend using HTTP rather than TCP. With TCP, the Lambda platform cannot acknowledge when it delivers telemetry to the application layer. Therefore, if your extension crashes, you might lose telemetry. HTTP does not have this limitation.

Before subscribing to receive telemetry, set up the local HTTP listener or TCP port. During setup, note the following:

- Lambda sends telemetry only to destinations that are inside the execution environment.
- Lambda retries the attempt to send telemetry (with backoff) if there is no listener, or if the POST request results in an error. If the telemetry listener crashes, then it continues to receive telemetry after Lambda restarts the execution environment.
- Lambda reserves port 9001. There are no other port number restrictions or recommendations.

Configuring memory usage and buffering

An execution environment's memory usage increases linearly as the number of subscribers increases. Subscriptions consume memory resources because each subscription opens a new memory buffer to store telemetry data. Buffer memory usage counts towards overall memory consumption in the execution environment.

When you subscribe to receive telemetry using the Telemetry API, you can buffer telemetry data and deliver it to subscribers in batches. To help optimize memory usage, you can specify a buffering configuration:

```
{
  "buffering": {
    "maxBytes": 256*1024,
    "maxItems": 1000,
    "timeoutMs": 100
  }
}
```

Buffering configuration settings

Parameter	Description	Defaults and limits
maxBytes	The maximum volume of telemetry (in bytes) to buffer in memory.	Default: 262,144 Minimum: 262,144 Maximum: 1,048,576
maxItems	The maximum number of events to buffer in memory.	Default: 10,000 Minimum: 1,000 Maximum: 10,000
timeoutMs	The maximum time (in milliseconds) to buffer a batch.	Default: 1,000 Minimum: 25 Maximum: 30,000

When configuring buffering, note the following points:

- If any of the input streams are closed, then Lambda flushes the logs. This can happen if, for example, the runtime crashes.
- Each subscriber can specify a different buffering configuration in their subscription request.
- Consider the buffer size that you need for reading the data. Expect to receive payloads as large as $2 * \text{maxBytes} + \text{metadataBytes}$, where `maxBytes` is part of your buffering configuration. To get an idea of how much `metadataBytes` you should account for, review the following metadata. Lambda adds metadata similar to this to each record:

```
{
  "time": "2022-08-20T12:31:32.123Z",
  "type": "function",
  "record": "Hello World"
}
```

- If the subscriber cannot process incoming telemetry fast enough, Lambda might drop records to keep memory utilization bounded. When this occurs, Lambda sends a `platform.logsDropped` event.

Sending a subscription request to the Telemetry API

Lambda extensions can subscribe to receive telemetry data by sending a subscription request to the Telemetry API. The subscription request should contain information about the types of events that you want the extension to subscribe to. In addition, the request can contain [delivery destination information \(p. 914\)](#) and a [buffering configuration \(p. 915\)](#).

Before sending a subscription request, you must have an extension ID (Lambda-Extension-Identifier). When you [register your extension with the Extensions API \(p. 913\)](#), you obtain an extension ID from the API response.

Subscription occurs during the [extension initialization phase \(p. 902\)](#). The following example shows an HTTP request to subscribe to all three telemetry streams: platform telemetry, function logs, and extension logs.

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
  "schemaVersion": "2022-07-01",
  "types": [
    "platform",
    "function",
    "extension"
  ],
  "buffering": {
    "maxItems": 1000,
    "maxBytes": 256*1024,
    "timeoutMs": 100
  },
  "destination": {
    "protocol": "HTTP",
    "URI": "http://sandbox.localdomain:8080"
  }
}
```

If the request succeeds, then the subscriber receives an HTTP 200 success response.

```
HTTP/1.1 200 OK
"OK"
```

Inbound Telemetry API messages

After subscribing using the Telemetry API, an extension automatically starts to receive telemetry from Lambda via POST requests to the telemetry listener. Each POST request body contains an array of Event objects. Event is a JSON object with the following schema:

```
{
  time: String,
  type: String,
  record: Object
}
```

- The `time` property defines when the Lambda platform generated the event. This isn't the same as when the event actually occurred. The string value of `time` is a timestamp in ISO 8601 format.

- The type property defines the event type. The following table describes all possible values.
- The record property defines a JSON object that contains the telemetry data. The schema of this JSON object depends on the type.

The following table summarizes all types of Event objects, and links to the [Telemetry API Event schema reference \(p. 922\)](#) for each event type.

Telemetry API message types

Category	Event type	Description	Event record schema
Platform event	platform.initStart	Function initialization started.	the section called "platform.initStart" (p. 924) schema
Platform event	platform.initRuntimeDone	Function initialization completed.	the section called "platform.initRuntimeDone" (p. 924) schema
Platform event	platform.initReport	A report of function initialization.	the section called "platform.initReport" (p. 925) schema
Platform event	platform.start	Function invocation started.	the section called "platform.start" (p. 925) schema
Platform event	platform.runtimeDone	The runtime finished processing an event with either success or failure.	the section called "platform.runtimeDone" (p. 926) schema
Platform event	platform.report	A report of function invocation.	the section called "platform.report" (p. 927) schema
Platform event	platform.restoreStart	Runtime restore started.	the section called "platform.restoreStart" (p. 927) schema
Platform event	platform.restoreRuntimeDone	Runtime restore completed.	the section called "platform.restoreRuntimeDone" schema
Platform event	platform.restoreReport	Report of runtime restore.	the section called "platform.restoreReport" (p. 929) schema
Platform event	platform.telemetrySubscription	The extension subscribed to the Telemetry API.	the section called "platform.telemetrySubscription" schema
Platform event	platform.logsDropped	Lambda dropped log entries.	the section called "platform.logsDropped" (p. 930) schema
Function logs	function	A log line from function code.	the section called "function" (p. 931) schema

Category	Event type	Description	Event record schema
Extension logs	extension	A log line from extension code.	the section called "extension" (p. 931) schema

Lambda Telemetry API reference

Use the Lambda Telemetry API endpoint to subscribe extensions to telemetry streams. You can retrieve the Telemetry API endpoint from the AWS_LAMBDA_RUNTIME_API environment variable. To send an API request, append the API version (2022-07-01/) and telemetry/. For example:

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

For the OpenAPI Specification (OAS) definition of the subscription responses version 2022-07-01, see the following:

- **HTTP** – [telemetry-api-http-schema.zip](#)
- **TCP** – [telemetry-api-tcp-schema.zip](#)

API operations

- [Subscribe \(p. 919\)](#)

Subscribe

To subscribe to a telemetry stream, a Lambda extension can send a Subscribe API request.

- **Path** – /telemetry
- **Method** – PUT
- **Headers**
 - Content-Type: application/json
- **Request body parameters**
 - **schemaVersion**
 - Required: Yes
 - Type: String
 - Valid values: "2022-07-01"
 - **destination** – The configuration settings that define the telemetry event destination and the protocol for event delivery.
 - Required: Yes
 - Type: Object

```
{  
    "protocol": "HTTP",  
    "URI": "http://sandbox.localdomain:8080"  
}
```

- **protocol** – The protocol that Lambda uses to send telemetry data.
 - Required: Yes
 - Type: String
 - Valid values: "HTTP"|"TCP"
- **URI** – The URI to send telemetry data to.
 - Required: Yes
 - Type: String
 - For more information, see [the section called "Specifying a destination protocol" \(p. 914\)](#).
- **types** – The types of telemetry that you want the extension to subscribe to.

- Required: Yes
- Type: Array of strings
- Valid values: "platform"|"function"|"extension"
- **buffering** – The configuration settings for event buffering.
 - Required: No
 - Type: Object

```
{  
    "buffering": {  
        "maxItems": 1000,  
        "maxBytes": 256*1024,  
        "timeoutMs": 100  
    }  
}
```

- **maxItems** – The maximum number of events to buffer in memory.
 - Required: No
 - Type: Integer
 - Default: 1,000
 - Minimum: 1,000
 - Maximum: 10,000
- **maxBytes** – The maximum volume of telemetry (in bytes) to buffer in memory.
 - Required: No
 - Type: Integer
 - Default: 262,144
 - Minimum: 262,144
 - Maximum: 1,048,576
- **timeoutMs** – The maximum time (in milliseconds) to buffer a batch.
 - Required: No
 - Type: Integer
 - Default: 1,000
 - Minimum: 25
 - Maximum: 30,000
- For more information, see [the section called “Configuring memory usage and buffering” \(p. 915\)](#).

Example Subscribe API request

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1  
{  
    "schemaVersion": "2022-07-01",  
    "types": [  
        "platform",  
        "function",  
        "extension"  
    ],  
    "buffering": {  
        "maxItems": 1000,  
        "maxBytes": 256*1024,  
        "timeoutMs": 100  
    },  
    "destination": {  
        "protocol": "HTTP",  
        "URI": "http://sandbox.localdomain:8080"  
    }  
}
```

```
    }  
}
```

If the Subscribe request succeeds, the extension receives an HTTP 200 success response:

```
HTTP/1.1 200 OK  
"OK"
```

If the Subscribe request fails, the extension receives an error response. For example:

```
HTTP/1.1 400 OK  
{  
    "errorType": "ValidationException",  
    "errorMessage": "URI port is not provided; types should not be empty"  
}
```

Here are some additional response codes that the extension can receive:

- 200 – Request completed successfully
- 202 – Request accepted. Subscription request response in local testing environment
- 400 – Bad request
- 500 – Service error

Lambda Telemetry API Event schema reference

Use the Lambda Telemetry API endpoint to subscribe extensions to telemetry streams. You can retrieve the Telemetry API endpoint from the `AWS_LAMBDA_RUNTIME_API` environment variable. To send an API request, append the API version (2022-07-01/) and `telemetry/`. For example:

`http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/`

For the OpenAPI Specification (OAS) definition of the subscription responses version 2022-07-01, see the following:

- **HTTP** – [telemetry-api-http-schema.zip](#)
- **TCP** – [telemetry-api-tcp-schema.zip](#)

The following table is a summary of all the types of Event objects that the Telemetry API supports.

Telemetry API message types

Category	Event type	Description	Event record schema
Platform event	<code>platform.initStart</code>	Function initialization started.	the section called "platform.initStart" (p. 924) schema
Platform event	<code>platform.initRuntimeDone</code>	Function initialization completed.	the section called "platform.initRuntimeDone" (p. 924) schema
Platform event	<code>platform.initReport</code>	A report of function initialization.	the section called "platform.initReport" (p. 925) schema
Platform event	<code>platform.start</code>	Function invocation started.	the section called "platform.start" (p. 925) schema
Platform event	<code>platform.runtimeDone</code>	The runtime finished processing an event with either success or failure.	the section called "platform.runtimeDone" (p. 926) schema
Platform event	<code>platform.report</code>	A report of function invocation.	the section called "platform.report" (p. 927) schema
Platform event	<code>platform.restoreStart</code>	Runtime restore started.	the section called "platform.restoreStart" (p. 927) schema
Platform event	<code>platform.restoreRuntimeDone</code>	Runtime restore completed.	the section called "platform.restoreRuntimeDone" (p. 927) schema
Platform event	<code>platform.restoreReport</code>	Report of runtime restore.	the section called "platform.restoreReport" (p. 927) schema

Category	Event type	Description	Event record schema
Platform event	platform.telemetrySubscription	The extension subscribed to the Telemetry API.	the section called "platform.telemetrySubscription" schema
Platform event	platform.logsDropped	Lambda dropped log entries.	the section called "platform.logsDropped" (p. 930) schema
Function logs	function	A log line from function code.	the section called "function" (p. 931) schema
Extension logs	extension	A log line from extension code.	the section called "extension" (p. 931) schema

Contents

- [Telemetry API Event object types \(p. 924\)](#)
 - [platform.initStart \(p. 924\)](#)
 - [platform.initRuntimeDone \(p. 924\)](#)
 - [platform.initReport \(p. 925\)](#)
 - [platform.start \(p. 925\)](#)
 - [platform.runtimeDone \(p. 926\)](#)
 - [platform.report \(p. 927\)](#)
 - [platform.restoreStart \(p. 927\)](#)
 - [platform.restoreRuntimeDone \(p. 928\)](#)
 - [platform.restoreReport \(p. 929\)](#)
 - [platform.extension \(p. 929\)](#)
 - [platform.telemetrySubscription \(p. 930\)](#)
 - [platform.logsDropped \(p. 930\)](#)
 - [function \(p. 931\)](#)
 - [extension \(p. 931\)](#)
- [Shared object types \(p. 931\)](#)
 - [InitPhase \(p. 931\)](#)
 - [InitReportMetrics \(p. 932\)](#)
 - [InitType \(p. 932\)](#)
 - [ReportMetrics \(p. 932\)](#)
 - [RestoreReportMetrics \(p. 932\)](#)
 - [RuntimeDoneMetrics \(p. 933\)](#)
 - [Span \(p. 933\)](#)
 - [Status \(p. 933\)](#)
 - [TraceContext \(p. 934\)](#)
 - [TracingType \(p. 934\)](#)

Telemetry API Event object types

This section details the types of Event objects that the Lambda Telemetry API supports. In the event descriptions, a question mark (?) indicates that the attribute may not be present in the object.

platform.initStart

A platform.initStart event indicates that the function initialization phase has started. A platform.initStart Event object has the following shape:

```
Event: Object
- time: String
- type: String = platform.initStart
- record: PlatformInitStart
```

The PlatformInitStart object has the following attributes:

- **initializationType** – [the section called “InitType” \(p. 932\)](#) object
- **phase** – [the section called “InitPhase” \(p. 931\)](#) object
- **runtimeVersion?** – String
- **runtimeVersionArn?** – String

The following is an example Event of type platform.initStart:

```
{
  "time": "2022-10-12T00:00:15.064Z",
  "type": "platform.initStart",
  "record": {
    "initializationType": "on-demand",
    "phase": "init",
    "runtimeVersion": "nodejs-14.v3",
    "runtimeVersionArn": "arn"
  }
}
```

platform.initRuntimeDone

A platform.initRuntimeDone event indicates that the function initialization phase has completed. A platform.initRuntimeDone Event object has the following shape:

```
Event: Object
- time: String
- type: String = platform.initRuntimeDone
- record: PlatformInitRuntimeDone
```

The PlatformInitRuntimeDone object has the following attributes:

- **initializationType** – [the section called “InitType” \(p. 932\)](#) object
- **phase** – [the section called “InitPhase” \(p. 931\)](#) object
- **status** – [the section called “Status” \(p. 933\)](#) object
- **spans?** – List of [the section called “Span” \(p. 933\)](#) objects

The following is an example Event of type 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

A platform.initReport event contains an overall report of the function initialization phase. A platform.initReport Event object has the following shape:

```
Event: Object  
- time: String  
- type: String = platform.initReport  
- record: PlatformInitReport
```

The PlatformInitReport object has the following attributes:

- **initializationType** – [the section called “InitType” \(p. 932\)](#) object
- **phase** – [the section called “InitPhase” \(p. 931\)](#) object
- **metrics** – [the section called “InitReportMetrics” \(p. 932\)](#) object
- **spans?** – List of [the section called “Span” \(p. 933\)](#) objects

The following is an example Event of type platform.initReport:

```
{  
    "time": "2022-10-12T00:01:15.000Z",  
    "type": "platform.initReport",  
    "record": {  
        "initializationType": "on-demand",  
        "phase": "init",  
        "metrics": {  
            "durationMs": 125.33  
        },  
        "spans": [  
            {  
                "name": "someTimeSpan",  
                "start": "2022-06-02T12:02:33.913Z",  
                "durationMs": 90.1  
            }  
        ]  
    }  
}
```

platform.start

A platform.start event indicates that the function invocation phase has started. A platform.start Event object has the following shape:

```
Event: Object
- time: String
- type: String = platform.start
- record: PlatformStart
```

The PlatformStart object has the following attributes:

- **requestId** – String
- **version?** – String
- **tracing?** – [the section called “TraceContext” \(p. 934\)](#)

The following is an example Event of type platform.start:

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.initStart",
    "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

A platform.runtimeDone event indicates that the function invocation phase has completed. A platform.runtimeDone Event object has the following shape:

```
Event: Object
- time: String
- type: String = platform.runtimeDone
- record: PlatformRuntimeDone
```

The PlatformRuntimeDone object has the following attributes:

- **errorType?** – String
- **metrics?** – [the section called “RuntimeDoneMetrics” \(p. 933\)](#) object
- **requestId** – String
- **status** – [the section called “Status” \(p. 933\)](#) object
- **spans?** – List of [the section called “Span” \(p. 933\)](#) objects
- **tracing?** – [the section called “TraceContext” \(p. 934\)](#) object

The following is an example Event of type 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

A `platform.report` event contains an overall report of the function initialization phase. A `platform.report` Event object has the following shape:

```

Event: Object
- time: String
- type: String = platform.report
- record: PlatformReport

```

The `PlatformReport` object has the following attributes:

- **metrics** – [the section called “ReportMetrics” \(p. 932\)](#) object
- **requestId** – String
- **spans?** – List of [the section called “Span” \(p. 933\)](#) objects
- **status** – [the section called “Status” \(p. 933\)](#) object
- **tracing?** – [the section called “TraceContext” \(p. 934\)](#) object

The following is an example Event of type `platform.report`:

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.report",
    "record": {
        "metrics": {
            "durationMs": 694,
            "initDurationMs": 397.68,
            "maxMemoryUsedMB": 84,
            "memorySizeMB": 128
        },
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
    }
}
```

platform.restoreStart

A `platform.restoreStart` event indicates that a function environment restoration event started. In an environment restoration event, Lambda creates the environment from a cached snapshot.

rather than initializing it from scratch. For more information, see [Lambda SnapStart \(p. 992\)](#). A `platform.restoreStart` Event object has the following shape:

```
Event: Object
- time: String
- type: String = platform.restoreStart
- record: PlatformRestoreStart
```

The `PlatformRestoreStart` object has the following attributes:

- **runtimeVersion?** – String
- **runtimeVersionArn?** – String

The following is an example Event of type `platform.restoreStart`:

```
{ "time": "2022-10-12T00:00:15.064Z",
  "type": "platform.restoreStart",
  "record": {
    "runtimeVersion": "nodejs-14.v3",
    "runtimeVersionArn": "arn"
  }
}
```

platform.restoreRuntimeDone

A `platform.restoreRuntimeDone` event indicates that a function environment restoration event completed. In an environment restoration event, Lambda creates the environment from a cached snapshot rather than initializing it from scratch. For more information, see [Lambda SnapStart \(p. 992\)](#). A `platform.restoreRuntimeDone` Event object has the following shape:

```
Event: Object
- time: String
- type: String = platform.restoreRuntimeDone
- record: PlatformRestoreRuntimeDone
```

The `PlatformRestoreRuntimeDone` object has the following attributes:

- **errorType?** – String
- **spans?** – List of [the section called “Span” \(p. 933\)](#) objects
- **status** – [the section called “Status” \(p. 933\)](#) object

The following is an example Event of type `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

A `platform.restoreReport` event contains an overall report of a function restoration event. A `platform.restoreReport` Event object has the following shape:

```
Event: Object
- time: String
- type: String = platform.restoreReport
- record: PlatformRestoreReport
```

The `PlatformRestoreReport` object has the following attributes:

- **metrics?** – [the section called “RestoreReportMetrics” \(p. 932\)](#) object
- **spans?** – List of [the section called “Span” \(p. 933\)](#) objects

The following is an example Event of type `platform.restoreReport`:

```
{
  "time": "2022-10-12T00:00:15.064Z",
  "type": "platform.restoreReport",
  "record": {
    "metrics": {
      "durationMs": 15.19
    },
    "spans": [
      {
        "name": "someTimeSpan",
        "start": "2022-08-02T12:01:23:521Z",
        "durationMs": 30.0
      }
    ]
  }
}
```

platform.extension

An extension event contains logs from the extension code. An `extension` Event object has the following shape:

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

The `PlatformExtension` object has the following attributes:

- **events** – List of String
- **name** – String
- **state** – String

The following is an example Event of type `platform.extension`:

```
{
  "time": "2022-10-12T00:02:15.000Z",
```

```
"type": "platform.extension",
"record": {
    "events": [ "INVOKE", "SHUTDOWN" ],
    "name": "my-telemetry-extension",
    "state": "Ready"
}
```

platform.telemetrySubscription

A platform.telemetrySubscription event contains information about an extension subscription. A platform.telemetrySubscription Event object has the following shape:

```
Event: Object
- time: String
- type: String = platform.telemetrySubscription
- record: PlatformTelemetrySubscription
```

The PlatformTelemetrySubscription object has the following attributes:

- **name** – String
- **state** – String
- **types** – List of String

The following is an example Event of type platform.telemetrySubscription:

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.telemetrySubscription",
    "record": {
        "name": "my-telemetry-extension",
        "state": "Subscribed",
        "types": [ "platform", "function" ]
    }
}
```

platform.logsDropped

A platform.logsDropped event contains information about dropped events. Lambda emits the platform.logsDropped event when an extension can't process one or more events. A platform.logsDropped Event object has the following shape:

```
Event: Object
- time: String
- type: String = platform.logsDropped
- record: PlatformLogsDropped
```

The PlatformLogsDropped object has the following attributes:

- **droppedBytes** – Integer
- **droppedRecords** – Integer
- **reason** – String

The following is an example Event of type platform.logsDropped:

```
{
```

```
"time": "2022-10-12T00:02:35.000Z",
"type": "platform.logsDropped",
"record": [
    "droppedBytes": 12345,
    "droppedRecords": 123,
    "reason": "Consumer seems to have fallen behind as it has not acknowledged receipt
of logs."
]
}
```

function

A function event contains logs from the function code. A Function Event object has the following shape:

```
Event: Object
- time: String
- type: String = function
- record: {}
```

The following is an example Event of type function:

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": "[INFO] Hello world, I am a function!"
}
```

extension

A extension event contains logs from the extension code. A extension Event object has the following shape:

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

The following is an example Event of type extension:

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": "[INFO] Hello world, I am an extension!"
}
```

Shared object types

This section details the types of shared objects that the Lambda Telemetry API supports.

InitPhase

A string enum that describes the phase when the initialization step occurs. In most cases, Lambda runs the function initialization code during the init phase. However, in some error cases, Lambda may re-run the function initialization code during the invoke phase. (This is called a *suppressed init*.)

- **Type – String**
- **Valid values – init|invoke|snap-start**

InitReportMetrics

An object that contains metrics about an initialization phase.

- **Type** – Object

An `InitReportMetrics` object has the following shape:

```
InitReportMetrics: Object
- durationMs: Double
```

The following is an example `InitReportMetrics` object:

```
{
    "durationMs": 247.88
}
```

InitType

A string enum that describes how Lambda initialized the environment.

- **Type** – String
- **Valid values** – on-demand|provisioned-concurrency

ReportMetrics

An object that contains metrics about a completed phase.

- **Type** – Object

A `ReportMetrics` object has the following shape:

```
ReportMetrics: Object
- billedDurationMs: Integer
- durationMs: Double
- initDurationMs?: Double
- maxMemoryUsedMB: Integer
- memorySizeMB: Integer
- restoreDurationMs?: Double
```

The following is an example `ReportMetrics` object:

```
{
    "billedDurationMs": 694,
    "durationMs": 693.92,
    "initDurationMs": 397.68,
    "maxMemoryUsedMB": 84,
    "memorySizeMB": 128
}
```

RestoreReportMetrics

An object that contains metrics about a completed restoration phase.

- **Type** – Object

A `RestoreReportMetrics` object has the following shape:

```
RestoreReportMetrics: Object
- durationMs: Double
```

The following is an example `RestoreReportMetrics` object:

```
{ "durationMs": 15.19 }
```

RuntimeDoneMetrics

An object that contains metrics about an invocation phase.

- **Type** – Object

A `RuntimeDoneMetrics` object has the following shape:

```
RuntimeDoneMetrics: Object
- durationMs: Double
- producedBytes?: Integer
```

The following is an example `RuntimeDoneMetrics` object:

```
{ "durationMs": 200.0,
  "producedBytes": 15 }
```

Span

An object that contains details about a span. A span represents a unit of work or operation in a trace. For more information about spans, see [Span](#) on the [Tracing API](#) page of the OpenTelemetry Docs website.

Lambda supports the following two spans for the `platform.RuntimeDone` event:

- The `responseLatency` span describes how long it took your Lambda function to start sending the response.
- The `responseDuration` span describes how long it took your Lambda function to finish sending the entire response.

The following is an example `responseLatency` span object:

```
{ "name": "responseLatency",
  "start": "2022-08-02T12:01:23.521Z",
  "durationMs": 23.02 }
```

Status

An object that describes the status of an initialization or invocation phase. If the status is either `failure` or `error`, then the `Status` object also contains an `errorType` field describing the error.

- **Type** – Object
- **Valid status values** – success|failure|error|timeout

TraceContext

An object that describes the properties of a trace.

- **Type** – Object

A TraceContext object has the following shape:

```
TraceContext: Object
- spanId?: String
- type: TracingType enum
- value: String
```

The following is an example TraceContext object:

```
{
  "spanId": "073a49012f3c312e",
  "type": "X-Amzn-Trace-Id",
  "value": "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
}
```

TracingType

A string enum that describes the type of tracing in a [the section called “TraceContext” \(p. 934\)](#) object.

- **Type** – String
- **Valid values** – X-Amzn-Trace-Id

Converting Lambda Telemetry API Event objects to OpenTelemetry Spans

The AWS Lambda Telemetry API schema is semantically compatible with OpenTelemetry (OTel). This means that you can convert your AWS Lambda Telemetry API Event objects to OpenTelemetry (OTel) Spans. When converting, you shouldn't map a single Event object to a single OTel Span. Instead, you should present all three events related to a lifecycle phase in a single OTel Span. For example, the `start`, `runtimeDone`, and `runtimeReport` events represent a single function invocation. Present all three of these events as a single OTel Span.

You can convert your events using Span Events or Child (nested) Spans. The tables on this page describe the mappings between Telemetry API schema properties and OTel Span properties for both approaches. For more information about OTel Spans, see [Span](#) on the **Tracing API** page of the OpenTelemetry Docs website.

Sections

- [Map to OTel Spans with Span Events \(p. 935\)](#)
- [Map to OTel Spans with Child Spans \(p. 937\)](#)

Map to OTel Spans with Span Events

In the following tables, `e` represents the event coming from the telemetry source.

Mapping the *Start events

OpenTelemetry	Lambda Telemetry API schema
<code>Span.Name</code>	Your extension generates this value based on the <code>type</code> field.
<code>Span.StartTime</code>	Use <code>e.time</code> .
<code>Span.EndTime</code>	N/A, because the event hasn't completed yet.
<code>Span.Kind</code>	Set to <code>Server</code> .
<code>Span.Status</code>	Set to <code>Unset</code> .
<code>Span.TraceId</code>	Parse the AWS X-Ray header found in <code>e.tracing.value</code> , then use the <code>TraceId</code> value.
<code>Span.ParentId</code>	Parse the X-Ray header found in <code>e.tracing.value</code> , then use the <code>Parent</code> value.
<code>Span.SpanId</code>	Use <code>e.tracing.spanId</code> if available. Otherwise, generate a new <code>SpanId</code> .
<code>Span.SpanContext.TraceState</code>	N/A for an X-Ray trace context.
<code>Span.SpanContext.TraceFlags</code>	Parse the X-Ray header found in <code>e.tracing.value</code> , then use the <code>Sampled</code> value.
<code>Span.Attributes</code>	Your extension can add any custom values here.

Mapping the *RuntimeDone events

OpenTelemetry	Lambda Telemetry API schema
Span.Name	Your extension generates the value based on the type field.
Span.StartTime	Use e.time from the matching *Start event. Alternatively, use e.time - e.metrics.durationMs.
Span.EndTime	N/A, because the event hasn't completed yet.
Span.Kind	Set to Server.
Span.Status	If e.status doesn't equal success, then set to Error. Otherwise, set to Ok.
Span.Events[]	Use e.spans[].
Span.Events[i].Name	Use e.spans[i].name.
Span.Events[i].Time	Use e.spans[i].start.
Span.TraceId	Parse the AWS X-Ray header found in e.tracing.value, then use the TraceId value.
Span.ParentId	Parse the X-Ray header found in e.tracing.value, then use the Parent value.
Span.SpanId	Use the same SpanId from the *Start event. If unavailable, then use e.tracing.spanId, or generate a new SpanId.
Span.SpanContext.TraceState	N/A for an X-Ray trace context.
Span.SpanContext.TraceFlags	Parse the X-Ray header found in e.tracing.value, then use the Sampled value.
Span.Attributes	Your extension can add any custom values here.

Mapping the *Report events

OpenTelemetry	Lambda Telemetry API schema
Span.Name	Your extension generates the value based on the type field.
Span.StartTime	Use e.time from the matching *Start event. Alternatively, use e.time - e.metrics.durationMs.
Span.EndTime	Use e.time.
Span.Kind	Set to Server.

OpenTelemetry	Lambda Telemetry API schema
Span.Status	Use the same value as the *RuntimeDone event.
Span.TraceId	Parse the AWS X-Ray header found in e.tracing.value, then use the TraceId value.
Span.ParentId	Parse the X-Ray header found in e.tracing.value, then use the Parent value.
Span.SpanId	Use the same SpanId from the *Start event. If unavailable, then use e.tracing.spanId, or generate a new SpanId.
Span.SpanContext.TraceState	N/A for an X-Ray trace context.
Span.SpanContext.TraceFlags	Parse the X-Ray header found in e.tracing.value, then use the Sampled value.
Span.Attributes	Your extension can add any custom values here.

Map to OTel Spans with Child Spans

The following table describes how to convert Lambda Telemetry API events into OTel Spans with Child (nested) Spans for *RuntimeDone Spans. For *Start and *Report mappings, refer to the tables in [the section called "Map to OTel Spans with Span Events" \(p. 935\)](#), as they're the same for Child Spans. In this table, e represents the event coming from the telemetry source.

Mapping the *RuntimeDone events

OpenTelemetry	Lambda Telemetry API schema
Span.Name	Your extension generates the value based on the type field.
Span.StartTime	Use e.time from the matching *Start event. Alternatively, use e.time - e.metrics.durationMs.
Span.EndTime	N/A, because the event hasn't completed yet.
Span.Kind	Set to Server.
Span.Status	If e.status doesn't equal success, then set to Error. Otherwise, set to Ok.
Span.TraceId	Parse the AWS X-Ray header found in e.tracing.value, then use the TraceId value.
Span.ParentId	Parse the X-Ray header found in e.tracing.value, then use the Parent value.
Span.SpanId	Use the same SpanId from the *Start event. If unavailable, then use e.tracing.spanId, or generate a new SpanId.

OpenTelemetry	Lambda Telemetry API schema
Span.SpanContext.TraceState	N/A for an X-Ray trace context.
Span.SpanContext.TraceFlags	Parse the X-Ray header found in e.tracing.value, then use the Sampled value.
Span.Attributes	Your extension can add any custom values here.
ChildSpan[i].Name	Use e.spans[i].name.
ChildSpan[i].StartTime	Use e.spans[i].start.
ChildSpan[i].EndTime	Use e.spans[i].start + e.spans[i].durations.
ChildSpan[i].Kind	Same as parent Span.Kind.
ChildSpan[i].Status	Same as parent Span.Status.
ChildSpan[i].TraceId	Same as parent Span.TraceId.
ChildSpan[i].ParentId	Use parent Span.SpanId.
ChildSpan[i].SpanId	Generate a new SpanId.
ChildSpan[i].SpanContext.TraceState	N/A for an X-Ray trace context.
ChildSpan[i].SpanContext.TraceFlags	Same as parent Span.SpanContext.TraceFlags.

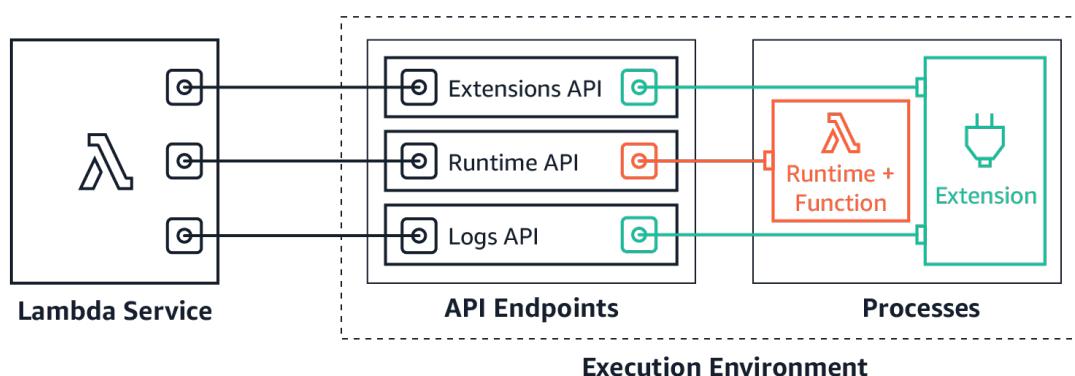
Lambda Logs API

Important

The Lambda Telemetry API supersedes the Lambda Logs API. **While the Logs API remains fully functional, we recommend using only the Telemetry API going forward.** You can subscribe your extension to a telemetry stream using either the Telemetry API or the Logs API. After subscribing using one of these APIs, any attempt to subscribe using the other API returns an error.

Lambda automatically captures runtime logs and streams them to Amazon CloudWatch. This log stream contains the logs that your function code and extensions generate, and also the logs that Lambda generates as part of the function invocation.

[Lambda extensions \(p. 900\)](#) can use the Lambda Runtime Logs API to subscribe to log streams directly from within the Lambda [execution environment \(p. 14\)](#). Lambda streams the logs to the extension, and the extension can then process, filter, and send the logs to any preferred destination.



The Logs API allows extensions to subscribe to three different logs streams:

- Function logs that the Lambda function generates and writes to `stdout` or `stderr`.
- Extension logs that extension code generates.
- Lambda platform logs, which record events and errors related to invocations and extensions.

Note

Lambda sends all logs to CloudWatch, even when an extension subscribes to one or more of the log streams.

Topics

- [Subscribing to receive logs \(p. 940\)](#)
- [Memory usage \(p. 940\)](#)
- [Destination protocols \(p. 940\)](#)
- [Buffering configuration \(p. 940\)](#)
- [Example subscription \(p. 941\)](#)
- [Sample code for Logs API \(p. 941\)](#)
- [Logs API reference \(p. 941\)](#)
- [Log messages \(p. 942\)](#)

Subscribing to receive logs

A Lambda extension can subscribe to receive logs by sending a subscription request to the Logs API.

To subscribe to receive logs, you need the extension identifier (`Lambda-Extension-Identifier`). First [register the extension \(p. 907\)](#) to receive the extension identifier. Then subscribe to the Logs API during [initialization \(p. 15\)](#). After the initialization phase completes, Lambda does not process subscription requests.

Note

Logs API subscription is idempotent. Duplicate subscribe requests do not result in duplicate subscriptions.

Memory usage

Memory usage increases linearly as the number of subscribers increases. Subscriptions consume memory resources because each subscription opens a new memory buffer to store the logs. To help optimize memory usage, you can adjust the [buffering configuration \(p. 940\)](#). Buffer memory usage counts towards overall memory consumption in the execution environment.

Destination protocols

You can choose one of the following protocols to receive the logs:

1. **HTTP** (recommended) – Lambda delivers logs to a local HTTP endpoint (`http://sandbox.localdomain:${PORT}/${PATH}`) as an array of records in JSON format. The `$PATH` parameter is optional. Note that only HTTP is supported, not HTTPS. You can choose to receive logs through PUT or POST.
2. **TCP** – Lambda delivers logs to a TCP port in [Newline delimited JSON \(NDJSON\) format](#).

We recommend using HTTP rather than TCP. With TCP, the Lambda platform cannot acknowledge when it delivers logs to the application layer. Therefore, you might lose logs if your extension crashes. HTTP does not share this limitation.

We also recommend setting up the local HTTP listener or the TCP port before subscribing to receive logs. During setup, note the following:

- Lambda sends logs only to destinations that are inside the execution environment.
- Lambda retries the attempt to send the logs (with backoff) if there is no listener, or if the POST or PUT request results in an error. If the log subscriber crashes, it continues to receive logs after Lambda restarts the execution environment.
- Lambda reserves port 9001. There are no other port number restrictions or recommendations.

Buffering configuration

Lambda can buffer logs and deliver them to the subscriber. You can configure this behavior in the subscription request by specifying the following optional fields. Note that Lambda uses the default value for any field that you do not specify.

- **timeoutMs** – The maximum time (in milliseconds) to buffer a batch. Default: 1,000. Minimum: 25. Maximum: 30,000.
- **maxBytes** – The maximum size (in bytes) of the logs to buffer in memory. Default: 262,144. Minimum: 262,144. Maximum: 1,048,576.
- **maxItems** – The maximum number of events to buffer in memory. Default: 10,000. Minimum: 1,000. Maximum: 10,000.

During buffering configuration, note the following points:

- Lambda flushes the logs if any of the input streams are closed, for example, if the runtime crashes.
- Each subscriber can specify a different buffering configuration in their subscription request.
- Consider the buffer size that you need for reading the data. Expect to receive payloads as large as $2 * \text{maxBytes} + \text{metadata}$, where maxBytes is configured in the subscribe request. For example, Lambda adds the following metadata bytes to each record:

```
{  
  "time": "2020-08-20T12:31:32.123Z",  
  "type": "function",  
  "record": "Hello World"  
}
```

- If the subscriber cannot process incoming logs quickly enough, Lambda might drop logs to keep memory utilization bounded. To indicate the number of dropped records, Lambda sends a platform.logsDropped log.

Example subscription

The following example shows a request to subscribe to the platform and function logs.

```
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"  
  }  
}
```

If the request succeeds, the subscriber receives an HTTP 200 success response.

```
HTTP/1.1 200 OK  
"OK"
```

Sample code for Logs API

For sample code showing how to send logs to a custom destination, see [Using AWS Lambda extensions to send logs to custom destinations](#) on the AWS Compute Blog.

For Python and Go code examples showing how to develop a basic Lambda extension and subscribe to the Logs API, see [AWS Lambda Extensions](#) on the AWS Samples GitHub repository. For more information about building a Lambda extension, see [the section called “Extensions API” \(p. 900\)](#).

Logs API reference

You can retrieve the Logs API endpoint from the AWS_LAMBDA_RUNTIME_API environment variable. To send an API request, use the prefix 2020-08-15/ before the API path. For example:

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs
```

The OpenAPI specification for the Logs API version **2020-08-15** is available here: [logs-api-request.zip](#)

Subscribe

To subscribe to one or more of the log streams available in the Lambda execution environment, extensions send a Subscribe API request.

Path – /logs

Method – PUT

Body parameters

destination – See [the section called “Destination protocols” \(p. 940\)](#). Required: yes. Type: strings.

buffering – See [the section called “Buffering configuration” \(p. 940\)](#). Required: no. Type: strings.

types – An array of the types of logs to receive. Required: yes. Type: array of strings. Valid values: "platform", "function", "extension".

schemaVersion – Required: no. Default value: "2020-08-15". Set to "2021-03-18" for the extension to receive [platform.runtimeDone \(p. 945\)](#) messages.

Response parameters

The OpenAPI specifications for the subscription responses version **2020-08-15** are available for the HTTP and TCP protocols:

- HTTP: [logs-api-http-response.zip](#)
- TCP: [logs-api-tcp-response.zip](#)

Response codes

- 200 – Request completed successfully
- 202 – Request accepted. Response to a subscription request during local testing.
- 4XX – Bad Request
- 500 – Service error

If the request succeeds, the subscriber receives an HTTP 200 success response.

```
HTTP/1.1 200 OK
"OK"
```

If the request fails, the subscriber receives an error response. For example:

```
HTTP/1.1 400 OK
{
    "errorType": "Logs.ValidationError",
    "errorMessage": "URI port is not provided; types should not be empty"
}
```

Log messages

The Logs API allows extensions to subscribe to three different logs streams:

- Function – Logs that the Lambda function generates and writes to `stdout` or `stderr`.
- Extension – Logs that extension code generates.
- Platform – Logs that the runtime platform generates, which record events and errors related to invocations and extensions.

Topics

- [Function logs \(p. 943\)](#)
- [Extension logs \(p. 943\)](#)
- [Platform logs \(p. 943\)](#)

Function logs

The Lambda function and internal extensions generate function logs and write them to `stdout` or `stderr`.

The following example shows the format of a function log message. { "time": "2020-08-20T12:31:32.123Z", "type": "function", "record": "ERROR encountered. Stack trace:\n\my-function (line 10)\n" }

Extension logs

Extensions can generate extension logs. The log format is the same as for a function log.

Platform logs

Lambda generates log messages for platform events such as `platform.start`, `platform.end`, and `platform.fault`.

Optionally, you can subscribe to the **2021-03-18** version of the Logs API schema, which includes the `platform.runtimeDone` log message.

Example platform log messages

The following example shows the platform start and platform end logs. These logs indicate the invocation start time and invocation end time for the invocation that the `requestId` specifies.

```
{  
  "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"}  
}
```

The `platform.initRuntimeDone` log message shows the status of the Runtime `init` sub-phase, which is part of the [Init lifecycle phase \(p. 15\)](#). When Runtime `init` is successful, the runtime sends a `/next` runtime API request (for the on-demand and provisioned-concurrency initialization types) or `restore/next` (for the snap-start initialization type). The following example shows a successful `platform.initRuntimeDone` log message for the snap-start initialization type.

```
{  
  "time": "2022-07-17T18:41:57.083Z",  
  "type": "platform.initRuntimeDone",  
  "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}  
}
```

```
{
  "record": {
    "initializationType": "snap-start",
    "status": "success"
  }
}
```

The **platform.initReport** log message shows how long the Init phase lasted and how many milliseconds you were billed for during this phase. When the initialization type is provisioned-concurrency, Lambda sends this message during invocation. When the initialization type is snap-start, Lambda sends this message after restoring the snapshot. The following example shows a **platform.initReport** log message for the snap-start initialization type.

```
{
  "time": "2022-07-17T18:41:57.083Z",
  "type": "platform.initReport",
  "record": {
    "initializationType": "snap-start",
    "metrics": {
      "durationMs": 731.79,
      "billedDurationMs": 732
    }
  }
}
```

The platform report log includes metrics about the invocation that the requestId specifies. The `initDurationMs` field is included in the log only if the invocation included a cold start. If AWS X-Ray tracing is active, the log includes X-Ray metadata. The following example shows a platform report log for an invocation that included a cold start.

```
{
  "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
    }
  }
}
```

The platform fault log captures runtime or execution environment errors. The following example shows a platform fault log message.

```
{
  "time": "2020-08-20T12:31:32.123Z",
  "type": "platform.fault",
  "record": "RequestId: d783b35e-a91d-4251-af17-035953428a2c Process exited before completing request"
}
```

Lambda generates a platform extension log when an extension registers with the extensions API. The following example shows a platform extension message.

```
{
  "time": "2020-08-20T12:31:32.123Z",
  "type": "platform.extension",
  "record": {"name": "Foo.bar",
```

```
        "state": "Ready",
        "events": ["INVOKE", "SHUTDOWN"]
    }
}
```

Lambda generates a platform logs subscription log when an extension subscribes to the logs API. The following example shows a logs subscription message.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsSubscription",
    "record": {"name": "Foo.bar",
               "state": "Subscribed",
               "types": ["function", "platform"],
    }
}
```

Lambda generates a platform logs dropped log when an extension is not able to process the number of logs that it is receiving. The following example shows a platform.logsDropped log message.

```
{
    "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
    }
}
```

The **platform.restoreStart** log message shows the time that the Restore phase started (snap-start initialization type only). Example:

```
{
    "time": "2022-07-17T18:43:44.782Z",
    "type": "platform.restoreStart",
    "record": {}
}
```

The **platform.restoreReport** log message shows how long the Restore phase lasted and how many milliseconds you were billed for during this phase (snap-start initialization type only). Example:

```
{
    "time": "2022-07-17T18:43:45.936Z",
    "type": "platform.restoreReport",
    "record": {
        "metrics": {
            "durationMs": 70.87,
            "billedDurationMs": 13
        }
    }
}
```

Platform runtimeDone messages

If you set the schema version to "2021-03-18" in the subscribe request, Lambda sends a platform.runtimeDone message after the function invocation completes either successfully or with an error. The extension can use this message to stop all the telemetry collection for this function invocation.

The OpenAPI specification for the Log event type in schema version **2021-03-18** is available here:
[schema-2021-03-18.zip](#)

Lambda generates the `platform.runtimeDone` log message when the runtime sends a `Next` or `Error` runtime API request. The `platform.runtimeDone` log informs consumers of the Logs API that the function invocation completes. Extensions can use this information to decide when to send all the telemetry collected during that invocation.

Examples

Lambda sends the `platform.runtimeDone` message after the runtime sends the `NEXT` request when the function invocation completes. The following examples show messages for each of the status values: success, failure, and timeout.

Example Example success message

```
{  
  "time": "2021-02-04T20:00:05.123Z",  
  "type": "platform.runtimeDone",  
  "record": {  
    "requestId": "6f7f0961f83442118a7af6fe80b88",  
    "status": "success"  
  }  
}
```

Example Example failure message

```
{  
  "time": "2021-02-04T20:00:05.123Z",  
  "type": "platform.runtimeDone",  
  "record": {  
    "requestId": "6f7f0961f83442118a7af6fe80b88",  
    "status": "failure"  
  }  
}
```

Example Example timeout message

```
{  
  "time": "2021-02-04T20:00:05.123Z",  
  "type": "platform.runtimeDone",  
  "record": {  
    "requestId": "6f7f0961f83442118a7af6fe80b88",  
    "status": "timeout"  
  }  
}
```

Example Example `platform.restoreRuntimeDone` message (`snap-start` initialization type only)

The `platform.restoreRuntimeDone` log message shows whether or not the Restore phase was successful. Lambda sends this message when the runtime sends a `restore/next` runtime API request. There are three possible statuses: success, failure, and timeout. The following example shows a successful `platform.restoreRuntimeDone` log message.

```
{  
  "time": "2022-07-17T18:43:45.936Z",  
  "type": "platform.restoreRuntimeDone",  
}
```

```
    "record":{  
        "status":"success"  
    }  
}
```

Troubleshooting issues in Lambda

The following topics provide troubleshooting advice for errors and issues that you might encounter when using the Lambda API, console, or tools. If you find an issue that is not listed here, you can use the [Feedback](#) button on this page to report it.

For more troubleshooting advice and answers to common support questions, visit the [AWS Knowledge Center](#).

For more information about debugging and troubleshooting Lambda applications, see [Debugging](#) in the *Lambda operator guide*.

Topics

- [Troubleshoot deployment issues in Lambda \(p. 948\)](#)
- [Troubleshoot invocation issues in Lambda \(p. 951\)](#)
- [Troubleshoot execution issues in Lambda \(p. 955\)](#)
- [Troubleshoot networking issues in Lambda \(p. 957\)](#)
- [Troubleshoot container image issues in Lambda \(p. 958\)](#)

Troubleshoot deployment issues in Lambda

When you update your function, Lambda deploys the change by launching new instances of the function with the updated code or settings. Deployment errors prevent the new version from being used and can arise from issues with your deployment package, code, permissions, or tools.

When you deploy updates to your function directly with the Lambda API or with a client such as the AWS CLI, you can see errors from Lambda directly in the output. If you use services like AWS CloudFormation, AWS CodeDeploy, or AWS CodePipeline, look for the response from Lambda in the logs or event stream for that service.

General: Permission is denied / Cannot load such file

Error: *EACCES: permission denied, open '/var/task/index.js'*

Error: *cannot load such file -- function*

Error: *[Errno 13] Permission denied: '/var/task/function.py'*

The Lambda runtime needs permission to read the files in your deployment package. You can use the chmod command to change the file mode. The following example commands make all files and folders in the current directory readable by any user.

```
chmod -R o+rX .
```

General: Error occurs when calling the UpdateFunctionCode

Error: *An error occurred (RequestEntityTooLargeException) when calling the UpdateFunctionCode operation*

When you upload a deployment package or layer archive directly to Lambda, the size of the ZIP file is limited to 50 MB. To upload a larger file, store it in Amazon S3 and use the `S3Bucket` and `S3Key` parameters.

Note

When you upload a file directly with the AWS CLI, AWS SDK, or otherwise, the binary ZIP file is converted to base64, which increases its size by about 30%. To allow for this, and the size of other parameters in the request, the actual request size limit that Lambda applies is larger. Due to this, the 50 MB limit is approximate.

Amazon S3: Error Code PermanentRedirect.

Error: *Error occurred while GetObject. S3 Error Code: PermanentRedirect. S3 Error Message: The bucket is in this region: us-east-2. Please use this region to retry the request*

When you upload a function's deployment package from an Amazon S3 bucket, the bucket must be in the same Region as the function. This issue can occur when you specify an Amazon S3 object in a call to [UpdateFunctionCode \(p. 1367\)](#), or use the package and deploy commands in the AWS CLI or AWS SAM CLI. Create a deployment artifact bucket for each Region where you develop applications.

General: Cannot find, cannot load, unable to import, class not found, no such file or directory

Error: *Cannot find module 'function'*

Error: *cannot load such file -- function*

Error: *Unable to import module 'function'*

Error: *Class not found: function.Handler*

Error: *fork/exec /var/task/function: no such file or directory*

Error: *Unable to load type 'Function.Handler' from assembly 'Function'.*

The name of the file or class in your function's handler configuration doesn't match your code. See the following entry for more information.

General: Undefined method handler

Error: *index.handler is undefined or not exported*

Error: *Handler 'handler' missing on module 'function'*

Error: *undefined method `handler' for #<LambdaHandler:0x000055b76ccbf98>*

Error: *No public method named handleRequest with appropriate method signature found on class function.Handler*

Error: *Unable to find method 'handleRequest' in type 'Function.Handler' from assembly 'Function'*

The name of the handler method in your function's handler configuration doesn't match your code. Each runtime defines a naming convention for handlers, such as `filename.methodname`. The handler is the method in your function's code that the runtime runs when your function is invoked.

For some languages, Lambda provides a library with an interface that expects a handler method to have a specific name. For details about handler naming for each language, see the following topics.

- [Building Lambda functions with Node.js \(p. 254\)](#)
- [Building Lambda functions with Python \(p. 318\)](#)
- [Building Lambda functions with Ruby \(p. 361\)](#)
- [Building Lambda functions with Java \(p. 386\)](#)
- [Building Lambda functions with Go \(p. 453\)](#)
- [Building Lambda functions with C# \(p. 487\)](#)
- [Building Lambda functions with PowerShell \(p. 528\)](#)

Lambda: Layer conversion failed

Error: *Lambda layer conversion failed. For advice on resolving this issue, see the Troubleshoot deployment issues in Lambda page in the Lambda User Guide.*

When you configure a Lambda function with a layer, Lambda merges the layer with your function code. If this process fails to complete, Lambda returns this error. If you encounter this error, take the following steps:

- Delete any unused files from your layer
- Delete any symbolic links in your layer
- Rename any files that have the same name as a directory in any of your function's layers

Lambda: InvalidParameterValueException or RequestEntityTooLargeException

Error: *InvalidParameterValueException: Lambda was unable to configure your environment variables because the environment variables you have provided exceeded the 4KB limit. String measured: {"A1": "uSFeY5cyPiPn7AtnX5BsM..."}*

Error: *RequestEntityTooLargeException: Request must be smaller than 5120 bytes for the UpdateFunctionConfiguration operation*

The maximum size of the variables object that is stored in the function's configuration must not exceed 4096 bytes. This includes key names, values, quotes, commas, and brackets. The total size of the HTTP request body is also limited.

```
{  
    "FunctionName": "my-function",  
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",  
    "Runtime": "nodejs18.x",  
    "Role": "arn:aws:iam::123456789012:role/lambda-role",  
    "Environment": {  
        "Variables": {  
            "BUCKET": "my-bucket",  
            "KEY": "file.txt"  
        }  
    },  
    ...  
}
```

In this example, the object is 39 characters and takes up 39 bytes when it's stored (without white space) as the string {"BUCKET": "my-bucket", "KEY": "file.txt"}. Standard ASCII characters in environment variable values use one byte each. Extended ASCII and Unicode characters can use between 2 bytes and 4 bytes per character.

Lambda: InvalidParameterValueException

Error: *InvalidParameterValueException: Lambda was unable to configure your environment variables because the environment variables you have provided contains reserved keys that are currently not supported for modification.*

Lambda reserves some environment variable keys for internal use. For example, AWS_REGION is used by the runtime to determine the current Region and cannot be overridden. Other variables, like PATH, are used by the runtime but can be extended in your function configuration. For a full list, see [Defined runtime environment variables \(p. 79\)](#).

Lambda: Concurrency and memory quotas

Error: *Specified ConcurrentExecutions for function decreases account's UnreservedConcurrentExecution below its minimum value*

Error: *'MemorySize' value failed to satisfy constraint: Member must have value less than or equal to 3008*

These errors occur when you exceed the concurrency or memory [quotas \(p. 1131\)](#) for your account. New AWS accounts have reduced concurrency and memory quotas. To resolve errors related to concurrency, you can [request a quota increase](#). You cannot request memory quota increases.

- **Concurrency:** You might get an error if you try to create a function using reserved or provisioned concurrency, or if your per-function concurrency request ([PutFunctionConcurrency \(p. 1327\)](#)) exceeds your account's concurrency quota.
- **Memory:** Errors occur if the amount of memory allocated to the function exceeds your account's memory quota.

Troubleshoot invocation issues in Lambda

When you invoke a Lambda function, Lambda validates the request and checks for scaling capacity before sending the event to your function or, for asynchronous invocation, to the event queue. Invocation errors can be caused by issues with request parameters, event structure, function settings, user permissions, resource permissions, or limits.

If you invoke your function directly, you see any invocation errors in the response from Lambda. If you invoke your function asynchronously with an event source mapping or through another service, you might find errors in logs, a dead-letter queue, or a failed-event destination. Error handling options and retry behavior vary depending on how you invoke your function and on the type of error.

For a list of error types that the `Invoke` operation can return, see [Invoke \(p. 1260\)](#).

IAM: lambda:InvokeFunction not authorized

Error: *User: arn:aws:iam::123456789012:user/developer is not authorized to perform: lambda:InvokeFunction on resource: my-function*

Your user, or the role that you assume, must have permission to invoke a function. This requirement also applies to Lambda functions and other compute resources that invoke functions. Add the AWS managed policy **AWSLambdaRole** to your user, or add a custom policy that allows the `lambda:InvokeFunction` action on the target function.

Note

Unlike other Lambda API operations, the name of the IAM action (`lambda:InvokeFunction`) doesn't match the name of the API operation (`Invoke`) for invoking a function.

For more information, see [Lambda resource access permissions \(p. 815\)](#).

Lambda: Operation cannot be performed ResourceConflictException

Error: *ResourceConflictException: The operation cannot be performed at this time. The function is currently in the following state: Pending*

When you connect a function to a virtual private cloud (VPC) at the time of creation, the function enters a Pending state while Lambda creates elastic network interfaces. During this time, you can't invoke or modify your function. If you connect your function to a VPC after creation, you can invoke it while the update is pending, but you can't modify its code or configuration.

For more information, see [Lambda function states \(p. 159\)](#).

Lambda: Function is stuck in Pending

Error: *A function is stuck in the Pending state for several minutes.*

If a function is stuck in the Pending state for more than six minutes, call one of the following API operations to unblock it:

- [UpdateFunctionCode \(p. 1367\)](#)
- [UpdateFunctionConfiguration \(p. 1377\)](#)
- [PublishVersion \(p. 1315\)](#)

Lambda cancels the pending operation and puts the function into the Failed state. You can then delete the function and recreate it, or attempt another update.

Lambda: One function is using all concurrency

Issue: *One function is using all of the available concurrency, causing other functions to be throttled.*

To divide your AWS account's available concurrency in an AWS Region into pools, use [reserved concurrency \(p. 210\)](#). Reserved concurrency ensures that a function can always scale to its assigned concurrency, and that it doesn't scale beyond its assigned concurrency.

General: Cannot invoke function with other accounts or services

Issue: *You can invoke your function directly, but it doesn't run when another service or account invokes it.*

You grant [other services \(p. 556\)](#) and accounts permission to invoke a function in the function's [resource-based policy \(p. 832\)](#). If the invoker is in another account, that user must also have [permission to invoke functions \(p. 823\)](#).

General: Function invocation is looping

Issue: *Function is invoked continuously in a loop.*

This typically occurs when your function manages resources in the same AWS service that triggers it. For example, it's possible to create a function that stores an object in an Amazon Simple Storage Service (Amazon S3) bucket that's configured with a [notification that invokes the function again \(p. 741\)](#). To stop

the function from running, on the [function configuration page \(p. 71\)](#), choose **Throttle**. Then, identify the code path or configuration error that caused the recursive invocation.

Lambda: Alias routing with provisioned concurrency

Issue: Provisioned concurrency spillover invocations during alias routing.

Lambda uses a simple probabilistic model to distribute the traffic between the two function versions. At low traffic levels, you might see a high variance between the configured and actual percentage of traffic on each version. If your function uses provisioned concurrency, you can avoid [spillover invocations \(p. 871\)](#) by configuring a higher number of provisioned concurrency instances during the time that alias routing is active.

Lambda: Cold starts with provisioned concurrency

Issue: You see cold starts after enabling provisioned concurrency.

When the number of concurrent executions on a function is less than or equal to the [configured level of provisioned concurrency \(p. 213\)](#), there shouldn't be any cold starts. To help you confirm if provisioned concurrency is operating normally, do the following:

- [Check that provisioned concurrency is enabled \(p. 213\)](#) on the function version or alias.

Note

Provisioned concurrency is not configurable on the [\\$LATEST version \(p. 115\)](#).

- Ensure that your triggers invoke the correct function version or alias. For example, if you're using Amazon API Gateway, check that API Gateway invokes the function version or alias with provisioned concurrency, not \$LATEST. To confirm that provisioned concurrency is being used, you can check the [ProvisionedConcurrencyInvocations Amazon CloudWatch metric \(p. 871\)](#). A non-zero value indicates that the function is processing invocations on initialized execution environments.
- Determine whether your function concurrency exceeds the configured level of provisioned concurrency by checking the [ProvisionedConcurrencySpilloverInvocations CloudWatch metric \(p. 871\)](#). A non-zero value indicates that all provisioned concurrency is in use and some invocation occurred with a cold start.
- Check your [invocation frequency \(p. 1131\)](#) (requests per second). Functions with provisioned concurrency have a maximum rate of 10 requests per second per provisioned concurrency. For example, a function configured with 100 provisioned concurrency can handle 1,000 requests per second. If the invocation rate exceeds 1,000 requests per second, some cold starts can occur.

Note

There is a known issue in which the first invocation on an initialized execution environment reports a non-zero **Init Duration** metric in CloudWatch Logs, even though no cold start has occurred. We're developing a fix to correct the reporting to CloudWatch Logs.

Lambda: Latency variability with provisioned concurrency

Issue: You see latency variability on the first invocation after enabling provisioned concurrency.

Depending on your function's runtime and memory configuration, it's possible to see some latency variability on the first invocation on an initialized execution environment. For example, .NET and other JIT runtimes can lazily load resources on the first invocation, leading to some latency variability (typically tens of milliseconds). This variability is more apparent on 128-MiB functions. You mitigate this by increasing the function's configured memory.

Lambda: Variability between initialization and invocation times

Issue: You see unexpected gaps between the times of your function's initialization and invocation phases.

For functions using [provisioned concurrency](#), Lambda initializes the execution environment shortly after you publish a function version and ensures that function instances are always available in advance of invocation. You can see large gaps between the times of your function's initialization and invocation.

For functions using unreserved (on-demand) concurrency, Lambda may proactively initialize a function instance, even if there's no invocation. When this happens, you can observe an unexpected time gap between your function's initialization and invocation phases. This gap can appear similar to what you would observe using provisioned concurrency.

Lambda: Cold starts with new versions

Issue: You see cold starts while deploying new versions of your function.

When you update a function alias, Lambda automatically shifts provisioned concurrency to the new version based on the weights configured on the alias.

Error: *KMSDisabledException: Lambda was unable to decrypt the environment variables because the KMS key used is disabled. Please check the function's KMS key settings.*

This error can occur if your AWS Key Management Service (AWS KMS) key is disabled, or if the grant that allows Lambda to use the key is revoked. If the grant is missing, configure the function to use a different key. Then, reassign the custom key to recreate the grant.

EFS: Function could not mount the EFS file system

Error: *EFSMountFailureException: The function could not mount the EFS file system with access point arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd.*

The mount request to the function's [file system \(p. 236\)](#) was rejected. Check the function's permissions, and confirm that its file system and access point exist and are ready for use.

EFS: Function could not connect to the EFS file system

Error: *EFSMountConnectivityException: The function couldn't connect to the Amazon EFS file system with access point arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd. Check your network configuration and try again.*

The function couldn't establish a connection to the function's [file system \(p. 236\)](#) with the NFS protocol (TCP port 2049). Check the [security group and routing configuration](#) for the VPC's subnets.

EFS: Function could not mount the EFS file system due to timeout

Error: *EFSMountTimeoutException: The function could not mount the EFS file system with access point {arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd} due to mount time out.*

The function could connect to the function's [file system \(p. 236\)](#), but the mount operation timed out. Try again after a short time and consider limiting the function's [concurrency \(p. 210\)](#) to reduce load on the file system.

Lambda: Lambda detected an IO process that was taking too long

EFSIOException: This function instance was stopped because Lambda detected an IO process that was taking too long.

A previous invocation timed out and Lambda couldn't terminate the function handler. This issue can occur when an attached file system runs out of burst credits and the baseline throughput is insufficient. To increase throughput, you can increase the size of the file system or use provisioned throughput. For more information, see [Throughput \(p. 667\)](#).

Troubleshoot execution issues in Lambda

When the Lambda runtime runs your function code, the event might be processed on an instance of the function that's been processing events for some time, or it might require a new instance to be initialized. Errors can occur during function initialization, when your handler code processes the event, or when your function returns (or fails to return) a response.

Function execution errors can be caused by issues with your code, function configuration, downstream resources, or permissions. If you invoke your function directly, you see function errors in the response from Lambda. If you invoke your function asynchronously, with an event source mapping, or through another service, you might find errors in logs, a dead-letter queue, or an on-failure destination. Error handling options and retry behavior vary depending on how you invoke your function and on the type of error.

When your function code or the Lambda runtime return an error, the status code in the response from Lambda is 200 OK. The presence of an error in the response is indicated by a header named X-Amz-Function-Error. 400 and 500-series status codes are reserved for [invocation errors \(p. 951\)](#).

Lambda: Execution takes too long

Issue: *Function execution takes too long.*

If your code takes much longer to run in Lambda than on your local machine, it may be constrained by the memory or processing power available to the function. [Configure the function with additional memory \(p. 71\)](#) to increase both memory and CPU.

Lambda: Logs or traces don't appear

Issue: *Logs don't appear in CloudWatch Logs.*

Issue: *Traces don't appear in AWS X-Ray.*

Your function needs permission to call CloudWatch Logs and X-Ray. Update its [execution role \(p. 816\)](#) to grant it permission. Add the following managed policies to enable logs and tracing.

- **AWSLambdaBasicExecutionRole**
- **AWSXRayDaemonWriteAccess**

When you add permissions to your function, update its code or configuration as well. This forces running instances of your function, which have outdated credentials, to stop and be replaced.

Note

It may take 5 to 10 minutes for logs to show up after a function invocation.

Lambda: The function returns before execution finishes

Issue: *(Node.js) Function returns before code finishes executing*

Many libraries, including the AWS SDK, operate asynchronously. When you make a network call or perform another operation that requires waiting for a response, libraries return an object called a promise that tracks the progress of the operation in the background.

To wait for the promise to resolve into a response, use the `await` keyword. This blocks your handler code from executing until the promise is resolved into an object that contains the response. If you don't need to use the data from the response in your code, you can return the promise directly to the runtime.

Some libraries don't return promises but can be wrapped in code that does. For more information, see [AWS Lambda function handler in Node.js \(p. 258\)](#).

AWS SDK: Versions and updates

Issue: *The AWS SDK included on the runtime is not the latest version*

Issue: *The AWS SDK included on the runtime updates automatically*

Runtimes for scripting languages include the AWS SDK and are periodically updated to the latest version. The current version for each runtime is listed on [runtimes page \(p. 37\)](#). To use a newer version of the AWS SDK, or to lock your functions to a specific version, you can bundle the library with your function code, or [create a Lambda layer \(p. 93\)](#). For details on creating a deployment package with dependencies, see the following topics:

Node.js

[Deploy Node.js Lambda functions with .zip file archives \(p. 263\)](#)

Python

[Deploy Python Lambda functions with .zip file archives \(p. 324\)](#)

Ruby

[Deploy Ruby Lambda functions with .zip file archives \(p. 365\)](#)

Java

[Deploy Java Lambda functions with .zip or JAR file archives \(p. 393\)](#)

Go

[Deploy Go Lambda functions with .zip file archives \(p. 460\)](#)

C#

[Deploy C# Lambda functions with .zip file archives \(p. 497\)](#)

PowerShell

[Deploy PowerShell Lambda functions with .zip file archives \(p. 530\)](#)

Python: Libraries load incorrectly

Issue: (Python) *Some libraries don't load correctly from the deployment package*

Libraries with extension modules written in C or C++ must be compiled in an environment with the same processor architecture as Lambda (Amazon Linux). For more information, see [Deploy Python Lambda functions with .zip file archives \(p. 324\)](#).

Troubleshoot networking issues in Lambda

By default, Lambda runs your functions in an internal virtual private cloud (VPC) with connectivity to AWS services and the internet. To access local network resources, you can [configure your function to connect to a VPC in your account \(p. 222\)](#). When you use this feature, you manage the function's internet access and network connectivity with Amazon Virtual Private Cloud (Amazon VPC) resources.

Network connectivity errors can result from issues with your VPC's routing configuration, security group rules, AWS Identity and Access Management (IAM) role permissions, or network address translation (NAT), or from the availability of resources such as IP addresses or network interfaces. Depending on the issue, you might see a specific error or timeout if a request can't reach its destination.

VPC: Function loses internet access or times out

Issue: *Your Lambda function loses internet access after connecting to a VPC.*

Error: *Error: connect ETIMEDOUT 176.32.98.189:443*

Error: *Error: Task timed out after 10.00 seconds*

Error: *ReadTimeoutError: Read timed out. (read timeout=15)*

When you connect a function to a VPC, all outbound requests go through the VPC. To connect to the internet, configure your VPC to send outbound traffic from the function's subnet to a NAT gateway in a public subnet. For more information and sample VPC configurations, see [Internet and service access for VPC-connected functions \(p. 228\)](#).

If some of your TCP connections are timing out, this may be due to packet fragmentation. Lambda functions cannot handle incoming fragmented TCP requests, since Lambda does not support IP fragmentation for TCP or ICMP.

VPC: Function needs access to AWS services without using the internet

Issue: *Your Lambda function needs access to AWS services without using the internet.*

To connect a function to AWS services from a private subnet with no internet access, use VPC endpoints. For a sample AWS CloudFormation template with VPC endpoints for Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB (DynamoDB), see [Sample VPC configurations \(p. 228\)](#).

VPC: Elastic network interface limit reached

Error: *ENILimitReachedException: The elastic network interface limit was reached for the function's VPC.*

When you connect a Lambda function to a VPC, Lambda creates an elastic network interface for each combination of subnet and security group attached to the function. The default service quota is 250 network interfaces per VPC. To request a quota increase, use the [Service Quotas console](#).

Troubleshoot container image issues in Lambda

Container: CodeArtifactUserException errors related to the code artifact.

Issue: *CodeArtifactUserPendingException error message*

The CodeArtifact is pending optimization. The function transitions to active state when Lambda completes the optimization. HTTP response code 409.

Issue: *CodeArtifactUserDeletedException error message*

The CodeArtifact is scheduled to be deleted. HTTP response code 409.

Issue: *CodeArtifactUserFailedException error message*

Lambda failed to optimize the code. You need to correct the code and upload it again. HTTP response code 409.

Container: ManifestKeyCustomerException errors related to the code manifest key.

Issue: *KMSAccessDeniedException error message*

You do not have permissions to access the key to decrypt the manifest. HTTP response code 502.

Issue: *TooManyRequestsException error message*

The client is being throttled. The current request rate exceeds the KMS subscription rate. HTTP response code 429.

Issue: *KMSNotFoundException error message*

Lambda cannot find the key to decrypt the manifest. HTTP response code 502.

Issue: *KMSDisabledException error message*

The key to decrypt the manifest is disabled. HTTP response code 502.

Issue: *KMSInvalidStateException error message*

The key is in a state (such as pending deletion or unavailable) such that Lambda cannot use the key to decrypt the manifest. HTTP response code 502.

Container: Error occurs on runtime InvalidEntrypoint

Issue: *You receive a Runtime.ExitError error message, or an error message with "errorType": "Runtime.InvalidEntrypoint".*

Verify that the ENTRYPPOINT to your container image includes the absolute path as the location. Also verify that the image does not contain a symlink as the ENTRYPPOINT.

Lambda: System provisioning additional capacity

Error: *"Error: We currently do not have sufficient capacity in the region you requested. Our system will be working on provisioning additional capacity."*

Retry the function invocation. If the retry fails, validate that the files required to run the function code can be read by any user. Lambda defines a default Linux user with least-privileged permissions. You need to verify that your application code does not rely on files that are restricted by other Linux users for execution.

CloudFormation: ENTRYPPOINT is being overridden with a null or empty value

Error: *You are using an AWS CloudFormation template, and your container ENTRYPPOINT is being overridden with a null or empty value.*

Review the `ImageConfig` resource in the AWS CloudFormation template. If you declare an `ImageConfig` resource in your template, you must provide non-empty values for all three of the properties.

AWS Lambda applications

An AWS Lambda application is a combination of Lambda functions, event sources, and other resources that work together to perform tasks. You can use AWS CloudFormation and other tools to collect your application's components into a single package that can be deployed and managed as one resource. Applications make your Lambda projects portable and enable you to integrate with additional developer tools, such as AWS CodePipeline, AWS CodeBuild, and the AWS Serverless Application Model command line interface (AWS SAM CLI).

The [AWS Serverless Application Repository](#) provides a collection of Lambda applications that you can deploy in your account with a few clicks. The repository includes both ready-to-use applications and samples that you can use as a starting point for your own projects. You can also submit your own projects for inclusion.

[AWS CloudFormation](#) enables you to create a template that defines your application's resources and lets you manage the application as a *stack*. You can more safely add or modify resources in your application stack. If any part of an update fails, AWS CloudFormation automatically rolls back to the previous configuration. With AWS CloudFormation parameters, you can create multiple environments for your application from the same template. [AWS SAM](#) extends AWS CloudFormation with a simplified syntax focused on Lambda application development.

The [AWS CLI](#) and [AWS SAM CLI](#) are command line tools for managing Lambda application stacks. In addition to commands for managing application stacks with the AWS CloudFormation API, the AWS CLI supports higher-level commands that simplify tasks such as uploading deployment packages and updating templates. The AWS SAM CLI provides additional functionality, including validating templates, testing locally, and integrating with CI/CD systems.

When creating an application, you can create its Git repository using either CodeCommit or an AWS CodeStar connection to GitHub. CodeCommit enables you to use the IAM console to manage SSH keys and HTTP credentials for your users. AWS CodeStar connections enables you to connect to your GitHub account. For more information about connections, see [What are connections?](#) in the *Developer Tools console User Guide*.

For more information about designing Lambda applications, see [Application design](#) in the *Lambda operator guide*.

Topics

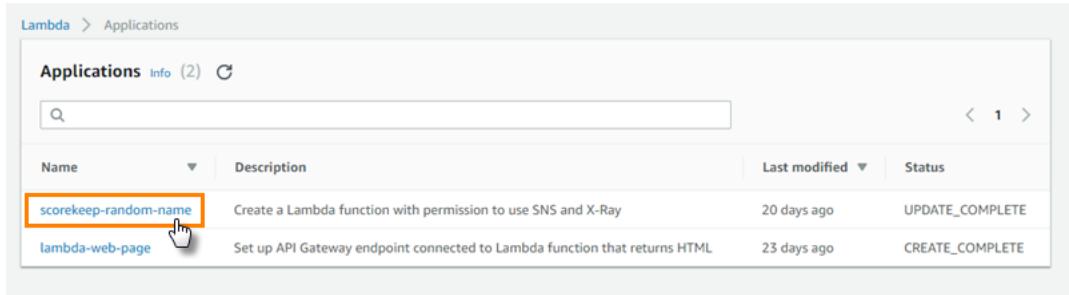
- [Managing applications in the AWS Lambda console \(p. 961\)](#)
- [Creating an application with continuous delivery in the Lambda console \(p. 964\)](#)
- [Rolling deployments for Lambda functions \(p. 973\)](#)
- [Invoking Lambda functions with the AWS Mobile SDK for Android \(p. 975\)](#)

Managing applications in the AWS Lambda console

The AWS Lambda console helps you monitor and manage your [Lambda applications \(p. 960\)](#). The **Applications** menu lists AWS CloudFormation stacks with Lambda functions. The menu includes stacks that you launch in AWS CloudFormation by using the AWS CloudFormation console, the AWS Serverless Application Repository, the AWS CLI, or the AWS SAM CLI.

To view a Lambda application

1. Open the Lambda console [Applications page](#).
2. Choose an application.



The overview shows the following information about your application.

- **AWS CloudFormation template or SAM template** – The template that defines your application.
- **Resources** – The AWS resources that are defined in your application's template. To manage your application's Lambda functions, choose a function name from the list.

Monitoring applications

The **Monitoring** tab shows an Amazon CloudWatch dashboard with aggregate metrics for the resources in your application.

To monitor a Lambda application

1. Open the Lambda console [Applications page](#).
2. Choose **Monitoring**.

By default, the Lambda console shows a basic dashboard. You can customize this page by defining custom dashboards in your application template. When your template includes one or more dashboards, the page shows your dashboards instead of the default dashboard. You can switch between dashboards with the drop-down menu on the top right of the page.

Custom monitoring dashboards

Customize your application monitoring page by adding one or more Amazon CloudWatch dashboards to your application template with the [AWS::CloudWatch::Dashboard](#) resource type. The following example creates a dashboard with a single widget that graphs the number of invocations of a function named `my-function`.

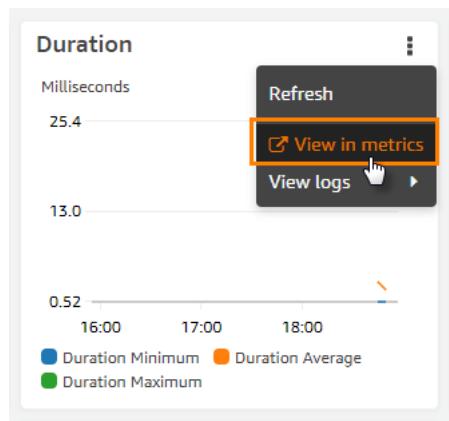
Example function dashboard template

```
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
              }
            }
          ]
        }
```

You can get the definition for any of the widgets in the default monitoring dashboard from the CloudWatch console.

To view a widget definition

1. Open the Lambda console [Applications page](#).
2. Choose an application that has the standard dashboard.
3. Choose **Monitoring**.
4. On any widget, choose **View in metrics** from the drop-down menu.



5. Choose **Source**.

For more information about authoring CloudWatch dashboards and widgets, see [Dashboard body structure and syntax](#) in the *Amazon CloudWatch API Reference*.

Creating an application with continuous delivery in the Lambda console

You can use the Lambda console to create an application with an integrated continuous delivery pipeline. With continuous delivery, every change that you push to your source control repository triggers a pipeline that builds and deploys your application automatically. The Lambda console provides starter projects for common application types with Node.js sample code and templates that create supporting resources.

In this tutorial, you create the following resources.

- **Application** – A Node.js Lambda function, build specification, and AWS Serverless Application Model (AWS SAM) template.
- **Pipeline** – An AWS CodePipeline pipeline that connects the other resources to enable continuous delivery.
- **Repository** – A Git repository in AWS CodeCommit. When you push a change, the pipeline copies the source code into an Amazon S3 bucket and passes it to the build project.
- **Trigger** – An Amazon EventBridge (CloudWatch Events) rule that watches the main branch of the repository and triggers the pipeline.
- **Build project** – An AWS CodeBuild build that gets the source code from the pipeline and packages the application. The source includes a build specification with commands that install dependencies and prepare the application template for deployment.
- **Deployment configuration** – The pipeline's deployment stage defines a set of actions that take the processed AWS SAM template from the build output, and deploy the new version with AWS CloudFormation.
- **Bucket** – An Amazon Simple Storage Service (Amazon S3) bucket for deployment artifact storage.
- **Roles** – The pipeline's source, build, and deploy stages have IAM roles that allow them to manage AWS resources. The application's function has an [execution role \(p. 816\)](#) that allows it to upload logs and can be extended to access other services.

Your application and pipeline resources are defined in AWS CloudFormation templates that you can customize and extend. Your application repository includes a template that you can modify to add Amazon DynamoDB tables, an Amazon API Gateway API, and other application resources. The continuous delivery pipeline is defined in a separate template outside of source control and has its own stack.

The pipeline maps a single branch in a repository to a single application stack. You can create additional pipelines to add environments for other branches in the same repository. You can also add stages to your pipeline for testing, staging, and manual approvals. For more information about AWS CodePipeline, see [What is AWS CodePipeline](#).

Sections

- [Prerequisites \(p. 965\)](#)
- [Create an application \(p. 965\)](#)
- [Invoke the function \(p. 966\)](#)
- [Add an AWS resource \(p. 967\)](#)
- [Update the permissions boundary \(p. 969\)](#)
- [Update the function code \(p. 969\)](#)
- [Next steps \(p. 970\)](#)
- [Troubleshooting \(p. 971\)](#)

- [Clean up \(p. 972\)](#)

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console \(p. 4\)](#) to create your first Lambda function.

To complete the following steps, you need the [AWS Command Line Interface \(AWS CLI\) version 2](#). Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.0.57 Python/3.7.4 Darwin/19.6.0 exe/x86_64
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#). Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

This tutorial uses CodeCommit for source control. To set up your local machine to access and update application code, see [Setting up](#) in the *AWS CodeCommit User Guide*.

Create an application

Create an application in the Lambda console. In Lambda, an application is an AWS CloudFormation stack with a Lambda function and any number of supporting resources. In this tutorial, you create an application that has a function and its execution role.

To create an application

1. Open the Lambda console [Applications page](#).
2. Choose **Create application**.
3. Choose **Author from scratch**.
4. Configure application settings.
 - **Application name** – **my-app**.
 - **Runtime** – **Node.js 14.x**.
 - **Source control service** – **CodeCommit**.
 - **Repository name** – **my-app-repo**.
 - **Permissions** – **Create roles and permissions boundary**.
5. Choose **Create**.

Lambda creates the pipeline and related resources and commits the sample application code to the Git repository. As resources are created, they appear on the overview page.

Logical ID	Physical ID	Type	Last modified
CodeCommitRepo	b4976f9b-9399-45f1-bd21-e7a9a315981d	CodeCommit Repository	9 seconds ago
PermissionsBoundaryPolicy	arn:aws:iam::123456789012:policy/my-app-us-east-2-PermissionsBoundary	IAM ManagedPolicy	13 seconds ago
S3Bucket	aws-us-east-2-123456789012-my-app-pipe	S3 Bucket	13 seconds ago

The **Infrastructure** stack contains the repository, build project, and other resources that combine to form a continuous delivery pipeline. When this stack finishes deploying, it in turn deploys the application stack that contains the function and execution role. These are the application resources that appear under **Resources**.

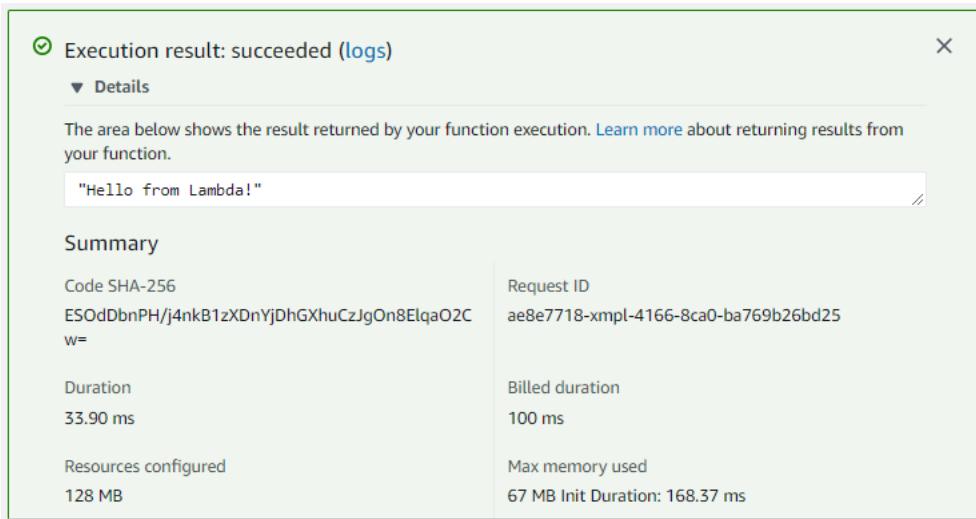
Invoke the function

When the deployment process completes, invoke the function from the Lambda console.

To invoke the application's function

1. Open the Lambda console [Applications page](#).
2. Choose **my-app**.
3. Under **Resources**, choose **helloFromLambdaFunction**.
4. Choose **Test**.
5. Configure a test event.
 - **Event name – event**
 - **Body – {}**
6. Choose **Create**.
7. Choose **Test**.

The Lambda console runs your function and displays the result. Expand the **Details** section under the result to see the output and execution details.



Add an AWS resource

In the previous step, Lambda console created a Git repository that contains function code, a template, and a build specification. You can add resources to your application by modifying the template and pushing changes to the repository. To get a copy of the application on your local machine, clone the repository.

To clone the project repository

1. Open the Lambda console [Applications page](#).
2. Choose **my-app**.
3. Choose **Code**.
4. Under **Repository details**, copy the HTTP or SSH repository URI, depending on the authentication mode that you configured during [setup \(p. 965\)](#).
5. To clone the repository, use the `git clone` command.

```
git clone ssh://git-codecommit.us-east-2.amazonaws.com/v1/repos/my-app-repo
```

To add a DynamoDB table to the application, define an `AWS::Serverless::SimpleTable` resource in the template.

To add a DynamoDB table

1. Open `template.yml` in a text editor.
2. Add a table resource, an environment variable that passes the table name to the function, and a permissions policy that allows the function to manage it.

Example `template.yml` - resources

```
...
Resources:
  ddbTable:
    Type: AWS::Serverless::SimpleTable
    Properties:
```

```

PrimaryKey:
  Name: id
  Type: String
ProvisionedThroughput:
  ReadCapacityUnits: 1
  WriteCapacityUnits: 1
helloFromLambdaFunction:
  Type: AWS::Serverless::Function
Properties:
  CodeUri: ./src/handlers/hello-from-lambda.helloFromLambdaHandler
  Handler: src/handlers/hello-from-lambda.helloFromLambdaHandler
  Runtime: nodejs14.x
  MemorySize: 128
  Timeout: 60
  Description: A Lambda function that returns a static string.
Environment:
  Variables:
    DDB_TABLE: !Ref ddbTable
Policies:
  - DynamoDBCrudPolicy:
      TableName: !Ref ddbTable
  - AWSLambdaBasicExecutionRole

```

3. Commit and push the change.

```
git commit -am "Add DynamoDB table"
git push
```

When you push a change, it triggers the application's pipeline. Use the **Deployments** tab of the application screen to track the change as it flows through the pipeline. When the deployment is complete, proceed to the next step.

Deployment	Resource type	Last updated time	Status
6 minutes ago	Lambda application	5 minutes ago	✓ Update complete
2 hours ago	Lambda application	2 hours ago	✓ Create complete

Update the permissions boundary

The sample application applies a *permissions boundary* to its function's execution role. The permissions boundary limits the permissions that you can add to the function's role. Without the boundary, users with write access to the project repository could modify the project template to give the function permission to access resources and services outside of the scope of the sample application.

In order for the function to use the DynamoDB permission that you added to its execution role in the previous step, you must extend the permissions boundary to allow the additional permissions. The Lambda console detects resources that aren't in the permissions boundary and provides an updated policy that you can use to update it.

To update the application's permissions boundary

1. Open the Lambda console [Applications page](#).
2. Choose your application.
3. Under **Resources**, choose **Edit permissions boundary**.
4. Follow the instructions shown to update the boundary to allow access to the new table.

For more information about permissions boundaries, see [Using permissions boundaries for AWS Lambda applications \(p. 845\)](#).

Update the function code

Next, update the function code to use the table. The following code uses the DynamoDB table to track the number of invocations processed by each instance of the function. It uses the log stream ID as a unique identifier for the function instance.

To update the function code

1. Add a new handler named `index.js` to the `src/handlers` folder with the following content.

Example `src/handlers/index.js`

```
const dynamodb = require('aws-sdk/clients/dynamodb');
const docClient = new dynamodb.DocumentClient();

exports.handler = async (event, context) => {
    const message = 'Hello from Lambda!';
    const tableName = process.env.DDB_TABLE;
    const logStreamName = context.logStreamName;
    var params = {
        TableName : tableName,
        Key: { id : logStreamName },
        UpdateExpression: 'set invocations = if_not_exists(invocations, :start)
+ :inc',
        ExpressionAttributeValues: {
            ':start': 0,
            ':inc': 1
        },
        ReturnValues: 'ALL_NEW'
    };
    await docClient.update(params).promise();

    const response = {
        body: JSON.stringify(message)
    };
    console.log(`body: ${response.body}`);
    return response;
}
```

}

2. Open the application template and change the handler value to `src/handlers/index.handler`.

Example template.yml

```
...
helloFromLambdaFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: ./
    Handler: src/handlers/index.handler
    Runtime: nodejs14.x
```

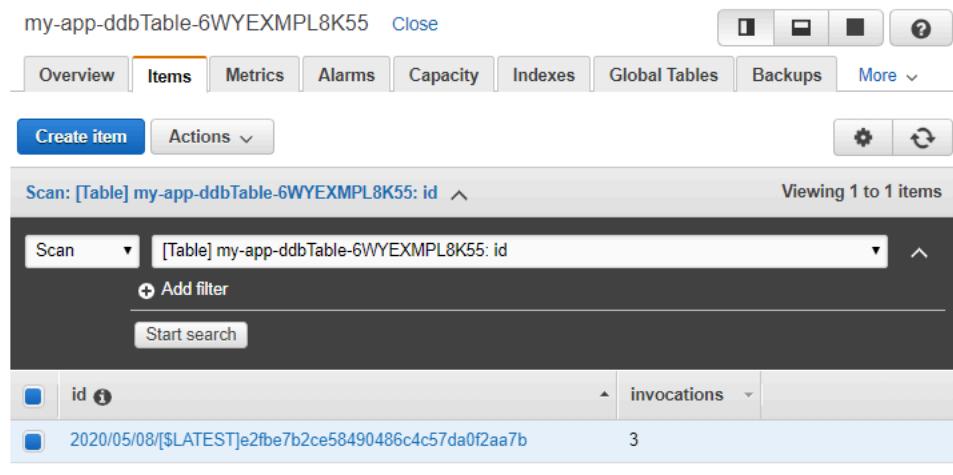
3. Commit and push the change.

```
git add . && git commit -m "Use DynamoDB table"
git push
```

After the code change is deployed, invoke the function a few times to update the DynamoDB table.

To view the DynamoDB table

1. Open the [Tables page of the DynamoDB console](#).
2. Choose the table that starts with **my-app**.
3. Choose **Items**.
4. Choose **Start search**.



Lambda creates additional instances of your function to handle multiple concurrent invocations. Each log stream in the CloudWatch Logs log group corresponds to a function instance. A new function instance is also created when you change your function's code or configuration. For more information on scaling, see [Lambda function scaling \(p. 197\)](#).

Next steps

The AWS CloudFormation template that defines your application resources uses the AWS Serverless Application Model transform to simplify the syntax for resource definitions, and automate uploading

the deployment package and other artifacts. AWS SAM also provides a command line interface (the AWS SAM CLI), which has the same packaging and deployment functionality as the AWS CLI, with additional features specific to Lambda applications. Use the AWS SAM CLI to test your application locally in a Docker container that emulates the Lambda execution environment.

- [Installing the AWS SAM CLI](#)
- [Testing and debugging serverless applications with AWS SAM](#)
- [Deploying serverless applications using CI/CD systems with AWS SAM](#)

AWS Cloud9 provides an online development environment that includes Node.js, the AWS SAM CLI, and Docker. With AWS Cloud9, you can start developing quickly and access your development environment from any computer. For instructions, see [Getting started](#) in the *AWS Cloud9 User Guide*.

For local development, AWS toolkits for integrated development environments (IDEs) let you test and debug functions before pushing them to your repository.

- [AWS Toolkit for JetBrains](#) – Plugin for PyCharm (Python) and IntelliJ (Java) IDEs.
- [AWS Toolkit for Eclipse](#) – Plugin for Eclipse IDE (multiple languages).
- [AWS Toolkit for Visual Studio Code](#) – Plugin for Visual Studio Code IDE (multiple languages).
- [AWS Toolkit for Visual Studio](#) – Plugin for Visual Studio IDE (multiple languages).

Troubleshooting

As you develop your application, you will likely encounter the following types of errors.

- **Build errors** – Issues that occur during the build phase, including compilation, test, and packaging errors.
- **Deployment errors** – Issues that occur when AWS CloudFormation isn't able to update the application stack. These include permissions errors, account quotas, service issues, or template errors.
- **Invocation errors** – Errors that are returned by a function's code or runtime.

For build and deployment errors, you can identify the cause of an error in the Lambda console.

To troubleshoot application errors

1. Open the Lambda console [Applications page](#).
2. Choose an application.
3. Choose **Deployments**.
4. To view the application's pipeline, choose **Deployment pipeline**.
5. Identify the action that encountered an error.
6. To view the error in context, choose **Details**.

For deployment errors that occur during the **ExecuteChangeSet** action, the pipeline links to a list of stack events in the AWS CloudFormation console. Search for an event with the status **UPDATE_FAILED**. Because AWS CloudFormation rolls back after an error, the relevant event is under several other events in the list. If AWS CloudFormation could not create a change set, the error appears under **Change sets** instead of under **Events**.

A common cause of deployment and invocation errors is a lack of permissions in one or more roles. The pipeline has a role for deployments (`CloudFormationRole`) that's equivalent to the [user permissions \(p. 823\)](#) that you would use to update an AWS CloudFormation stack directly. If you add

resources to your application or enable Lambda features that require user permissions, the deployment role is used. You can find a link to the deployment role under **Infrastructure** in the application overview.

If your function accesses other AWS services or resources, or if you enable features that require the function to have additional permissions, the function's [execution role \(p. 816\)](#) is used. All execution roles that are created in your application template are also subject to the application's permissions boundary. This boundary requires you to explicitly grant access to additional services and resources in IAM after adding permissions to the execution role in the template.

For example, to [connect a function to a virtual private cloud \(p. 222\)](#) (VPC), you need user permissions to describe VPC resources. The execution role needs permission to manage network interfaces. This requires the following steps.

1. Add the required user permissions to the deployment role in IAM.
2. Add the execution role permissions to the permissions boundary in IAM.
3. Add the execution role permissions to the execution role in the application template.
4. Commit and push to deploy the updated execution role.

After you address permissions errors, choose **Release change** in the pipeline overview to rerun the build and deployment.

Clean up

You can continue to modify and use the sample to develop your own application. If you are done using the sample, delete the application to avoid paying for the pipeline, repository, and storage.

To delete the application

1. Open the [AWS CloudFormation console](#).
2. Delete the application stack – **my-app**.
3. Open the [Amazon S3 console](#).
4. Delete the artifact bucket – **us-east-2-123456789012-my-app-pipe**.
5. Return to the AWS CloudFormation console and delete the infrastructure stack – **serverlessrepo-my-app-toolchain**.

Function logs are not associated with the application or infrastructure stack in AWS CloudFormation. Delete the log group separately in the CloudWatch Logs console.

To delete the log group

1. Open the [Log groups page](#) of the Amazon CloudWatch console.
2. Choose the function's log group (`/aws/lambda/my-app-helloFromLambdaFunction-YV1VXmplk7QK`).
3. Choose **Actions**, and then choose **Delete log group**.
4. Choose **Yes, Delete**.

Rolling deployments for Lambda functions

Use rolling deployments to control the risks associated with introducing new versions of your Lambda function. In a rolling deployment, the system automatically deploys the new version of the function and gradually sends an increasing amount of traffic to the new version. The amount of traffic and rate of increase are parameters that you can configure.

You configure a rolling deployment by using AWS CodeDeploy and AWS SAM. CodeDeploy is a service that automates application deployments to Amazon computing platforms such as Amazon EC2 and AWS Lambda. For more information, see [What is CodeDeploy?](#). By using CodeDeploy to deploy your Lambda function, you can easily monitor the status of the deployment and initiate a rollback if you detect any issues.

AWS SAM is an open-source framework for building serverless applications. You create an AWS SAM template (in YAML format) to specify the configuration of the components required for the rolling deployment. AWS SAM uses the template to create and configure the components. For more information, see [What is the AWS SAM?](#).

In a rolling deployment, AWS SAM performs these tasks:

- It configures your Lambda function and creates an alias.

The alias routing configuration is the underlying capability that implements the rolling deployment.

- It creates a CodeDeploy application and deployment group.

The deployment group manages the rolling deployment and the rollback (if needed).

- It detects when you create a new version of your Lambda function.
- It triggers CodeDeploy to start the deployment of the new version.

Example AWS SAM Lambda template

The following example shows an [AWS SAM template](#) for a simple rolling deployment.

```
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: nodejs12.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
```

This template defines a Lambda function named `myDateTimeFunction` with the following properties.

AutoPublishAlias

The AutoPublishAlias property creates an alias named live. In addition, the AWS SAM framework automatically detects when you save new code for the function. The framework then publishes a new function version and updates the live alias to point to the new version.

DeploymentPreference

The DeploymentPreference property determines the rate at which the CodeDeploy application shifts traffic from the original version of the Lambda function to the new version. The value Linear10PercentEvery2Minutes shifts an additional ten percent of the traffic to the new version every two minutes.

For a list of the predefined deployment configurations, see [Deployment configurations](#).

For a detailed tutorial on how to use CodeDeploy with Lambda functions, see [Deploy an updated Lambda function with CodeDeploy](#).

Invoking Lambda functions with the AWS Mobile SDK for Android

You can call a Lambda function from a mobile application. Put business logic in functions to separate its development lifecycle from that of front-end clients, making mobile applications less complex to develop and maintain. With the Mobile SDK for Android, you [use Amazon Cognito to authenticate users and authorize requests \(p. 975\)](#).

When you invoke a function from a mobile application, you choose the event structure, [invocation type \(p. 118\)](#), and permission model. You can use [aliases \(p. 85\)](#) to enable seamless updates to your function code, but otherwise the function and application are tightly coupled. As you add more functions, you can create an API layer to decouple your function code from your front-end clients and improve performance.

To create a fully-featured web API for your mobile and web applications, use Amazon API Gateway. With API Gateway, you can add custom authorizers, throttle requests, and cache results for all of your functions. For more information, see [Using AWS Lambda with Amazon API Gateway \(p. 562\)](#).

Topics

- [Tutorial: Using AWS Lambda with the Mobile SDK for Android \(p. 975\)](#)
- [Sample function code \(p. 981\)](#)

Tutorial: Using AWS Lambda with the Mobile SDK for Android

In this tutorial, you create a simple Android mobile application that uses Amazon Cognito to get credentials and invokes a Lambda function.

The mobile application retrieves AWS credentials from an Amazon Cognito identity pool and uses them to invoke a Lambda function with an event that contains request data. The function processes the request and returns a response to the front-end.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in [Create a Lambda function with the console \(p. 4\)](#) to create your first Lambda function.

To complete the following steps, you need the [AWS Command Line Interface \(AWS CLI\) version 2](#). Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.0.57 Python/3.7.4 Darwin/19.6.0 exe/x86_64
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

Note

In Windows, some Bash CLI commands that you commonly use with Lambda (such as zip) are not supported by the operating system's built-in terminals. To get a Windows-integrated

version of Ubuntu and Bash, [install the Windows Subsystem for Linux](#). Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

Create the execution role

Create the [execution role \(p. 816\)](#) that gives your function permission to access AWS resources.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
 - **Trusted entity – AWS Lambda**.
 - **Permissions – AWSLambdaBasicExecutionRole**.
 - **Role name – lambda-android-role**.

The **AWSLambdaBasicExecutionRole** policy has the permissions that the function needs to write logs to CloudWatch Logs.

Create the function

The following example uses data to generate a string response.

Note

For sample code in other languages, see [Sample function code \(p. 981\)](#).

Example index.js

```
exports.handler = function(event, context, callback) {
  console.log("Received event: ", event);
  var data = {
    "greetings": "Hello, " + event.firstName + " " + event.lastName + "."
  };
  callback(null, data);
}
```

To create the function

1. Copy the sample code into a file named `index.js`.
2. Create a deployment package.

```
zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command.

```
aws lambda create-function --function-name AndroidBackendLambdaFunction \
--zip-file file://function.zip --handler index.handler --runtime nodejs12.x \
--role arn:aws:iam::123456789012:role/lambda-android-role
```

Test the Lambda function

Invoke the function manually using the sample event data.

To test the Lambda function (AWS CLI)

1. Save the following sample event JSON in a file, `input.txt`.

```
{ "firstName": "first-name", "lastName": "last-name" }
```

2. Run the following `invoke` command:

```
aws lambda invoke --function-name AndroidBackendLambdaFunction \
--payload file://file-path/input.txt outfile.txt
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

Create an Amazon Cognito identity pool

In this section, you create an Amazon Cognito identity pool. The identity pool has two IAM roles. You update the IAM role for unauthenticated users and grant permissions to run the `AndroidBackendLambdaFunction` Lambda function.

For more information about IAM roles, see [IAM roles](#) in the *IAM User Guide*. For more information about Amazon Cognito services, see the [Amazon Cognito](#) product detail page.

To create an identity pool

1. Open the [Amazon Cognito console](#).
2. Create a new identity pool called `JavaFunctionAndroidEventHandlerPool`. Before you follow the procedure to create an identity pool, note the following:
 - The identity pool you are creating must allow access to unauthenticated identities because our example mobile application does not require a user log in. Therefore, make sure to select the **Enable access to unauthenticated identities** option.
 - Add the following statement to the permission policy associated with the unauthenticated identities.

```
{
    "Effect": "Allow",
    "Action": [
        "lambda:InvokeFunction"
    ],
    "Resource": [
        "arn:aws:lambda:us-
east-1:123456789012:function:AndroidBackendLambdaFunction"
    ]
}
```

The resulting policy will be as follows:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "mobileanalytics:PutEvents",
                "cognito-sync:*"
            ]
        }
    ]
}
```

```
        ],
        "Resource":[
            "*"
        ]
    },
{
    "Effect":"Allow",
    "Action":[
        "lambda:invokefunction"
    ],
    "Resource":[
        "arn:aws:lambda:us-east-1:account-id:function:AndroidBackendLambdaFunction"
    ]
}
]
```

For instructions about how to create an identity pool, log in to the [Amazon Cognito console](#) and follow the **New Identity Pool** wizard.

3. Note the identity pool ID. You specify this ID in your mobile application you create in the next section. The app uses this ID when it sends request to Amazon Cognito to request for temporary security credentials.

Create an Android application

Create a simple Android mobile application that generates events and invokes Lambda functions by passing the event data as parameters.

The following instructions have been verified using Android studio.

1. Create a new Android project called **AndroidEventGenerator** using the following configuration:
 - Select the **Phone and Tablet** platform.
 - Choose **Blank Activity**.
2. In the build.gradle (Module:app) file, add the following in the dependencies section:

```
compile 'com.amazonaws:aws-android-sdk-core:2.2.+'
compile 'com.amazonaws:aws-android-sdk-lambda:2.2.+'
```

3. Build the project so that the required dependencies are downloaded, as needed.
4. In the Android application manifest (**AndroidManifest.xml**), add the following permissions so that your application can connect to the Internet. You can add them just before the `</manifest>` end tag.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

5. In **MainActivity**, add the following imports:

```
import com.amazonaws.mobileconnectors.lambdainvoker.*;
import com.amazonaws.auth.CognitoCachingCredentialsProvider;
import com.amazonaws.regions.Regions;
```

6. In the package section, add the following two classes (**RequestClass** and **ResponseClass**). Note that the POJO is same as the POJO you created in your Lambda function in the preceding section.

- **RequestClass.** The instances of this class act as the POJO (Plain Old Java Object) for event data which consists of first and last name. If you are using Java example for your Lambda function you created in the preceding section, this POJO is same as the POJO you created in your Lambda function code.

```
package com.example....lambdaeventgenerator;
public class RequestClass {
    String firstName;
    String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public RequestClass(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public RequestClass() {
    }
}
```

- **ResponseClass**

```
package com.example....lambdaeventgenerator;
public class ResponseClass {
    String greetings;

    public String getGreetings() {
        return greetings;
    }

    public void setGreetings(String greetings) {
        this.greetings = greetings;
    }

    public ResponseClass(String greetings) {
        this.greetings = greetings;
    }

    public ResponseClass() {
    }
}
```

7. In the same package, create interface called MyInterface for invoking the AndroidBackendLambdaFunction Lambda function.

```
package com.example....lambdaeventgenerator;
import com.amazonaws.mobileconnectors.lambdainvoker.LambdaFunction;
```

```

public interface MyInterface {

    /**
     * Invoke the Lambda function "AndroidBackendLambdaFunction".
     * The function name is the method name.
     */
    @LambdaFunction
    ResponseClass AndroidBackendLambdaFunction(RequestClass request);

}

```

The `@LambdaFunction` annotation in the code maps the specific client method to the same-name Lambda function.

8. To keep the application simple, we are going to add code to invoke the Lambda function in the `onCreate()` event handler. In `MainActivity`, add the following code toward the end of the `onCreate()` code.

```

// Create an instance of CognitoCachingCredentialsProvider
CognitoCachingCredentialsProvider cognitoProvider = new
    CognitoCachingCredentialsProvider(
        this.getApplicationContext(), "identity-pool-id", Regions.US_WEST_2);

// Create LambdaInvokerFactory, to be used to instantiate the Lambda proxy.
LambdaInvokerFactory factory = new LambdaInvokerFactory(this.getApplicationContext(),
    Regions.US_WEST_2, cognitoProvider);

// Create the Lambda proxy object with a default Json data binder.
// You can provide your own data binder by implementing
// LambdaDataBinder.
final MyInterface myInterface = factory.build(MyInterface.class);

RequestClass request = new RequestClass("John", "Doe");
// The Lambda function invocation results in a network call.
// Make sure it is not called from the main thread.
new AsyncTask<RequestClass, Void, ResponseClass>() {
    @Override
    protected ResponseClass doInBackground(RequestClass... params) {
        // invoke "echo" method. In case it fails, it will throw a
        // LambdaFunctionException.
        try {
            return myInterface.AndroidBackendLambdaFunction(params[0]);
        } catch (LambdaFunctionException lfe) {
            Log.e("Tag", "Failed to invoke echo", lfe);
            return null;
        }
    }

    @Override
    protected void onPostExecute(ResponseClass result) {
        if (result == null) {
            return;
        }

        // Do a toast
        Toast.makeText(MainActivity.this, result.getGreetings(),
            Toast.LENGTH_LONG).show();
    }
}.execute(request);

```

9. Run the code and verify it as follows:
 - The `Toast.makeText()` displays the response returned.

- Verify that CloudWatch Logs shows the log created by the Lambda function. It should show the event data (first name and last name). You can also verify this in the AWS Lambda console.

Sample function code

Sample code is available for the following languages.

Topics

- [Node.js \(p. 981\)](#)
- [Java \(p. 981\)](#)

Node.js

The following example uses data to generate a string response.

Example index.js

```
exports.handler = function(event, context, callback) {
    console.log("Received event: ", event);
    var data = {
        "greetings": "Hello, " + event.firstName + " " + event.lastName + "."
    };
    callback(null, data);
}
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Node.js Lambda functions with .zip file archives \(p. 263\)](#).

Java

The following example uses data to generate a string response.

In the code, the handler (`myHandler`) uses the `RequestClass` and `ResponseClass` types for the input and output. The code provides implementation for these types.

Example HelloPojo.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;

public class HelloPojo {

    // Define two classes/POJOs for use with Lambda function.
    public static class RequestClass {
        String firstName;
        String lastName;

        public String getFirstName() {
            return firstName;
        }

        public void setFirstName(String firstName) {
            this.firstName = firstName;
        }
    }
}
```

```
public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public RequestClass(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

public RequestClass() {
}

public static class ResponseClass {
    String greetings;

    public String getGreetings() {
        return greetings;
    }

    public void setGreetings(String greetings) {
        this.greetings = greetings;
    }

    public ResponseClass(String greetings) {
        this.greetings = greetings;
    }

    public ResponseClass() {
    }
}

public static ResponseClass myHandler(RequestClass request, Context context){
    String greetingString = String.format("Hello %s, %s.", request.firstName,
request.lastName);
    context.getLogger().log(greetingString);
    return new ResponseClass(greetingString);
}
}
```

Dependencies

- [aws-lambda-java-core](#)

Build the code with the Lambda library dependencies to create a deployment package. For instructions, see [Deploy Java Lambda functions with .zip or JAR file archives \(p. 393\)](#).

Orchestrating functions with Step Functions

AWS Step Functions is an orchestration service that lets you connect Lambda functions together into serverless workflows, called state machines. Use Step Functions to orchestrate serverless applications workflows (for example, a store checkout process), build long-running workflows for IT automation and human-approval use cases, or create high-volume short-duration workflows for streaming data processing and ingestion.

Topics

- [State machine application patterns \(p. 983\)](#)
- [Managing state machines in the Lambda console \(p. 986\)](#)
- [Orchestration examples with Step Functions \(p. 988\)](#)

State machine application patterns

In Step Functions, you orchestrate your resources using state machines, which are defined using a JSON-based, structured language called [Amazon States Language](#).

Sections

- [State machine components \(p. 983\)](#)
- [State machine application patterns \(p. 983\)](#)
- [Applying patterns to state machines \(p. 984\)](#)
- [Example branching application pattern \(p. 984\)](#)

State machine components

State machines contain elements called [states](#) that make up your workflow. The logic of each state determines which state comes next, what data to pass along, and when to terminate the workflow. A state is referred to by its name, which can be any string, but which must be unique within the scope of the entire state machine.

To create a state machine that uses Lambda, you need the following components:

1. An AWS Identity and Access Management (IAM) role for Lambda with one or more permissions policies (such as [AWSLambdaRole](#) service permissions).
2. One or more Lambda functions (with the IAM role attached) for your specific runtime.
3. A state machine authored in Amazon States Language.

State machine application patterns

You can create complex orchestrations for state machines using application patterns such as:

- **Catch and retry** – Handle errors using sophisticated catch-and-retry functionality.

- **Branching** – Design your workflow to choose different branches based on Lambda function output.
- **Chaining** – Connect functions into a series of steps, with the output of one step providing the input to the next step.
- **Parallelism** – Run functions in parallel, or use dynamic parallelism to invoke a function for every member of any array.

Applying patterns to state machines

The following shows how you can apply these application patterns to a state machine within an Amazon States Language definition.

Catch and Retry

A Catch field and a Retry field add catch-and-retry logic to a state machine. [Catch](#) ("Type": "Catch") is an array of objects that define a fallback state. [Retry](#) ("Type": "Retry") is an array of objects that define a retry policy if the state encounters runtime errors.

Branching

A Choice state adds branching logic to a state machine. [Choice](#) ("Type": "Choice") is an array of rules that determine which state the state machine transitions to next.

Chaining

A "Chaining" pattern describes multiple Lambda functions connected together in a state machine. You can use chaining to create reusable workflow invocations from a [Task](#) ("Type": "Task") state of a state machine.

Parallelism

A Parallel state adds parallelism logic to a state machine. You can use a [Parallel](#) state ("Type": "Parallel") to create parallel branches of invocation in your state machine.

Dynamic parallelism

A Map state adds dynamic "for-each" loop logic to a state machine. You can use a [Map](#) state ("Type": "Map") to run a set of steps for each element of an input array in a state machine. While the Parallel state invokes multiple branches of steps using the same input, a Map state invokes the same steps for multiple entries of the array.

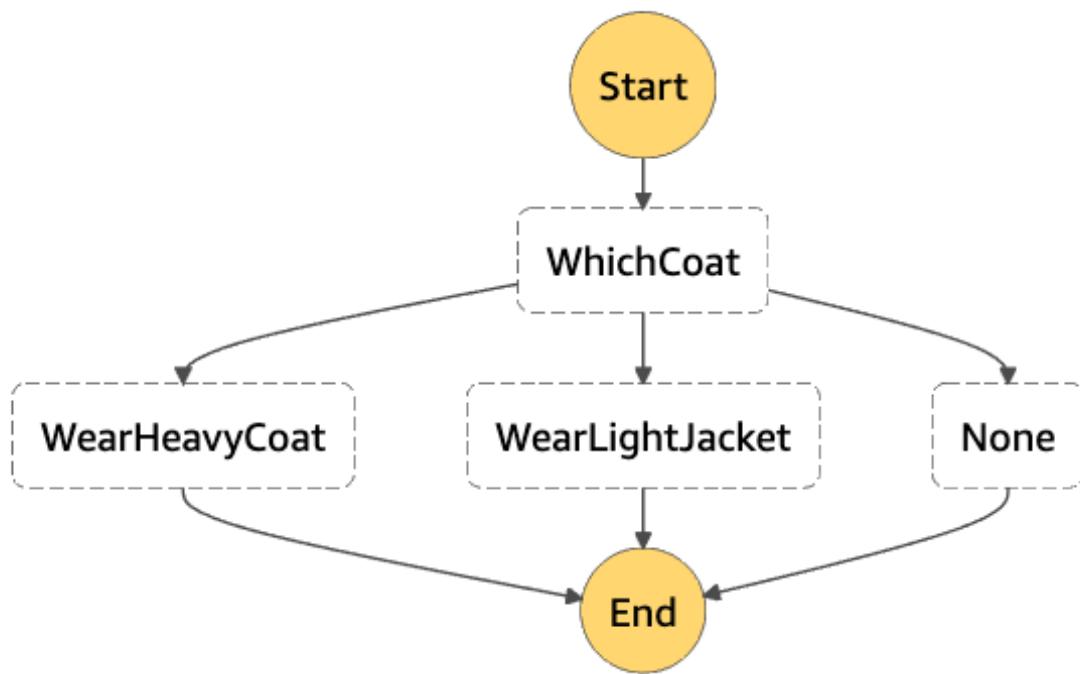
In addition to application patterns, Step Functions supports various [service integration patterns](#), including the ability to pause a workflow for human approval, or to call a legacy system or other third party.

Example branching application pattern

In the following example, the WhichCoat state machine defined in the Amazon States Language (ASL) definition shows a branching application pattern with a [Choice](#) state ("Type": "Choice"). If the condition of one of the three Choice states is met, the Lambda function is invoked as a [Task](#):

1. The WearHeavyCoat state invokes the wear_heavy_coat Lambda function and returns a message.
2. The WearLightJacket state invokes the wear_light_jacket Lambda function and returns a message.
3. The None state invokes the no_jacket Lambda function and returns a message.

The WhichCoat state machine has the following structure:



Example Example Amazon States Language definition

The following Amazon States Language definition of the WhichCoat state machine uses a Variable [context object](#) called Weather. If one of the three conditions in StringEquals is met, the Lambda function defined in the [Resource field's Amazon Resource Name \(ARN\)](#) is invoked.

```
{  
    "Comment": "Coat Indicator State Machine",  
    "StartAt": "WhichCoat",  
    "States": {  
        "WhichCoat": {  
            "Type": "Choice",  
            "Choices": [  
                {  
                    "Variable": "$.Weather",  
                    "StringEquals": "FREEZING",  
                    "Next": "WearHeavyCoat"  
                },  
                {  
                    "Variable": "$.Weather",  
                    "StringEquals": "COOL",  
                    "Next": "WearLightJacket"  
                },  
                {  
                    "Variable": "$.Weather",  
                    "StringEquals": "WARM",  
                    "Next": "None"  
                }  
            ]  
        },  
        "WearHeavyCoat": {  
            "Type": "Task",  
            "Resource": "arn:aws:lambda:us-west-2:01234567890:function:wear_heavy_coat",  
            "End": true  
        },  
        "WearLightJacket": {},  
        "None": {}  
    }  
}
```

```

    "WearLightJacket": {
        "Type": "Task",
        "Resource": "arn:aws:lambda:us-west-2:01234567890:function:wear_light_jacket",
        "End": true
    },
    "None": {
        "Type": "Task",
        "Resource": "arn:aws:lambda:us-west-2:01234567890:function:no_coat",
        "End": true
    }
}

```

Example Example Python function

The following Lambda function in Python (`wear_heavy_coat`) can be invoked for the state machine defined in the previous example. If the `WhichCoat` state machine equals a string value of `FREEZING`, the `wear_heavy_coat` function is invoked from Lambda, and the user receives the message that corresponds with the function: "You should wear a heavy coat today."

```

from __future__ import print_function

import datetime

def wear_heavy_coat(message, context):
    print(message)

    response = {}
    response['Weather'] = message['Weather']
    response['Timestamp'] = datetime.datetime.now().strftime("%Y-%m-%d %H-%M-%S")
    response['Message'] = 'You should wear a heavy coat today.'

    return response

```

Example Example invocation data

The following input data runs the `WearHeavyCoat` state that invokes the `wear_heavy_coat` Lambda function, when the `Weather` variable is equal to a string value of `FREEZING`.

```
{
    "Weather": "FREEZING"
}
```

For more information, see [Creating a Step Functions State Machine That Uses Lambda](#) in the *AWS Step Functions Developer Guide*.

Managing state machines in the Lambda console

You can use the Lambda console to edit and view details about your Step Functions state machines and the Lambda functions that they use.

Sections

- [Viewing state machine details \(p. 987\)](#)

- [Editing a state machine \(p. 987\)](#)
- [Running a state machine \(p. 987\)](#)

Viewing state machine details

The Lambda console displays a list of your state machines in the current AWS Region that contain at least one workflow step that invokes a Lambda function.

Choose a state machine to view a graphical representation of the workflow. Steps highlighted in blue represent Lambda functions. Use the graph controls to zoom in, zoom out, and center the graph.

Note

When a Lambda function is [dynamically referenced with JsonPath](#) in the state machine definition, the function details cannot be shown in the Lambda console. Instead, the function name is listed as a **Dynamic reference**, and the corresponding steps in the graph are grayed out.

To view state machine details

1. Open the Lambda console [Step Functions state machines page](#).
2. Choose a state machine.
<result>

*The Lambda console opens the Details page.
</result>*

For more information, see [Step Functions in the AWS Step Functions Developer Guide](#).

Editing a state machine

When you want to edit a state machine, Lambda opens the **Edit definition** page of the Step Functions console.

To edit a state machine

1. Open the Lambda console [Step Functions state machine page](#).
2. Choose a state machine.
3. Choose **Edit**.

The Step Functions console opens the Edit definition page.

4. Edit the state machine and choose **Save**.

For more information about editing state machines, see [Step Functions state machine language](#) in the [AWS Step Functions Developer Guide](#).

Running a state machine

When you want to run a state machine, Lambda opens the **New execution** page of the Step Functions console.

To run a state machine

1. Open the Lambda console [Step Functions state machines page](#).
2. Choose a state machine.

3. Choose **Execute**.

The Step Functions console opens the **New execution** page.

4. (Optional) Edit the state machine and choose **Start execution**.

For more information about running state machines, see [Step Functions state machine execution concepts](#) in the *AWS Step Functions Developer Guide*.

Orchestration examples with Step Functions

All work in your Step Functions state machine is done by [Tasks](#). A Task performs work by using an activity, a Lambda function, or by passing parameters to the API actions of other [Supported AWS Service Integrations for Step Functions](#).

Sections

- [Configuring a Lambda function as a task \(p. 988\)](#)
- [Configuring a state machine as an event source \(p. 988\)](#)
- [Handling function and service errors \(p. 989\)](#)
- [AWS CloudFormation and AWS SAM \(p. 990\)](#)

Configuring a Lambda function as a task

Step Functions can invoke Lambda functions directly from a Task state in an [Amazon States Language](#) definition.

```
...  
    "MyStateName":{  
        "Type":"Task",  
        "Resource":"arn:aws:lambda:us-west-2:01234567890:function:my_lambda_function",  
        "End":true  
    }  
...
```

You can create a Task state that invokes your Lambda function with the input to the state machine or any JSON document.

Example [event.json – Input to random-error function \(p. 1015\)](#)

```
{  
    "max-depth": 10,  
    "current-depth": 0,  
    "error-rate": 0.05  
}
```

Configuring a state machine as an event source

You can create a Step Functions state machine that invokes a Lambda function. The following example shows a Task state that invokes version 1 of a function named `my-function` with an event payload that has three keys. When the function returns a successful response, the state machine continues to the next task.

Example Example state machine

```
...
"Invoke": {
    "Type": "Task",
    "Resource": "arn:aws:states:::lambda:invoke",
    "Parameters": {
        "FunctionName": "arn:aws:lambda:us-east-2:123456789012:function:my-function:1",
        "Payload": {
            "max-depth": 10,
            "current-depth": 0,
            "error-rate": 0.05
        }
    },
    "Next": "NEXT_STATE",
    "TimeoutSeconds": 25
}
```

Permissions

Your state machine needs permission to call the Lambda API to invoke a function. To grant it permission, add the AWS managed policy [AWSLambdaRole](#) or a function-scoped inline policy to its role. For more information, see [How AWS Step Functions Works with IAM](#) in the [AWS Step Functions Developer Guide](#).

The FunctionName and Payload parameters map to parameters in the [Invoke \(p. 1260\)](#) API operation. In addition to these, you can also specify the InvocationType and ClientContext parameters. For example, to invoke the function asynchronously and continue to the next state without waiting for a result, you can set InvocationType to Event:

```
"InvocationType": "Event"
```

Instead of hard-coding the event payload in the state machine definition, you can use the input from the state machine execution. The following example uses the input specified when you run the state machine as the event payload:

```
"Payload.$": "$"
```

You can also invoke a function asynchronously and wait for it to make a callback with the AWS SDK. To do this, set the state's resource to `arn:aws:states:::lambda:invoke.waitForTaskToken`.

For more information, see [Invoke Lambda with Step Functions](#) in the [AWS Step Functions Developer Guide](#).

Handling function and service errors

When your function or the Lambda service returns an error, you can retry the invocation or continue to a different state based on the error type.

The following example shows an invoke task that retries on 5XX series Lambda API exceptions (`ServiceException`), throttles (`TooManyRequestsException`), runtime errors (`Lambda.Unknown`), and a function-defined error named `function.MaxDepthError`. It also catches an error named `function.DoublesRolledError` and continues to a state named `CaughtException` when it occurs.

Example Example catch and retry pattern

```
...
"Invoke": {
    "Type": "Task",
    "Resource": "arn:aws:states:::lambda:invoke",
```

```

"Retry": [
  {
    "ErrorEquals": [
      "function.MaxDepthError",
      "Lambda.TooManyRequestsException",
      "Lambda.ServiceException",
      "Lambda.Unknown"
    ],
    "MaxAttempts": 5
  }
],
"Catch": [
  {
    "ErrorEquals": [ "function.DoublesRolledError" ],
    "Next": "CaughtException"
  }
],
"Parameters": {
  "FunctionName": "arn:aws:lambda:us-east-2:123456789012:function:my-function:1",
  ...
}

```

To catch or retry function errors, create a custom error type. The name of the error type must match the `errorType` in the formatted error response that Lambda returns when you throw an error.

For more information on error handling in Step Functions, see [Handling Error Conditions Using a Step Functions State Machine](#) in the [AWS Step Functions Developer Guide](#).

AWS CloudFormation and AWS SAM

You can define state machines using a AWS CloudFormation template with AWS Serverless Application Model (AWS SAM). Using AWS SAM, you can define the state machine inline in the template or in a separate file. The following example shows a state machine that invokes a Lambda function that handles errors. It refers to a function resource defined in the same template (not shown).

Example Example branching pattern in template.yml

```

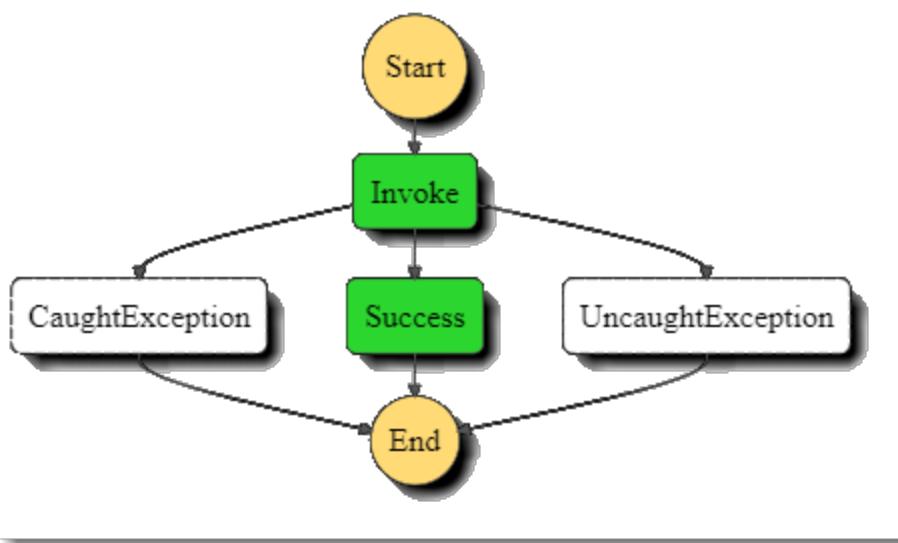
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: An AWS Lambda application that uses AWS Step Functions.
Resources:
  statemachine:
    Type: AWS::Serverless::StateMachine
    Properties:
      DefinitionSubstitutions:
        FunctionArn: !GetAtt function.Arn
        Payload: |
          {
            "max-depth": 5,
            "current-depth": 0,
            "error-rate": 0.2
          }
      Definition:
        StartAt: Invoke
        States:
          Invoke:
            Type: Task
            Resource: arn:aws:states:::lambda:invoke
            Parameters:
              FunctionName: "${FunctionArn}"
              Payload: "${Payload}"
              InvocationType: Event
        Retry:
          - ErrorEquals:

```

```

- function.MaxDepthError
- function.MaxDepthError
- Lambda.TooManyRequestsException
- Lambda.ServiceException
- Lambda.Unknown
IntervalSeconds: 1
MaxAttempts: 5
Catch:
- Equals:
  - function.DoublesRolledError
  Next: CaughtException
- Equals:
  - States.ALL
  Next: UncaughtException
  Next: Success
CaughtException:
Type: Pass
Result: The function returned an error.
End: true
UncaughtException:
Type: Pass
Result: Invocation failed.
End: true
Success:
Type: Pass
Result: Invocation succeeded!
End: true
Events:
scheduled:
Type: Schedule
Properties:
Description: Run every minute
Schedule: rate(1 minute)
Type: STANDARD
Policies:
- AWSLambdaRole
...
    
```

This creates a state machine with the following structure:



For more information, see [AWS::Serverless::StateMachine](#) in the *AWS Serverless Application Model Developer Guide*.

Improving startup performance with Lambda SnapStart

Lambda SnapStart for Java can improve startup performance for latency-sensitive applications by up to 10x at no extra cost, typically with no changes to your function code. The largest contributor to startup latency (often referred to as cold start time) is the time that Lambda spends initializing the function, which includes loading the function's code, starting the runtime, and initializing the function code.

With SnapStart, Lambda initializes your function when you publish a function version. Lambda takes a [Firecracker microVM](#) snapshot of the memory and disk state of the initialized [execution environment \(p. 14\)](#), encrypts the snapshot, and caches it for low-latency access. When you invoke the function version for the first time, and as the invocations scale up, Lambda resumes new execution environments from the cached snapshot instead of initializing them from scratch, improving startup latency.

Important

If your applications depend on uniqueness of state, you must evaluate your function code and verify that it is resilient to snapshot operations. For more information, see [Handling uniqueness with Lambda SnapStart \(p. 999\)](#).

Supported features and limitations

SnapStart supports the Java 11 and Java 17 (`java11` and `java17`) [managed runtimes \(p. 37\)](#). Other managed runtimes (such as `nodejs18.x` and `python3.10`), [custom runtimes \(p. 59\)](#), and container images are not supported.

SnapStart does not support [provisioned concurrency \(p. 213\)](#), the [arm64 architecture \(p. 29\)](#), [Amazon Elastic File System \(Amazon EFS\) \(p. 666\)](#), [AWS X-Ray \(p. 807\)](#), or ephemeral storage greater than 512 MB.

To work with SnapStart, you can use the Lambda console, the AWS Command Line Interface (AWS CLI), the Lambda API, the AWS SDK for Java, AWS CloudFormation, AWS Serverless Application Model (AWS SAM), and AWS Cloud Development Kit (AWS CDK). For more information, see [Activating and managing Lambda SnapStart \(p. 995\)](#).

Note

You can use SnapStart only on [published function versions \(p. 83\)](#) and [aliases \(p. 85\)](#) that point to versions. You can't use SnapStart on a function's unpublished version (\$LATEST).

Supported Regions

SnapStart is available in the following AWS Regions:

- US East (N. Virginia)
- US East (Ohio)
- US West (N. California)
- US West (Oregon)
- Asia Pacific (Mumbai)

- Asia Pacific (Seoul)
- Asia Pacific (Singapore)
- Asia Pacific (Sydney)
- Asia Pacific (Tokyo)
- Canada (Central)
- Europe (Frankfurt)
- Europe (Ireland)
- Europe (London)
- Europe (Stockholm)
- South America (São Paulo)

Compatibility considerations

With SnapStart, Lambda uses a single snapshot as the initial state for multiple execution environments. If your function uses any of the following during the [initialization phase \(p. 15\)](#), then you might need to make some changes before using SnapStart:

Uniqueness

If your initialization code generates unique content that is included in the snapshot, then the content might not be unique when it is reused across execution environments. To maintain uniqueness when using SnapStart, you must generate unique content after initialization. This includes unique IDs, unique secrets, and entropy that's used to generate pseudorandomness. To learn how to restore uniqueness, see [Handling uniqueness with Lambda SnapStart \(p. 999\)](#).

Network connections

The state of connections that your function establishes during the initialization phase isn't guaranteed when Lambda resumes your function from a snapshot. Validate the state of your network connections and re-establish them as necessary. In most cases, network connections that an AWS SDK establishes automatically resume. For other connections, review the [best practices \(p. 1006\)](#).

Temporary data

Some functions download or initialize ephemeral data, such as temporary credentials or cached timestamps, during the initialization phase. Refresh ephemeral data in the function handler before using it, even when not using SnapStart.

SnapStart pricing

There's no additional cost for SnapStart. You're charged based on the number of requests for your functions, the time that it takes your code to run, and the memory configured for your function. Duration is calculated from the time that your code begins running until it returns or otherwise ends, rounded up to the nearest 1 ms.

Duration charges apply to code that runs in the function [handler \(p. 389\)](#), initialization code that's declared outside of the handler, the time it takes for the runtime (JVM) to load, and any code that runs in a [runtime hook \(p. 1001\)](#). For more information about how Lambda calculates duration, see [Monitoring for Lambda SnapStart \(p. 1003\)](#).

For functions configured with SnapStart, Lambda periodically recycles the execution environments and re-runs your initialization code. For resiliency, Lambda creates snapshots in multiple Availability Zones.

Charges apply each time that Lambda re-runs your initialization code in another Availability Zone. For more information about how Lambda calculates charges, see [AWS Lambda Pricing](#).

Comparing Lambda SnapStart and provisioned concurrency

Both Lambda SnapStart and [provisioned concurrency \(p. 213\)](#) can reduce cold starts and outlier latencies when a function scales up. SnapStart helps you improve startup performance by up to 10x at no extra cost. Provisioned concurrency keeps functions initialized and ready to respond in double-digit milliseconds. Configuring provisioned concurrency incurs charges to your AWS account. Use provisioned concurrency if your application has strict cold start latency requirements. You can't use both SnapStart and provisioned concurrency on the same function version.

Note

SnapStart works best when used with function invocations at scale. Functions that are invoked infrequently might not experience the same performance improvements.

Additional resources

In addition to reading the other topics in this chapter, we also recommend that you try the [Starting up faster with AWS Lambda SnapStart](#) workshop and watch the [Fast cold starts for your Java functions](#) session from AWS re:Invent 2022.

Topics

- [Activating and managing Lambda SnapStart \(p. 995\)](#)
- [Handling uniqueness with Lambda SnapStart \(p. 999\)](#)
- [Runtime hooks for Lambda SnapStart \(p. 1001\)](#)
- [Monitoring for Lambda SnapStart \(p. 1003\)](#)
- [Security model for Lambda SnapStart \(p. 1005\)](#)
- [Best practices for working with Lambda SnapStart \(p. 1006\)](#)

Activating and managing Lambda SnapStart

To use SnapStart, activate SnapStart on a new or existing Lambda function. Then, publish and invoke a function version.

Topics

- [Activating SnapStart \(console\) \(p. 995\)](#)
- [Activating SnapStart \(AWS CLI\) \(p. 995\)](#)
- [Activating SnapStart \(API\) \(p. 997\)](#)
- [Lambda SnapStart and function states \(p. 997\)](#)
- [Updating a snapshot \(p. 998\)](#)
- [Using SnapStart with the AWS SDK for Java \(p. 998\)](#)
- [Using SnapStart with AWS CloudFormation, AWS SAM, and AWS CDK \(p. 998\)](#)
- [Deleting snapshots \(p. 998\)](#)

Activating SnapStart (console)

To activate SnapStart for a function

1. Open the [Functions page](#) of the Lambda console.
2. Choose the name of a function.
3. Choose **Configuration**, and then choose **General configuration**.
4. On the **General configuration** pane, choose **Edit**.
5. On the **Edit basic settings** page, for **SnapStart**, choose **Published versions**.
6. Choose **Save**.
7. [Publish a function version \(p. 83\)](#). Lambda initializes your code, creates a snapshot of the initialized execution environment, and then caches the snapshot for low-latency access.
8. [Invoke the function version \(p. 84\)](#).

Activating SnapStart (AWS CLI)

To activate SnapStart for an existing function

1. Update the function configuration by running the [update-function-configuration](#) command with the **--snap-start** option.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --snap-start ApplyOn=PublishedVersions
```

2. Publish a function version with the [publish-version](#) command.

```
aws lambda publish-version \
  --function-name my-function
```

3. Confirm that SnapStart is activated for the function version by running the [get-function-configuration](#) command and specifying the version number. The following example specifies version 1.

```
aws lambda get-function-configuration \
```

```
--function-name my-function:1
```

If the response shows that [OptimizationStatus \(p. 1462\)](#) is On and [State \(p. 1234\)](#) is Active, then SnapStart is activated and a snapshot is available for the specified function version.

```
"SnapStart": {
    "ApplyOn": "PublishedVersions",
    "OptimizationStatus": "On"
},
"State": "Active",
```

4. Invoke the function version by running the [invoke](#) command and specifying the version. The following example invokes version 1.

```
aws lambda invoke \
--cli-binary-format raw-in-base64-out \
--function-name my-function:1 \
--payload '{ "name": "Bob" }' \
response.json
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

To activate SnapStart when you create a new function

1. Create a function by running the [create-function](#) command with the **--snap-start** option. For **--role**, specify the Amazon Resource Name (ARN) of your [execution role \(p. 816\)](#).

```
aws lambda create-function \
--function-name my-function \
--runtime "java17" \
--zip-file fileb://my-function.zip \
--handler my-function.handler \
--role arn:aws:iam::111122223333:role/lambda-ex \
--snap-start ApplyOn=PublishedVersions
```

2. Create a version with the [publish-version](#) command.

```
aws lambda publish-version \
--function-name my-function
```

3. Confirm that SnapStart is activated for the function version by running the [get-function-configuration](#) command and specifying the version number. The following example specifies version 1.

```
aws lambda get-function-configuration \
--function-name my-function:1
```

If the response shows that [OptimizationStatus \(p. 1462\)](#) is On and [State \(p. 1234\)](#) is Active, then SnapStart is activated and a snapshot is available for the specified function version.

```
"SnapStart": {
    "ApplyOn": "PublishedVersions",
    "OptimizationStatus": "On"
},
"State": "Active",
```

4. Invoke the function version by running the [invoke](#) command and specifying the version. The following example invokes version 1.

```
aws lambda invoke \
--cli-binary-format raw-in-base64-out \
--function-name my-function:1 \
--payload '{ "name": "Bob" }' \
response.json
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see [AWS CLI supported global command line options](#).

Activating SnapStart (API)

To activate SnapStart

1. Do one of the following:
 - Create a new function with SnapStart activated by using the [CreateFunction \(p. 1165\)](#) API action with the [SnapStart \(p. 1461\)](#) parameter.
 - Activate SnapStart for an existing function by using the [UpdateFunctionConfiguration \(p. 1377\)](#) action with the [SnapStart \(p. 1461\)](#) parameter.
2. Publish a function version with the [PublishVersion \(p. 1315\)](#) action. Lambda initializes your code, creates a snapshot of the initialized execution environment, and then caches the snapshot for low-latency access.
3. Confirm that SnapStart is activated for the function version by using the [GetFunctionConfiguration \(p. 1229\)](#) action. Specify a version number to confirm that SnapStart is activated for that version. If the response shows that [OptimizationStatus \(p. 1462\)](#) is On and [State \(p. 1234\)](#) is Active, then SnapStart is activated and a snapshot is available for the specified function version.

```
"SnapStart": {
    "ApplyOn": "PublishedVersions",
    "OptimizationStatus": "On"
},
"State": "Active",
```

4. Invoke the function version with the [Invoke \(p. 1260\)](#) action.

Lambda SnapStart and function states

The following function states can occur when you use SnapStart. They can also occur when Lambda periodically recycles the execution environment and re-runs the initialization code for a function that's configured with SnapStart.

- Pending – Lambda is initializing your code and taking a snapshot of the initialized execution environment. Any invocations or other API actions that operate on the function version will fail.
- Active – Snapshot creation is complete and you can invoke the function. To use SnapStart, you must invoke the published function version, not the unpublished version (\$LATEST).
- Inactive – The function version hasn't been invoked for 14 days. When the function version becomes Inactive, Lambda deletes the snapshot. If you invoke the function version after 14 days, Lambda returns a SnapStartNotReadyException response and begins initializing a new snapshot. Wait until the function version reaches the Active state, and then invoke it again.

- Failed – Lambda encountered an error when running the initialization code or creating the snapshot.

Updating a snapshot

Lambda creates a snapshot for each published function version. To update a snapshot, publish a new function version. Lambda automatically updates your snapshots with the latest runtime and security patches.

Using SnapStart with the AWS SDK for Java

To make AWS SDK calls from your function, Lambda generates an ephemeral set of credentials by assuming your function's execution role. These credentials are available as environment variables during your function's invocation. You don't need to provide credentials for the SDK directly in code. By default, the credential provider chain sequentially checks each place where you can set credentials and chooses the first available—usually the environment variables (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_SESSION_TOKEN`).

Note

When SnapStart is activated, the Java runtime automatically uses the container credentials (`AWS_CONTAINER_CREDENTIALS_FULL_URI` and `AWS_CONTAINER_AUTHORIZATION_TOKEN`) instead of the access key environment variables. This prevents credentials from expiring before the function is restored.

Using SnapStart with AWS CloudFormation, AWS SAM, and AWS CDK

- **AWS CloudFormation:** Declare the [SnapStart](#) entity in your template.
- **AWS Serverless Application Model (AWS SAM):** Declare the [SnapStart](#) property in your template.
- **AWS Cloud Development Kit (AWS CDK):** Use the [SnapStartProperty](#) type.

Deleting snapshots

Lambda deletes snapshots when:

- You delete the function or function version.
- You don't invoke the function version for 14 days. After 14 days without an invocation, the function version transitions to the [Inactive \(p. 997\)](#) state. If you invoke the function version after 14 days, Lambda returns a `SnapStartNotReadyException` response and begins initializing a new snapshot. Wait until the function version reaches the [Active \(p. 997\)](#) state, and then invoke it again.

Lambda removes all resources associated with deleted snapshots in compliance with the General Data Protection Regulation (GDPR).

Handling uniqueness with Lambda SnapStart

When invocations scale up on a SnapStart function, Lambda uses a single initialized snapshot to resume multiple execution environments. If your initialization code generates unique content that is included in the snapshot, then the content might not be unique when it is reused across execution environments. To maintain uniqueness when using SnapStart, you must generate unique content after initialization. This includes unique IDs, unique secrets, and entropy that's used to generate pseudorandomness.

We recommend the following best practices to help you maintain uniqueness in your code. Lambda also provides an open-source [SnapStart scanning tool \(p. 1000\)](#) to help check for code that assumes uniqueness. If you generate unique data during the initialization phase, then you can use a [runtime hook \(p. 1001\)](#) to restore uniqueness. With runtime hooks, you can run specific code immediately before Lambda takes a snapshot or immediately after Lambda resumes a function from a snapshot.

Avoid saving state that depends on uniqueness during initialization

During the [initialization phase \(p. 15\)](#) of your function, avoid caching data that's intended to be unique, such as generating a unique ID for logging. Instead, we recommend that you generate unique data inside your function handler or use a [runtime hook \(p. 1001\)](#).

Example – Generating a unique ID in function handler

The following example demonstrates how to generate a UUID in the 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)
            uniqueSandboxId = UUID.randomUUID();
        System.out.println("Unique Sandbox Id: " + uniqueSandboxId);
        return "Hello, World!";
    }
}
```

Use cryptographically secure pseudorandom number generators (CSPRNGs)

If your application depends on randomness, we recommend that you use cryptographically secure random number generators (CSPRNGs). The Lambda managed runtime for Java includes two built-in CSPRNGs (OpenSSL 1.0.2 and `java.security.SecureRandom`) that automatically maintain randomness with SnapStart. Software that always gets random numbers from `/dev/random` or `/dev/urandom` also maintains randomness with SnapStart.

Example – `java.security.SecureRandom`

The following example uses `java.security.SecureRandom`, which generates unique number sequences even when the function is restored from a snapshot.

```
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.nextInt());
    }
    return "Hello, World!";
}
```

SnapStart scanning tool

Lambda provides a scanning tool to help you check for code that assumes uniqueness. The SnapStart scanning tool is an open-source [SpotBugs](#) plugin that runs a static analysis against a set of rules. The scanning tool helps identify potential code implementations that might break assumptions regarding uniqueness. For installation instructions and a list of checks that the scanning tool performs, see the [aws-lambda-snapstart-java-rules](#) repository on GitHub.

To learn more about handling uniqueness with SnapStart, see [Starting up faster with AWS Lambda SnapStart](#) on the AWS Compute Blog.

Runtime hooks for Lambda SnapStart

You can use runtime hooks to implement code before Lambda creates a snapshot or after Lambda resumes a function from a snapshot. Runtime hooks are available as part of the open-source Coordinated Restore at Checkpoint (CRaC) project. CRaC is in development for the [Open Java Development Kit \(OpenJDK\)](#). For an example of how to use CRaC with a reference application, see the [CRaC](#) repository on GitHub. CRaC uses three main elements:

- Resource – An interface with two methods, `beforeCheckpoint()` and `afterRestore()`. Use these methods to implement the code that you want to run before a snapshot and after a restore.
- Context `<R extends Resource>` – To receive notifications for checkpoints and restores, a Resource must be registered with a Context.
- Core – The coordination service, which provides the default global Context via the static method `Core.getGlobalContext()`.

For more information about Context and Resource, see [Package org.crac](#) in the CRaC documentation.

Use the following steps to implement runtime hooks with the [org.crac package](#). The Lambda runtime contains a customized CRaC context implementation that calls your runtime hooks before checkpointing and after restoring.

Step 1: Update the build configuration

Add the `org.crac` dependency to the build configuration. The following example uses Gradle. For examples for other build systems, see the [Apache Maven documentation](#).

```
dependencies {  
    compile group: 'com.amazonaws', name: 'aws-lambda-java-core', version: '1.2.1'  
    # All other project dependencies go here:  
    # ...  
    # Then, add the org.crac dependency:  
    implementation group: 'io.github.crac', name: 'org-crac', version: '0.1.3'  
}
```

Step 2: Update the Lambda handler

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. When the handler exits or returns a response, it becomes available to handle another event.

For more information, see [AWS Lambda function handler in Java \(p. 389\)](#).

The following example handler shows how to run code before checkpointing (`beforeCheckpoint()`) and after restoring (`afterRestore()`). This handler also registers the Resource to the runtime-managed global Context.

Note

When Lambda creates a snapshot, your initialization code can run for up to 15 minutes. The time limit is 130 seconds or the [configured function timeout \(p. 73\)](#) (maximum 900 seconds), whichever is higher. Your `beforeCheckpoint()` runtime hooks count towards the initialization code time limit. When Lambda restores a snapshot, the runtime (JVM) must load and `afterRestore()` runtime hooks must complete within the timeout limit (10 seconds). Otherwise, you'll get a `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");
    }
}

```

Context maintains only a [WeakReference](#) to the registered object. If a [Resource](#) is garbage collected, runtime hooks do not run. Your code must maintain a strong reference to the Resource to guarantee that the runtime hook runs.

Here are two examples of patterns to avoid:

Example – Object without a strong reference

```
Core.getGlobalContext().register( new MyResource() );
```

Example – Objects of anonymous classes

```

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 {
        // ...
    }
} );

```

Instead, maintain a strong reference. In the following example, the registered resource isn't garbage collected and runtime hooks run consistently.

Example – Object with a strong reference

```

Resource myResource = new MyResource(); // This reference must be maintained to prevent the
                                         // registered resource from being garbage collected
Core.getGlobalContext().register( myResource );

```

Monitoring for Lambda SnapStart

You can monitor your Lambda SnapStart functions using Amazon CloudWatch and the [Lambda Telemetry API \(p. 911\)](#).

Note

The AWS_LAMBDA_LOG_GROUP_NAME and AWS_LAMBDA_LOG_STREAM_NAME [environment variables \(p. 79\)](#) are not available in Lambda SnapStart functions.

CloudWatch for SnapStart

There are a few differences with the [CloudWatch log stream \(p. 873\)](#) format for SnapStart functions:

- **Initialization logs** – When a new execution environment is created, the REPORT doesn't include the Init Duration field. That's because Lambda initializes SnapStart functions when you create a version instead of during function invocation. For SnapStart functions, the Init Duration field is in the INIT_REPORT record. This record shows duration details for the [Init phase \(p. 15\)](#), including the duration of any beforeCheckpoint [runtime hooks \(p. 1001\)](#).
- **Invocation logs** – When a new execution environment is created, the REPORT includes the Restore Duration and Billed Restore Duration fields:
 - Restore Duration: The time it takes for Lambda to restore a snapshot, load the runtime (JVM), and run any afterRestore runtime hooks.
 - Billed Restore Duration: The time it takes for Lambda to load the runtime (JVM) and run any afterRestore hooks. You are not charged for the time it takes to restore a snapshot.

Note

Duration charges apply to code that runs in the function [handler \(p. 389\)](#), initialization code that's declared outside of the handler, the time it takes for the runtime (JVM) to load, and any code that runs in a [runtime hook \(p. 1001\)](#). For more information, see [SnapStart pricing \(p. 993\)](#).

The cold start duration is the sum of Restore Duration + Duration.

The following example is a Lambda Insights query that returns the latency percentiles for SnapStart functions. For more information about Lambda Insights queries, see [Example workflow using queries to troubleshoot a function \(p. 868\)](#).

```
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
```

Telemetry API events for SnapStart

Lambda sends the following SnapStart events to the [Telemetry API \(p. 911\)](#):

- [platform.restoreStart \(p. 927\)](#) – Shows the time when the [Restore phase \(p. 15\)](#) started.
- [platform.restoreRuntimeDone \(p. 928\)](#) – Shows whether the Restore phase was successful. Lambda sends this message when the runtime sends a restore/next runtime API request. There are three possible statuses: success, failure, and timeout.

- [platform.restoreReport \(p. 929\)](#) – Shows how long the Restore phase lasted and how many milliseconds you were billed for during this phase.

Amazon API Gateway and function URL metrics

If you create a web API [using API Gateway \(p. 562\)](#), then you can use the [IntegrationLatency](#) metric to measure end-to-end latency (the time between when API Gateway relays a request to the backend and when it receives a response from the backend).

If you're using a [Lambda function URL \(p. 166\)](#), then you can use the [UrlRequestLatency \(p. 184\)](#) metric to measure end-to-end latency (the time between when the function URL receives a request and when the function URL returns a response).

Security model for Lambda SnapStart

Lambda SnapStart supports encryption at rest. Lambda encrypts snapshots with an AWS KMS key. By default, Lambda uses an AWS managed key. If this default behavior suits your workflow, then you don't need to set up anything else. Otherwise, you can use the `--kms-key-arn` option in the [create-function](#) or [update-function-configuration](#) command to provide an AWS KMS customer managed key. You might do this to control rotation of the KMS key or to meet the requirements of your organization for managing KMS keys. Customer managed keys incur standard AWS KMS charges. For more information, see [AWS Key Management Service pricing](#).

When you delete a SnapStart function or function version, all Invoke requests to that function or function version fail. Lambda automatically deletes snapshots that are not invoked for 14 days. Lambda removes all resources associated with deleted snapshots in compliance with the General Data Protection Regulation (GDPR).

Best practices for working with Lambda SnapStart

Topics

- [Network connections \(p. 1006\)](#)
- [Performance tuning \(p. 1006\)](#)

Network connections

The state of connections that your function establishes during the initialization phase isn't guaranteed when Lambda resumes your function from a snapshot. In most cases, network connections that an AWS SDK establishes automatically resume. For other connections, we recommend the following best practices.

Re-establish network connections

Always re-establish your network connections when your function resumes from a snapshot. We recommend that you re-establish network connections in the function handler. Alternatively, you can use an [afterRestore runtime hook \(p. 1001\)](#).

Don't use hostname as a unique execution environment identifier

We recommend against using hostname to identify your execution environment as a unique node or container in your applications. With SnapStart, a single snapshot is used as the initial state for multiple execution environments, and all execution environments return the same hostname value for `InetAddress.getLocalHost()`. For applications that require a unique execution environment identity or hostname value, we recommend that you generate a unique ID in the function handler. Or, use an [afterRestore runtime hook \(p. 1001\)](#) to generate a unique ID, and then use the unique ID as the identifier for the execution environment.

Avoid binding connections to fixed source ports

We recommend that you avoid binding network connections to fixed source ports. Connections are re-established when a function resumes from a snapshot, and network connections that are bound to a fixed source port might fail.

Avoid using Java DNS cache

Lambda functions already cache DNS responses. If you use another DNS cache with SnapStart, then you might experience connection timeouts when the function resumes from a snapshot.

Performance tuning

Note

SnapStart works best when used with function invocations at scale. Functions that are invoked infrequently might not experience the same performance improvements.

To maximize the benefits of SnapStart, we recommend that you preload classes that contribute to startup latency in your initialization code instead of in the function handler. This moves the latency associated with heavy class loading out of the invocation path, optimizing startup performance with SnapStart.

If you can't preload classes during initialization, then we recommend that you preload classes with dummy invocations. To do this, update the function handler code, as shown in the following example from the [pet store function](#) on the AWS Labs GitHub repository.

```
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);
    }
}
```

Lambda sample applications

The GitHub repository for this guide includes sample applications that demonstrate the use of various languages and AWS services. Each sample application includes scripts for easy deployment and cleanup, an AWS SAM template, and supporting resources.

Node.js

Sample Lambda applications in Node.js

- [blank-nodejs](#) – A Node.js function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.
- [nodejs-apig](#) – A function with a public API endpoint that processes an event from API Gateway and returns an HTTP response.
- [rds-mysql](#) – A function that relays queries to a MySQL for RDS Database. This sample includes a private VPC and database instance configured with a password in AWS Secrets Manager.
- [efs-nodejs](#) – A function that uses an Amazon EFS file system in a Amazon VPC. This sample includes a VPC, file system, mount targets, and access point configured for use with Lambda.
- [list-manager](#) – A function processes events from an Amazon Kinesis data stream and update aggregate lists in Amazon DynamoDB. The function stores a record of each event in a MySQL for RDS Database in a private VPC. This sample includes a private VPC with a VPC endpoint for DynamoDB and a database instance.
- [error-processor](#) – A Node.js function generates errors for a specified percentage of requests. A CloudWatch Logs subscription invokes a second function when an error is recorded. The processor function uses the AWS SDK to gather details about the request and stores them in an Amazon S3 bucket.

Python

Sample Lambda applications in Python

- [blank-python](#) – A Python function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.

Ruby

Sample Lambda applications in Ruby

- [blank-ruby](#) – A Ruby function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.
- [Ruby Code Samples for AWS Lambda](#) – Code samples written in Ruby that demonstrate how to interact with AWS Lambda.

Java

Sample Lambda applications in Java

- [java-basic](#) – A collection of minimal Java functions with unit tests and variable logging configuration.
- [java-events](#) – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These

functions use the latest version of the [aws-lambda-java-events \(p. 393\)](#) library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.

- [s3-java](#) – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- [Use API Gateway to invoke a Lambda function](#) – A Java function that scans a Amazon DynamoDB table that contains employee information. It then uses Amazon Simple Notification Service to send a text message to employees celebrating their work anniversaries. This example uses API Gateway to invoke the function.

Sample Spring applications in Lambda

- [spring-cloud-function-samples](#) – An example that shows how to use the [Spring Cloud Function](#) framework to create AWS Lambda functions.

Go

Lambda provides the following sample applications for the Go runtime:

Sample Lambda applications in Go

- [blank-go](#) – A Go function that shows the use of Lambda's Go libraries, logging, environment variables, and the AWS SDK.

C#

Sample Lambda applications in C#

- [blank-csharp](#) – A C# function that shows the use of Lambda's .NET libraries, logging, environment variables, AWS X-Ray tracing, unit tests, and the AWS SDK.
- [ec2-spot](#) – A function that manages spot instance requests in Amazon EC2.

PowerShell

Lambda provides the following sample applications for the PowerShell runtime:

- [blank-powershell](#) – A PowerShell function that shows the use of logging, environment variables, and the AWS SDK.

To deploy a sample application, follow the instructions in its README file. To learn more about the architecture and use cases of an application, read the topics in this chapter.

Topics

- [Blank function sample application for AWS Lambda \(p. 1010\)](#)
- [Error processor sample application for AWS Lambda \(p. 1015\)](#)
- [List manager sample application for AWS Lambda \(p. 1018\)](#)

Blank function sample application for AWS Lambda

The blank function sample application is a starter application that demonstrates common operations in Lambda with a function that calls the Lambda API. It shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK. Explore this application to learn about building Lambda functions in your programming language, or use it as a starting point for your own projects.

Variants of this sample application are available for the following languages:

Variants

- Node.js – [blank-nodejs](#).
- Python – [blank-python](#).
- Ruby – [blank-ruby](#).
- Java – [blank-java](#).
- Go – [blank-go](#).
- C# – [blank-csharp](#).
- PowerShell – [blank-powershell](#).

The examples in this topic highlight code from the Node.js version, but the details are generally applicable to all variants.

You can deploy the sample in a few minutes with the AWS CLI and AWS CloudFormation. Follow the instructions in the [README](#) to download, configure, and deploy it in your account.

Sections

- [Architecture and handler code \(p. 1010\)](#)
- [Deployment automation with AWS CloudFormation and the AWS CLI \(p. 1011\)](#)
- [Instrumentation with the AWS X-Ray \(p. 1013\)](#)
- [Dependency management with layers \(p. 1013\)](#)

Architecture and handler code

The sample application consists of function code, an AWS CloudFormation template, and supporting resources. When you deploy the sample, you use the following AWS services:

- AWS Lambda – Runs function code, sends logs to CloudWatch Logs, and sends trace data to X-Ray. The function also calls the Lambda API to get details about the account's quotas and usage in the current Region.
- [AWS X-Ray](#) – Collects trace data, indexes traces for search, and generates a service map.
- [Amazon CloudWatch](#) – Stores logs and metrics.
- [AWS Identity and Access Management \(IAM\)](#) – Grants permission.
- [Amazon Simple Storage Service \(Amazon S3\)](#) – Stores the function's deployment package during deployment.
- [AWS CloudFormation](#) – Creates application resources and deploys function code.

Standard charges apply for each service. For more information, see [AWS Pricing](#).

The function code shows a basic workflow for processing an event. The handler takes an Amazon Simple Queue Service (Amazon SQS) event as input and iterates through the records that it contains, logging the contents of each message. It logs the contents of the event, the context object, and environment variables. Then it makes a call with the AWS SDK and passes the response back to the Lambda runtime.

Example [blank-nodejs/function/index.js](#) – Handler code

```
// Handler
exports.handler = async function(event, context) {
    event.Records.forEach(record => {
        console.log(record.body)
    })
    console.log('## ENVIRONMENT VARIABLES: ' + serialize(process.env))
    console.log('## CONTEXT: ' + serialize(context))
    console.log('## EVENT: ' + serialize(event))

    return getAccountSettings()
}

// Use SDK client
var getAccountSettings = function(){
    return lambda.getAccountSettings().promise()
}

var serialize = function(object) {
    return JSON.stringify(object, null, 2)
}
```

The input/output types for the handler and support for asynchronous programming vary per runtime. In this example, the handler method is `async`, so in Node.js this means that it must return a promise back to the runtime. The Lambda runtime waits for the promise to be resolved and returns the response to the invoker. If the function code or AWS SDK client return an error, the runtime formats the error into a JSON document and returns that.

The sample application doesn't include an Amazon SQS queue to send events, but uses an event from Amazon SQS ([event.json](#)) to illustrate how events are processed. To add an Amazon SQS queue to your application, see [Using Lambda with Amazon SQS \(p. 778\)](#).

Deployment automation with AWS CloudFormation and the AWS CLI

The sample application's resources are defined in an AWS CloudFormation template and deployed with the AWS CLI. The project includes simple shell scripts that automate the process of setting up, deploying, invoking, and tearing down the application.

The application template uses an AWS Serverless Application Model (AWS SAM) resource type to define the model. AWS SAM simplifies template authoring for serverless applications by automating the definition of execution roles, APIs, and other resources.

The template defines the resources in the application *stack*. This includes the function, its execution role, and a Lambda layer that provides the function's library dependencies. The stack does not include the bucket that the AWS CLI uses during deployment or the CloudWatch Logs log group.

Example [blank-nodejs/template.yml](#) – Serverless resources

```
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:
    Handler: index.handler
    Runtime: nodejs12.x
    CodeUri: function/
    Description: Call the AWS Lambda API
    Timeout: 10
    # Function's execution role
    Policies:
      - AWSLambdaBasicExecutionRole
      - AWSLambda_ReadOnlyAccess
      - AWSXrayWriteOnlyAccess
    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:
        - nodejs12.x

```

When you deploy the application, AWS CloudFormation applies the AWS SAM transform to the template to generate an AWS CloudFormation template with standard types such as AWS::Lambda::Function and AWS::IAM::Role.

Example processed template

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "An AWS Lambda application that calls the Lambda API.",
  "Resources": {
    "function": {
      "Type": "AWS::Lambda::Function",
      "Properties": {
        "Layers": [
          {
            "Ref": "libs32xmpl61b2"
          }
        ],
        "TracingConfig": {
          "Mode": "Active"
        },
        "Code": {
          "S3Bucket": "lambda-artifacts-6b000xmpl1e9bf2a",
          "S3Key": "3d3axmpl473d249d039d2d7a37512db3"
        },
        "Description": "Call the AWS Lambda API",
        "Tags": [
          {
            "Value": "SAM",
            "Key": "lambda:createdBy"
          }
        ],
      }
    }
  }
}
```

In this example, the Code property specifies an object in an Amazon S3 bucket. This corresponds to the local path in the CodeUri property in the project template:

```
CodeUri: function/.
```

To upload the project files to Amazon S3, the deployment script uses commands in the AWS CLI. The `cloudformation package` command preprocesses the template, uploads artifacts, and replaces local paths with Amazon S3 object locations. The `cloudformation deploy` command deploys the processed template with a AWS CloudFormation change set.

Example [blank-nodejs/3-deploy.sh](#) – Package and deploy

```
#!/bin/bash
set -eo pipefail
ARTIFACT_BUCKET=$(cat bucket-name.txt)
aws cloudformation package --template-file template.yml --s3-bucket $ARTIFACT_BUCKET --
output-template-file out.yml
aws cloudformation deploy --template-file out.yml --stack-name blank-nodejs --capabilities
CAPABILITY_NAMED_IAM
```

The first time you run this script, it creates a AWS CloudFormation stack named `blank-nodejs`. If you make changes to the function code or template, you can run it again to update the stack.

The cleanup script ([blank-nodejs/5-cleanup.sh](#)) deletes the stack and optionally deletes the deployment bucket and function logs.

Instrumentation with the AWS X-Ray

The sample function is configured for tracing with [AWS X-Ray](#). With the tracing mode set to active, Lambda records timing information for a subset of invocations and sends it to X-Ray. X-Ray processes the data to generate a *service map* that shows a client node and two service nodes.

The first service node (AWS : : Lambda) represents the Lambda service, which validates the invocation request and sends it to the function. The second node, AWS : : Lambda : : Function, represents the function itself.

To record additional detail, the sample function uses the X-Ray SDK. With minimal changes to the function code, the X-Ray SDK records details about calls made with the AWS SDK to AWS services.

Example [blank-nodejs/function/index.js](#) – Instrumentation

```
const AWSXRay = require('aws-xray-sdk-core')
const AWS = AWSXRay.captureAWS(require('aws-sdk'))

// Create client outside of handler to reuse
const lambda = new AWS.Lambda()
```

Instrumenting the AWS SDK client adds an additional node to the service map and more detail in traces. In this example, the service map shows the sample function calling the Lambda API to get details about storage and concurrency usage in the current Region.

The trace shows timing details for the invocation, with subsegments for function initialization, invocation, and overhead. The invocation subsegment has a subsegment for the AWS SDK call to the `GetAccountSettings` API operation.

You can include the X-Ray SDK and other libraries in your function's deployment package, or deploy them separately in a Lambda layer. For Node.js, Ruby, and Python, the Lambda runtime includes the AWS SDK in the execution environment.

Dependency management with layers

You can install libraries locally and include them in the deployment package that you upload to Lambda, but this has its drawbacks. Larger file sizes cause increased deployment times and can prevent you from

testing changes to your function code in the Lambda console. To keep the deployment package small and avoid uploading dependencies that haven't changed, the sample app creates a [Lambda layer \(p. 93\)](#) and associates it with the function.

Example [blank-nodejs/template.yml](#) – Dependency layer

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs12.x
      CodeUri: function/
      Description: Call the AWS Lambda API
      Timeout: 10
      # Function's execution role
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
      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:
        - nodejs12.x
```

The `2-build-layer.sh` script installs the function's dependencies with `npm` and places them in a folder with the [structure required by the Lambda runtime \(p. 94\)](#).

Example [2-build-layer.sh](#) – Preparing the layer

```
#!/bin/bash
set -eo pipefail
mkdir -p lib/nodejs
rm -rf node_modules lib/nodejs/node_modules
npm install --production
mv node_modules lib/nodejs/
```

The first time that you deploy the sample application, the AWS CLI packages the layer separately from the function code and deploys both. For subsequent deployments, the layer archive is only uploaded if the contents of the `lib` folder have changed.

Error processor sample application for AWS Lambda

The Error Processor sample application demonstrates the use of AWS Lambda to handle events from an [Amazon CloudWatch Logs subscription \(p. 597\)](#). CloudWatch Logs lets you invoke a Lambda function when a log entry matches a pattern. The subscription in this application monitors the log group of a function for entries that contain the word ERROR. It invokes a processor Lambda function in response. The processor function retrieves the full log stream and trace data for the request that caused the error, and stores them for later use.

Function code is available in the following files:

- Random error – [random-error/index.js](#)
- Processor – [processor/index.js](#)

You can deploy the sample in a few minutes with the AWS CLI and AWS CloudFormation. To download, configure, and deploy it in your account, follow the instructions in the [README](#).

Sections

- [Architecture and event structure \(p. 1015\)](#)
- [Instrumentation with AWS X-Ray \(p. 1016\)](#)
- [AWS CloudFormation template and additional resources \(p. 1016\)](#)

Architecture and event structure

The sample application uses the following AWS services.

- AWS Lambda – Runs function code, sends logs to CloudWatch Logs, and sends trace data to X-Ray.
- Amazon CloudWatch Logs – Collects logs, and invokes a function when a log entry matches a filter pattern.
- AWS X-Ray – Collects trace data, indexes traces for search, and generates a service map.
- Amazon Simple Storage Service (Amazon S3) – Stores deployment artifacts and application output.

Standard charges apply for each service.

A Lambda function in the application generates errors randomly. When CloudWatch Logs detects the word ERROR in the function's logs, it sends an event to the processor function for processing.

Example CloudWatch Logs message event

```
{  
  "awslogs": {  
    "data": "H4sIAAAAAAAAHQQT0/DMAzFv0vEkbLYcdJkt4qVXmCDteIAm1DbZKjs  
+kdpB0Jo350MhsQFyVLsZ+unl/fJWje05asrPgbH5..."  
  }  
}
```

When it's decoded, the data contains details about the log event. The function uses these details to identify the log stream, and parses the log message to get the ID of the request that caused the error.

Example decoded CloudWatch Logs event data

```
{  
    "messageType": "DATA_MESSAGE",  
    "owner": "123456789012",  
    "logGroup": "/aws/lambda/lambda-error-processor-randomerror-1GD4SSDNACNP4",  
    "logStream": "2019/04/04/[$LATEST]63311769a9d742f19cedf8d2e38995b9",  
    "subscriptionFilters": [  
        "lambda-error-processor-subscription-150PDVQ59CG07"  
    ],  
    "logEvents": [  
        {  
            "id": "34664632210239891980253245280462376874059932423703429141",  
            "timestamp": 1554415868243,  
            "message": "2019-04-04T22:11:08.243Z\t1d2c1444-efd1-43ec-  
b16e-8fb2d37508b8\tERROR\n"  
        }  
    ]  
}
```

The processor function uses information from the CloudWatch Logs event to download the full log stream and X-Ray trace for a request that caused an error. It stores both in an Amazon S3 bucket. To allow the log stream and trace time to finalize, the function waits for a short period of time before accessing the data.

Instrumentation with AWS X-Ray

The application uses [AWS X-Ray \(p. 807\)](#) to trace function invocations and the calls that functions make to AWS services. X-Ray uses the trace data that it receives from functions to create a service map that helps you identify errors.

The two Node.js functions are configured for active tracing in the template, and are instrumented with the AWS X-Ray SDK for Node.js in code. With active tracing, Lambda tags adds a tracing header to incoming requests and sends a trace with timing details to X-Ray. Additionally, the random error function uses the X-Ray SDK to record the request ID and user information in annotations. The annotations are attached to the trace, and you can use them to locate the trace for a specific request.

The processor function gets the request ID from the CloudWatch Logs event, and uses the AWS SDK for JavaScript to search X-Ray for that request. It uses AWS SDK clients, which are instrumented with the X-Ray SDK, to download the trace and log stream. Then it stores them in the output bucket. The X-Ray SDK records these calls, and they appear as subsegments in the trace.

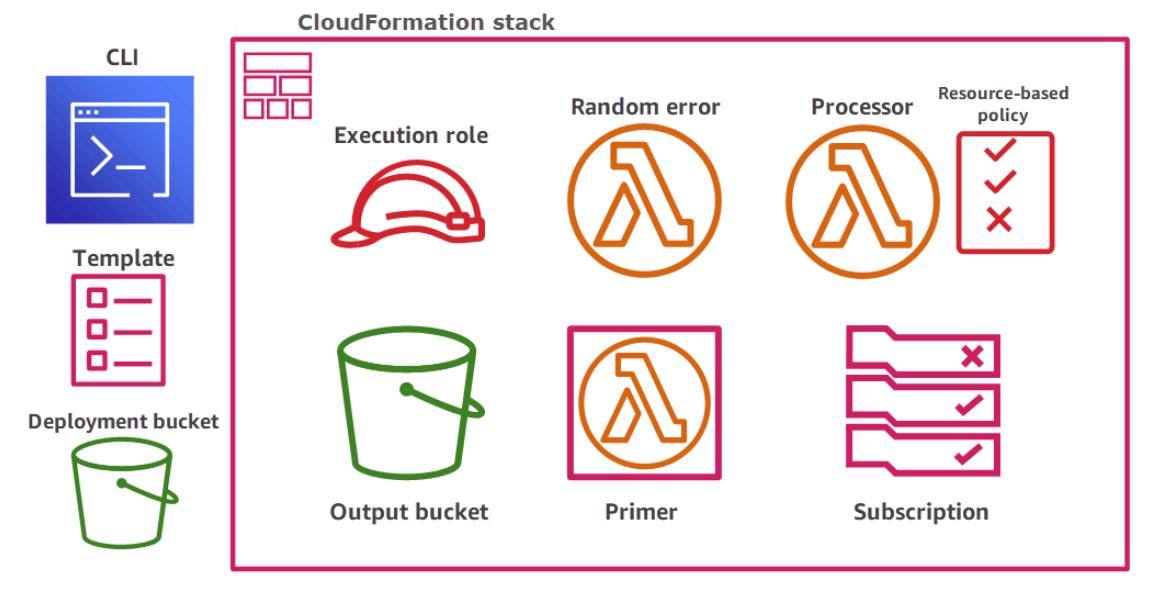
AWS CloudFormation template and additional resources

The application is implemented in two Node.js modules and deployed with an AWS CloudFormation template and shell scripts. The template creates the processor function, the random error function, and the following supporting resources.

- Execution role – An IAM role that grants the functions permission to access other AWS services.
- Primer function – An additional function that invokes the random error function to create a log group.
- Custom resource – An AWS CloudFormation custom resource that invokes the primer function during deployment to ensure that the log group exists.
- CloudWatch Logs subscription – A subscription for the log stream that triggers the processor function when the word ERROR is logged.
- Resource-based policy – A permission statement on the processor function that allows CloudWatch Logs to invoke it.

- Amazon S3 bucket – A storage location for output from the processor function.

View the [application template](#) on GitHub.



To work around a limitation of Lambda's integration with AWS CloudFormation, the template creates an additional function that runs during deployments. All Lambda functions come with a CloudWatch Logs log group that stores output from function executions. However, the log group isn't created until the function is invoked for the first time.

To create the subscription, which depends on the existence of the log group, the application uses a third Lambda function to invoke the random error function. The template includes the code for the primer function inline. An AWS CloudFormation custom resource invokes it during deployment. DependsOn properties ensure that the log stream and resource-based policy are created prior to the subscription.

List manager sample application for AWS Lambda

The list manager sample application demonstrates the use of AWS Lambda to process records in an Amazon Kinesis data stream. A Lambda event source mapping reads records from the stream in batches and invokes a Lambda function. The function uses information from the records to update documents in Amazon DynamoDB and stores the records it processes in Amazon Relational Database Service (Amazon RDS).

Clients send records to a Kinesis stream, which stores them and makes them available for processing. The Kinesis stream is used like a queue to buffer records until they can be processed. Unlike an Amazon SQS queue, records in a Kinesis stream are not deleted after they are processed, so multiple consumers can process the same data. Records in Kinesis are also processed in order, where queue items can be delivered out of order. Records are deleted from the stream after 7 days.

In addition to the function that processes events, the application includes a second function for performing administrative tasks on the database. Function code is available in the following files:

- Processor – [processor/index.js](#)
- Database admin – [dbadmin/index.js](#)

You can deploy the sample in a few minutes with the AWS CLI and AWS CloudFormation. To download, configure, and deploy it in your account, follow the instructions in the [README](#).

Sections

- [Architecture and event structure \(p. 1018\)](#)
- [Instrumentation with AWS X-Ray \(p. 1020\)](#)
- [AWS CloudFormation templates and additional resources \(p. 1020\)](#)

Architecture and event structure

The sample application uses the following AWS services:

- Kinesis – Receives events from clients and stores them temporarily for processing.
- AWS Lambda – Reads from the Kinesis stream and sends events to the function's handler code.
- DynamoDB – Stores lists generated by the application.
- Amazon RDS – Stores a copy of processed records in a relational database.
- AWS Secrets Manager – Stores the database password.
- Amazon VPC – Provides a private local network for communication between the function and database.

Pricing

Standard charges apply for each service.

The application processes JSON documents from clients that contain information necessary to update a list. It supports two types of list: tally and ranking. A *tally* contains values that are added to the current value for key if it exists. Each entry processed for a user increases the value of a key in the specified table.

The following example shows a document that increases the xp (experience points) value for a user's stats list.

Example record – Tally type

```
{
```

```
{
  "title": "stats",
  "user": "bill",
  "type": "tally",
  "entries": {
    "xp": 83
  }
}
```

A *ranking* contains a list of entries where the value is the order in which they are ranked. A ranking can be updated with different values that overwrite the current value, instead of incrementing it. The following example shows a ranking of favorite movies:

Example record – Ranking type

```
{
  "title": "favorite movies",
  "user": "mike",
  "type": "rank",
  "entries": {
    "blade runner": 1,
    "the empire strikes back": 2,
    "alien": 3
  }
}
```

A Lambda [event source mapping \(p. 131\)](#) read records from the stream in batches and invokes the processor function. The event that the function handler received contains an array of objects that each contain details about a record, such as when it was received, details about the stream, and an encoded representation of the original record document.

Example [events/kinesis.json](#) – Record

```
{
  "Records": [
    {
      "kinesis": {
        "kinesisSchemaVersion": "1.0",
        "partitionKey": "0",
        "sequenceNumber": "49598630142999655949899443842509554952738656579378741250",
        "data": "eyJ0aXRsZSI6ICJmYXvcml0ZSBtb3ZpZXMiLCaidXNlcii6ICJyZGx5c2N0IiwgInR5cGUIoAicmFuayIsICJlbnRyaWVzIjog
          "approximateArrivalTimestamp": 1570667770.615
      },
      "eventSource": "aws:kinesis",
      "eventVersion": "1.0",
      "eventID": "shardId-000000000000:49598630142999655949899443842509554952738656579378741250",
      "eventName": "aws:kinesis:record",
      "invokeIdentityArn": "arn:aws:iam::123456789012:role/list-manager-
processorRole-7FYXMPH7IUS",
      "awsRegion": "us-east-2",
      "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/list-manager-
stream-87B3XMPFLF1AZ"
    },
    ...
  ]
}
```

When it's decoded, the data contains a record. The function uses the record to update the user's list and an aggregate list that stores accumulated values across all users. It also stores a copy of the event in the application's database.

Instrumentation with AWS X-Ray

The application uses [AWS X-Ray \(p. 807\)](#) to trace function invocations and the calls that functions make to AWS services. X-Ray uses the trace data that it receives from functions to create a service map that helps you identify errors.

The Node.js function is configured for active tracing in the template, and is instrumented with the AWS X-Ray SDK for Node.js in code. The X-Ray SDK records a subsegment for each call made with an AWS SDK or MySQL client.

The function uses the AWS SDK for JavaScript in Node.js to read and write to two tables for each record. The primary table stores the current state of each combination of list name and user. The aggregate table stores lists that combine data from multiple users.

AWS CloudFormation templates and additional resources

The application is implemented in Node.js modules and deployed with an AWS CloudFormation template and shell scripts. The application template creates two functions, a Kinesis stream, DynamoDB tables and the following supporting resources.

Application resources

- Execution role – An IAM role that grants the functions permission to access other AWS services.
- Lambda event source mapping – Reads records from the data stream and invokes the function.

View the [application template](#) on GitHub.

A second template, [template-vpcrds.yml](#), creates the Amazon VPC and database resources. While it is possible to create all of the resources in one template, separating them makes it easier to clean up the application and allows the database to be reused with multiple applications.

Infrastructure resources

- VPC – A virtual private cloud network with private subnets, a route table, and a VPC endpoint that allows the function to communicate with DynamoDB without an internet connection.
- Database – An Amazon RDS database instance and a subnet group that connects it to the VPC.

Using Lambda with an AWS SDK

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples, and documentation that make it easier for developers to build applications in their preferred language.

SDK documentation	Code examples
AWS SDK for C++	AWS SDK for C++ code examples
AWS SDK for Go	AWS SDK for Go code examples
AWS SDK for Java	AWS SDK for Java code examples
AWS SDK for JavaScript	AWS SDK for JavaScript code examples
AWS SDK for Kotlin	AWS SDK for Kotlin code examples
AWS SDK for .NET	AWS SDK for .NET code examples
AWS SDK for PHP	AWS SDK for PHP code examples
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) code examples
AWS SDK for Ruby	AWS SDK for Ruby code examples
AWS SDK for Rust	AWS SDK for Rust code examples
AWS SDK for Swift	AWS SDK for Swift code examples

For examples specific to Lambda, see [Code examples for Lambda using AWS SDKs \(p. 1022\)](#).

Example availability

Can't find what you need? Request a code example by using the **Provide feedback** link at the bottom of this page.

Code examples for Lambda using AWS SDKs

The following code examples show how to use Lambda with an AWS software development kit (SDK).

Actions are code excerpts that show you how to call individual service functions.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Cross-service examples are sample applications that work across multiple AWS services.

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Get started

Hello Lambda

The following code examples show how to get started using Lambda.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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}");
        });
    }
}
```

- For API details, see [ListFunctions](#) in [AWS SDK for .NET API Reference](#).

[Go](#)

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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() {
    sdkConfig, err := config.LoadDefaultConfig(context.TODO())
    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(context.TODO(),
        &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)
        }
    }
}
```

- For API details, see [ListFunctions](#) in *AWS SDK for Go API Reference*.

Code examples

- [Actions for Lambda using AWS SDKs \(p. 1024\)](#)
 - [Create a Lambda function using an AWS SDK \(p. 1024\)](#)

- [Delete a Lambda function using an AWS SDK \(p. 1033\)](#)
- [Get a Lambda function using an AWS SDK \(p. 1037\)](#)
- [Invoke a Lambda function using an AWS SDK \(p. 1042\)](#)
- [List Lambda functions using an AWS SDK \(p. 1048\)](#)
- [Update Lambda function code using an AWS SDK \(p. 1052\)](#)
- [Update Lambda function configuration using an AWS SDK \(p. 1058\)](#)
- [Scenarios for Lambda using AWS SDKs \(p. 1062\)](#)
 - [Get started creating and invoking Lambda functions using an AWS SDK \(p. 1063\)](#)
- [Cross-service examples for Lambda using AWS SDKs \(p. 1122\)](#)
 - [Create an API Gateway REST API to track COVID-19 data \(p. 1123\)](#)
 - [Create a lending library REST API \(p. 1123\)](#)
 - [Create a messenger application with Step Functions \(p. 1124\)](#)
 - [Create a photo asset management application that lets users manage photos using labels \(p. 1125\)](#)
 - [Create a websocket chat application with API Gateway \(p. 1125\)](#)
 - [Invoke a Lambda function from a browser \(p. 1126\)](#)
 - [Use API Gateway to invoke a Lambda function \(p. 1126\)](#)
 - [Use Step Functions to invoke Lambda functions \(p. 1128\)](#)
 - [Use scheduled events to invoke a Lambda function \(p. 1129\)](#)

Actions for Lambda using AWS SDKs

The following code examples demonstrate how to perform individual Lambda actions with AWS SDKs. These excerpts call the Lambda API and are not intended to be run in isolation. Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

The following examples include only the most commonly used actions. For a complete list, see the [AWS Lambda API Reference](#).

Examples

- [Create a Lambda function using an AWS SDK \(p. 1024\)](#)
- [Delete a Lambda function using an AWS SDK \(p. 1033\)](#)
- [Get a Lambda function using an AWS SDK \(p. 1037\)](#)
- [Invoke a Lambda function using an AWS SDK \(p. 1042\)](#)
- [List Lambda functions using an AWS SDK \(p. 1048\)](#)
- [Update Lambda function code using an AWS SDK \(p. 1052\)](#)
- [Update Lambda function configuration using an AWS SDK \(p. 1058\)](#)

Create a Lambda function using an AWS SDK

The following code examples show how to create a Lambda function.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <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 response = await
    _lambdaService.CreateFunctionAsync(createFunctionRequest);
    return response.FunctionArn;
}
```

- For API details, see [CreateFunction](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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.
#ifndef USE_CPP_LAMBDA_FUNCTION
    request.SetRuntime(Aws::Lambda::Model::Runtime::provided_a12);
    request.SetTimeout(15);
    request.SetMemorySize(128);

    // Assume the AWS Lambda function was built in Docker with same
    // architecture
    // as this code.
#endif
#ifdef __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)
#ifndef
    request.SetRuntime(Aws::Lambda::Model::Runtime::python3_8);
#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);
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;
}
```

- For API details, see [CreateFunction](#) in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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(functionName string, handlerName  
string,  
iamRoleArn *string, zipPackage *bytes.Buffer) types.State {  
var state types.State  
_, err := wrapper.LambdaClient.CreateFunction(context.TODO(),  
&lambda.CreateFunctionInput{  
    Code:      &types.FunctionCode{ZipFile: zipPackage.Bytes()},  
    FunctionName: aws.String(functionName),  
    Role:      iamRoleArn,  
    Handler:   aws.String(handlerName),  
    Publish:   true,  
    Runtime:   types.RuntimePython38,  
})  
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(context.TODO(), &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  
}
```

- For API details, see [CreateFunction](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public static void createLambdaFunction(LambdaClient awsLambda,
                                         String functionName,
                                         String filePath,
                                         String role,
                                         String handler) {

    try {
        LambdaWaiter waiter = awsLambda.waiter();
        InputStream is = new FileInputStream(filePath);
        SdkBytes fileToUpload = SdkBytes.fromInputStream(is);

        FunctionCode code = FunctionCode.builder()
            .zipFile(fileToUpload)
            .build();

        CreateFunctionRequest functionRequest = CreateFunctionRequest.builder()
            .functionName(functionName)
            .description("Created by the Lambda Java API")
            .code(code)
            .handler(handler)
            .runtime(Runtime.JAVA8)
            .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);
        System.out.println("The function ARN is " +
        functionResponse.functionArn());

    } catch(LambdaException | FileNotFoundException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
}
```

- For API details, see [CreateFunction](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const createFunction = async (funcName, roleArn) => {
```

```
const client = createClientForDefaultRegion(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);
};
```

- For API details, see [CreateFunction](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

Note

This is prerelease documentation for a feature in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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.Java8
    }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
        return functionResponse.functionArn
    }
}
```

- For API details, see [CreateFunction](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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",
        ]);
    });
}
```

- For API details, see [CreateFunction](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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='python3.8',
    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
```

- For API details, see [CreateFunction](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @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. This
  #                      must include the file name and the function name.
  # @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:,
      handler: handler_name,
      runtime: "ruby2.7",
      code: {
        zip_file: deployment_package
      },
      environment: {
        variables: {
          "LOG_LEVEL" => "info"
        }
      }
    })
  end
end
```

```
@lambda_client.wait_until(:function_active_v2, { 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
```

- For API details, see [CreateFunction](#) in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

Note

This documentation is for an SDK in developer preview release. The SDK is subject to change and is not recommended for use in production.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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.
```

- For API details, see [CreateFunction](#) in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Delete a Lambda function using an AWS SDK

The following code examples show how to delete a Lambda function.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <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.StatusCode == System.Net.HttpStatusCode.NoContent;
}
```

- For API details, see [DeleteFunction](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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;  
}
```

- For API details, see [DeleteFunction](#) in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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(functionName string) {  
    _, err := wrapper.LambdaClient.DeleteFunction(context.TODO(),  
        &lambda.DeleteFunctionInput{  
            FunctionName: aws.String(functionName),  
        })  
    if err != nil {  
        log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)  
    }  
}
```

- For API details, see [DeleteFunction](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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);
    }
}
```

- For API details, see [DeleteFunction](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const deleteFunction = (funcName) => {
    const client = createClientForDefaultRegion(LambdaClient);
    const command = new DeleteFunctionCommand({ FunctionName: funcName });
    return client.send(command);
};
```

- For API details, see [DeleteFunction](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

Note

This is prerelease documentation for a feature in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun delLambdaFunction(myFunctionName: String) {

    val request = DeleteFunctionRequest {
        functionName = myFunctionName
    }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- For API details, see [DeleteFunction](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public function deleteFunction($functionName)
{
    return $this->lambdaClient->deleteFunction([
        'FunctionName' => $functionName,
    ]);
}
```

- For API details, see [DeleteFunction](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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
```

- For API details, see [DeleteFunction](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
```

```
@lambda_client = Aws::Lambda::Client.new
@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
```

- For API details, see [DeleteFunction](#) in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

Note

This documentation is for an SDK in developer preview release. The SDK is subject to change and is not recommended for use in production.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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.
```

- For API details, see [DeleteFunction](#) in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Get a Lambda function using an AWS SDK

The following code examples show how to get a Lambda function.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <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;
}
```

- For API details, see [GetFunction](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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;
}
```

```
}
```

- For API details, see [GetFunction](#) in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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(functionName string) types.State {  
    var state types.State  
    funcOutput, err := wrapper.LambdaClient.GetFunction(context.TODO(),  
&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  
}
```

- For API details, see [GetFunction](#) in *AWS SDK for Go API Reference*.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const getFunction = (funcName) => {  
    const client = createClientForDefaultRegion(LambdaClient);  
    const command = new GetFunctionCommand({ FunctionName: funcName });  
    return client.send(command);  
};
```

- For API details, see [GetFunction](#) in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public function getFunction($functionName)
{
    return $this->lambdaClient->getFunction([
        'FunctionName' => $functionName,
    ]);
}
```

- For API details, see [GetFunction](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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
```

- For API details, see [GetFunction](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @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:
      }
    )
    rescue Aws::Lambda::Errors::ResourceNotFoundException => e
      @logger.debug("Could not find function: #{function_name}:\n #{e.message}")
      nil
  end
```

- For API details, see [GetFunction](#) in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

Note

This documentation is for an SDK in developer preview release. The SDK is subject to change and is not recommended for use in production.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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_lmdinparamvalueex.
    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.
```

- For API details, see [GetFunction](#) in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Invoke a Lambda function using an AWS SDK

The following code examples show how to invoke a Lambda function.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <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;
}
```

- For API details, see [Invoke](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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;  
}
```

- For API details, see [Invoke](#) in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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(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(context.TODO(),
&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
}
```

- For API details, see [Invoke](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public static void invokeFunction(LambdaClient awsLambda, String functionName)
{
    InvokeResponse res = null ;
    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) ;

        // Setup an InvokeRequest.
        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);
    }
}
```

- For API details, see [Invoke](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const invoke = async (funcName, payload) => {
  const client = createClientForDefaultRegion(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 };
};
```

- For API details, see [Invoke](#) in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

Note

This is prerelease documentation for a feature in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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}")
    }
}
```

- For API details, see [Invoke](#) in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public function invoke($functionName, $params, $logType = 'None')
{
```

```
        return $this->lambdaClient->invoke([
            'FunctionName' => $functionName,
            'Payload' => json_encode($params),
            'LogType' => $logType,
        ]);
    }
```

- For API details, see [Invoke](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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
```

- For API details, see [Invoke](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class LambdaWrapper
  attr_accessor :lambda_client
```

```

def initialize
  @lambda_client = Aws::Lambda::Client.new
  @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: }
  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

```

- For API details, see [Invoke](#) in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

Note

This documentation is for an SDK in developer preview release. The SDK is subject to change and is not recommended for use in production.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```

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.
```

- For API details, see [Invoke](#) in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

List Lambda functions using an AWS SDK

The following code examples show how to list Lambda functions.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <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.Paginator.ListFunctions(new ListFunctionsRequest());  
    await foreach (var function in functionPaginator.Functions)  
    {  
        functionList.Add(function);  
    }  
  
    return functionList;  
}
```

- For API details, see [ListFunctions](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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());
```

- For API details, see [ListFunctions](#) in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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(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(context.TODO())
    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
}
```

- For API details, see [ListFunctions](#) in *AWS SDK for Go API Reference*.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const listFunctions = async () => {
    const client = createClientForDefaultRegion(LambdaClient);
    const command = new ListFunctionsCommand({});

    return client.send(command);
};
```

- For API details, see [ListFunctions](#) in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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,
    ]);
}
```

- For API details, see [ListFunctions](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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:
            funcPaginator = self.lambda_client.getPaginator('list_functions')
            for funcPage in funcPaginator.paginate():
                for func in funcPage['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
```

- For API details, see [ListFunctions](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @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 executing #{function_name}:\n #{e.message}")
end
```

- For API details, see [ListFunctions](#) in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

Note

This documentation is for an SDK in developer preview release. The SDK is subject to change and is not recommended for use in production.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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.
```

- For API details, see [ListFunctions](#) in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Update Lambda function code using an AWS SDK

The following code examples show how to update Lambda function code.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```

/// <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}.");
}

```

- For API details, see [UpdateFunctionCode](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```

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);
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;
}
```

- For API details, see [UpdateFunctionCode](#) in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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(functionName string, zipPackage
    *bytes.Buffer) types.State {
    var state types.State
    _, err := wrapper.LambdaClient.UpdateFunctionCode(context.TODO(),
    &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(context.TODO(), &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
}
```

- For API details, see [UpdateFunctionCode](#) in *AWS SDK for Go API Reference*.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = createClientForDefaultRegion(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);
};
```

- For API details, see [UpdateFunctionCode](#) in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public function updateFunctionCode($functionName, $s3Bucket, $s3Key)
{
    return $this->lambdaClient->updateFunctionCode([
        'FunctionName' => $functionName,
        'S3Bucket' => $s3Bucket,
        'S3Key' => $s3Key,
    ]);
}
```

- For API details, see [UpdateFunctionCode](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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

```

- For API details, see [UpdateFunctionCode](#) in AWS SDK for Python (Boto3) API Reference.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```

class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @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,
      zip_file: deployment_package
    )
    @lambda_client.wait_until(:function_updated_v2, { function_name: }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  end
end

```

```
    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
```

- For API details, see [UpdateFunctionCode](#) in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

Note

This documentation is for an SDK in developer preview release. The SDK is subject to change and is not recommended for use in production.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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_lmdinvaliddodesigex.
    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.
```

- For API details, see [UpdateFunctionCode](#) in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Update Lambda function configuration using an AWS SDK

The following code examples show how to update Lambda function configuration.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/// <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;
}
```

- For API details, see [UpdateFunctionConfiguration](#) in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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;
}
```

- For API details, see [UpdateFunctionConfiguration](#) in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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(functionName string,
    envVars map[string]string) {
    _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(context.TODO(),
        &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)
    }
}
```

- For API details, see [UpdateFunctionConfiguration](#) in *AWS SDK for Go API Reference*.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`.${dirname}functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  return client.send(command);
};
```

- For API details, see [UpdateFunctionConfiguration](#) in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
public function updateFunctionConfiguration($functionName, $handler,
$environment = '')
{
    return $this->lambdaClient->updateFunctionConfiguration([
        'FunctionName' => $functionName,
        'Handler' => "$handler.lambda_handler",
        'Environment' => $environment,
    ]);
}
```

- For API details, see [UpdateFunctionConfiguration](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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.
:returns: 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
```

- For API details, see [UpdateFunctionConfiguration](#) in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @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:,
      environment: {
        variables: {
          "LOG_LEVEL" => log_level
        }
      }
    })
    @lambda_client.wait_until(:function_updated_v2, { 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
  end
end
```

```
end
```

- For API details, see [UpdateFunctionConfiguration](#) in *AWS SDK for Ruby API Reference*.

SAP ABAP

SDK for SAP ABAP

Note

This documentation is for an SDK in developer preview release. The SDK is subject to change and is not recommended for use in production.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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_memoriesize = 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_lmdinvalparamvalueex.  
                    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.
```

- For API details, see [UpdateFunctionConfiguration](#) in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Scenarios for Lambda using AWS SDKs

The following code examples show you how to implement common scenarios in Lambda with AWS SDKs. These scenarios show you how to accomplish specific tasks by calling multiple functions within Lambda.

Each scenario includes a link to GitHub, where you can find instructions on how to set up and run the code.

Examples

- [Get started creating and invoking Lambda functions using an AWS SDK \(p. 1063\)](#)

Get started creating and invoking Lambda functions using an AWS SDK

The following code examples show how to:

- Create an IAM role and Lambda function, then upload handler code.
- Invoke the function with a single parameter and get results.
- Update the function code and configure with an environment variable.
- Invoke the function with new parameters and get results. Display the returned execution log.
- List the functions for your account, then clean up resources.

For more information, see [Create a Lambda function with the console](#).

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create methods that perform Lambda actions.

```
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 response = await
_lambdaService.CreateFunctionAsync(createFunctionRequest);
        return response.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.StatusCode == 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.Paginator.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;
}

}

```

Create a function that runs the scenario.

```

global using LambdaActions;
global using LambdaScenarioCommon;
global using Amazon.Lambda;
global using Amazon.IdentityManagement;
global using System.Threading.Tasks;
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.IdentityManagement;
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.WriteLine("Enter the first value: ");
            value = Console.ReadLine();
        }
        while (value == string.Empty);

        string? value2;
        do
        {
            Console.WriteLine("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>
    ///<summary>
    ///</summary>
    public void PressEnter()
    {
        Console.Write("\nPress <Enter> to continue. ");
        _ = Console.ReadLine();
        Console.WriteLine();
    }

    ///<summary>
    ///<summary>
    ///</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>
    ///<summary>
    ///</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>
    ///<summary>
    ///</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();
    }
}

```

Define a Lambda handler that increments a number.

```

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.DefaultLambdaJsonSerializer))  
  
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;  
        }  
    }  
}
```

Define a second Lambda handler that performs arithmetic operations.

```
using Amazon.Lambda.Core;  
using System.Diagnostics;  
  
// Assembly attribute to enable the Lambda function's JSON input to be converted  
// into a .NET class.  
[assembly:  
LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer))]  
  
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;
}
```

- For API details, see the following topics in *AWS SDK for .NET API Reference*.
 - [CreateFunction](#)
 - [DeleteFunction](#)
 - [GetFunction](#)
 - [Invoke](#)
 - [ListFunctions](#)
 - [UpdateFunctionCode](#)
 - [UpdateFunctionConfiguration](#)

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
///! Get started with functions scenario.
/*!
 \sa getStartedWithFunctionsScenario()
 \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.
#ifndef USE_CPP_LAMBDA_FUNCTION
    request.SetRuntime(Aws::Lambda::Model::Runtime::provided_a12);
    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_8);
#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);
    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().GetErrorCode() ==
        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::JsonValue jsonPayload;
    jsonPayload.WithString("action", "increment");
    jsonPayload.WithInteger("number", increment);
    if (invokeLambdaFunction(jsonPayload, Aws::Lambda::Model::LogType::Tail,
        invokeResult, client)) {
        Aws::Utils::JsonValue jsonValue(invokeResult.GetPayload());
        Aws::Map<Aws::String, Aws::Utils::Json::JsonValue> 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);
    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::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::JsonValue jsonValue(calculatedResult.GetPayload());
        Aws::Map<Aws::String, Aws::Utils::Json::JsonValue> 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.
    {
        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;
        }
    }

    // 10. Delete the IAM role.
    return deleteIamRole(clientConfig);
}

//! Routine which invokes a Lambda function and returns the result.
/*!
    \\sa invokeLambdaFunction()
    \\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::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;
}

```

- For API details, see the following topics in *AWS SDK for C++ API Reference*.
 - [CreateFunction](#)
 - [DeleteFunction](#)
 - [GetFunction](#)
 - [Invoke](#)
 - [ListFunctions](#)
 - [UpdateFunctionCode](#)
 - [UpdateFunctionConfiguration](#)

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create an interactive scenario that shows you how to get started with Lambda functions.

```

// 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() {
    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()
    funcName := scenario.CreateFunction(role)
    scenario.InvokeIncrement(funcName)
    scenario.UpdateFunction(funcName)
    scenario.InvokeCalculator(funcName)
    scenario.ListFunctions()
    scenario.Cleanup(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() *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(context.TODO(), &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(context.TODO(), &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(context.TODO(), &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(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(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(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(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(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(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(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(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(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() {
count := scenario.questioner.AskInt(
    "Let's list functions for your account. How many do you want to see?", demotools.NotEmpty{})
functions := scenario.functionWrapper.ListFunctions(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(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(context.TODO(),
        &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(context.TODO(), &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)
        }
    }
}
}

```

```

        *policy.PolicyArn, *role.RoleName, err)
    }
}
_, err = iamClient.DeleteRole(context.TODO(), &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(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.")
}
}

```

Create a struct that wraps individual Lambda actions.

```

// 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(functionName string) types.State {
var state types.State
funcOutput, err := wrapper.LambdaClient.GetFunction(context.TODO(),
&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(functionName string, handlerName
string,
iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
var state types.State
_, err := wrapper.LambdaClient.CreateFunction(context.TODO(),
&lambda.CreateFunctionInput{
    Code:      &types.FunctionCode{ZipFile: zipPackage.Bytes()},
    FunctionName: aws.String(functionName),
    Role:      iamRoleArn,
}

```

```

Handler:      aws.String(handlerName),
Publish:     true,
Runtime:      types.RuntimePython38,
})
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(context.TODO(), &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(functionName string, zipPackage
*bytes.Buffer) types.State {
var state types.State
_, err := wrapper.LambdaClient.UpdateFunctionCode(context.TODO(),
&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(context.TODO(), &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(functionName string,
envVars map[string]string) {

```

```

    _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(context.TODO(),
    &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(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(context.TODO())
        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(functionName string) {
    _, err := wrapper.LambdaClient.DeleteFunction(context.TODO(),
    &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(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(context.TODO(),
    &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"`
}

```

Create a struct that implements functions to help run the scenario.

```

// IScenarioHelper abstracts I/O and wait functions from a scenario so that they
// can be mocked for unit testing.
type IScearioHelper 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
}
```

Define a Lambda handler that increments a number.

```
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
```

Define a second Lambda handler that performs arithmetic operations.

```
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
```

- For API details, see the following topics in *AWS SDK for Go API Reference*.
 - [CreateFunction](#)
 - [DeleteFunction](#)
 - [GetFunction](#)
 - [Invoke](#)
 - [ListFunctions](#)
 - [UpdateFunctionCode](#)
 - [UpdateFunctionConfiguration](#)

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/*
 * 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 = "\n" +
            "Usage:\n" +
            "      <functionName> <filePath> <role> <handler> <bucketName> <key> \n
\n" +
            "Where:\n" +
            "      functionName - The name of the Lambda function. \n"+
            "      filePath - The path to the .zip or .jar where the code is located.
\n"+
            "      role - The AWS Identity and Access Management (IAM) service role
that has Lambda permissions. \n"+
            "      handler - The fully qualified method name (for example,
example.Handler::handleRequest). \n"+
            "      bucketName - The Amazon Simple Storage Service (Amazon S3) bucket
name that contains the .zip or .jar used to update the Lambda function's code.
\n"+
            "      key - The Amazon S3 key name that represents the .zip or .jar (for
example, LambdaHello-1.0-SNAPSHOT.jar). " ;

        if (args.length != 6) {
            System.out.println(usage);
            System.exit(1);
        }

        String functionName = args[0];
        String filePath = args[1];
```

```
String role = args[2];
String handler = args[3];
String bucketName = args[4];
String key = args[5];

Region region = Region.US_WEST_2;
LambdaClient awsLambda = LambdaClient.builder()
    .region(region)
    .credentialsProvider(ProfileCredentialsProvider.create())
    .build();

System.out.println(DASHES);
System.out.println("Welcome to the AWS Lambda example scenario.");
System.out.println(DASHES);

System.out.println(DASHES);
System.out.println("1. Create an AWS Lambda function.");
String funArn = createLambdaFunction(awsLambda, functionName, filePath,
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();
}

public static String createLambdaFunction(LambdaClient awsLambda,
String functionName,
```

```

        String filePath,
        String role,
        String handler) {

    try {
        LambdaWaiter waiter = awsLambda.waiter();
        InputStream is = new FileInputStream(filePath);
        SdkBytes fileToUpload = SdkBytes.fromInputStream(is);

        FunctionCode code = FunctionCode.builder()
            .zipFile(fileToUpload)
            .build();

        CreateFunctionRequest functionRequest = CreateFunctionRequest.builder()
            .functionName(functionName)
            .description("Created by the Lambda Java API")
            .code(code)
            .handler(handler)
            .runtime(Runtime.JAVA8)
            .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 | FileNotFoundException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
    return "";
}

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);
    }
}

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);
    }

    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);
        }
    }

    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);
    }
}

    public static void updateFunctionConfiguration(LambdaClient awsLambda, String
functionName, String handler ){
    try {
        UpdateFunctionConfigurationRequest configurationRequest =
UpdateFunctionConfigurationRequest.builder()

```

```
.functionName(functionName)
.handler(handler)
.runtime(Runtime.JAVA11 )
.build();

awsLambda.updateFunctionConfiguration(configurationRequest);

} catch(LambdaException e) {
    System.err.println(e.getMessage());
    System.exit(1);
}

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);
    }
}
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
 - [CreateFunction](#)
 - [DeleteFunction](#)
 - [GetFunction](#)
 - [Invoke](#)
 - [ListFunctions](#)
 - [UpdateFunctionCode](#)
 - [UpdateFunctionConfiguration](#)

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create an AWS Identity and Access Management (IAM) role that grants Lambda permission to write to logs.

```
log(`Creating role ${NAME_ROLE_LAMBDA}...`);

const response = await createRole({
    AssumeRolePolicyDocument: parseString({
        Version: "2012-10-17",
        Statement: [
            {
                Effect: "Allow",
                Principal: {
                    Service: "lambda.amazonaws.com",

```

```

        ],
        Action: "sts:AssumeRole",
    ],
},
}),
RoleName: 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);
};

```

Create a Lambda function and upload handler code.

```

const createFunction = async (funcName, roleArn) => {
    const client = createClientForDefaultRegion(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);
};

```

Invoke the function with a single parameter and get results.

```

const invoke = async (funcName, payload) => {
    const client = createClientForDefaultRegion(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 };
};

```

Update the function code and configure its Lambda environment with an environment variable.

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = createClientForDefaultRegion(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 = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  return client.send(command);
};
```

List the functions for your account.

```
const listFunctions = async () => {
  const client = createClientForDefaultRegion(LambdaClient);
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

Delete the IAM role and the Lambda function.

```
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);
};

const deleteFunction = (funcName) => {
  const client = createClientForDefaultRegion(LambdaClient);
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [CreateFunction](#)

- [DeleteFunction](#)
- [GetFunction](#)
- [Invoke](#)
- [ListFunctions](#)
- [UpdateFunctionCode](#)
- [UpdateFunctionConfiguration](#)

Kotlin

SDK for Kotlin

Note

This is prerelease documentation for a feature in preview release. It is subject to change.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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.Java8
    }

    // Create a Lambda function using a waiter
    LambdaClient { region = "us-west-2" }.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-west-2" }.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-west-2" }.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-west-2" }.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-west-2" }.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.Java11
    }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        awsLambda.updateFunctionConfiguration(configurationRequest)
    }
}

suspend fun delFunction(myFunctionName: String) {

```

```
    val request = DeleteFunctionRequest {
        functionName = myFunctionName
    }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- For API details, see the following topics in *AWS SDK for Kotlin API reference*.
 - [CreateFunction](#)
 - [DeleteFunction](#)
 - [GetFunction](#)
 - [Invoke](#)
 - [ListFunctions](#)
 - [UpdateFunctionCode](#)
 - [UpdateFunctionConfiguration](#)

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
namespace Lambda;

use Aws\S3\S3Client;
use GuzzleHttp\Psr7\Stream;
use Iam\IAMService;

class GettingStartedWithLambda
{
    public function run()
    {
        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";
$bucketName = "test-example-bucket-$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";
}
}

```

- For API details, see the following topics in *AWS SDK for PHP API Reference*.
 - [CreateFunction](#)
 - [DeleteFunction](#)
 - [GetFunction](#)
 - [Invoke](#)
 - [ListFunctions](#)
 - [UpdateFunctionCode](#)
 - [UpdateFunctionConfiguration](#)

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Define a Lambda handler that increments a number.

```

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
```

Define a second Lambda handler that performs arithmetic operations.

```
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
```

Create functions that wrap Lambda actions.

```

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']
                )
        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.
    """

```

```

:rtype: 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='python3.8',
        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:
        funcPaginator = self.lambda_client.getPaginator('list_functions')
        for funcPage in funcPaginator.paginate():
            for func in funcPage['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

```

Create a function that runs the scenario.

```

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.getLogger(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']} "
          f"{action_params['y']} "
          f"resulted in {json.loads(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!")

```

- For API details, see the following topics in *AWS SDK for Python (Boto3) API Reference*.
 - [CreateFunction](#)
 - [DeleteFunction](#)
 - [GetFunction](#)
 - [Invoke](#)
 - [ListFunctions](#)

- [UpdateFunctionCode](#)
- [UpdateFunctionConfiguration](#)

Ruby

SDK for Ruby

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Set up pre-requisite IAM permissions for a Lambda function capable of writing logs.

```
# 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)  
    role_policy = {  
        'Version': "2012-10-17",  
        'Statement': [  
            {  
                'Effect': "Allow",  
                'Principal': {  
                    'Service': "lambda.amazonaws.com"  
                },  
                'Action': "sts:AssumeRole"  
            }  
        ]  
    }  
    case action  
    when "create"  
        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  
            }  
        )  
        $iam_client.wait_until(:role_exists, { role_name: iam_role_name }) do |w|  
            w.max_attempts = 5  
            w.delay = 5  
        end  
        @logger.debug("Successfully created IAM role: #{role['role']['arn']}")  
        @logger.debug("Enforcing a 10-second sleep to allow IAM role to activate  
fully.")  
        sleep(10)  
        return role, role_policy.to_json  
    when "destroy"  
        $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}")
else
    raise "Incorrect action provided. Must provide 'create' or 'destroy'"
end
rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error creating role or attaching policy:\n#{e.message}")
end

```

Define a Lambda handler that increments a number provided as an invocation parameter.

```

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

```

Zip your Lambda function into a deployment package.

```

# Creates a Lambda deployment package in .zip format.
# This zip can be passed directly as a string to Lambda when creating the
function.
#
# @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) {
        |zipfile|
        zipfile.add("lambda_function.rb", "#{source_file}.rb")
    }
    @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
```

Create a new Lambda function.

```
# 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 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: }) 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
```

Invoke your Lambda function with optional runtime parameters.

```
# 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: }
  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
```

Update your Lambda function's configuration to inject a new environment variable.

```
# 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:,
    environment: {
      variables: {
        "LOG_LEVEL" => log_level
      }
    }
  })
  @lambda_client.wait_until(:function_updated_v2, { 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
```

Update your Lambda function's code with a different deployment package containing different code.

```
# 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:,
    zip_file: deployment_package
  )
  @lambda_client.wait_until(:function_updated_v2, { 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
```

List all existing Lambda functions using the built-in paginator.

```
# 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 executing #{function_name}:\n #{e.message}")
end
```

Delete a specific Lambda function.

```
# 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:
  )
  print "Done!".green
rescue Aws::Lambda::Errors::ServiceException => e
  @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
end
```

- For API details, see the following topics in *AWS SDK for Ruby API Reference*.
 - [CreateFunction](#)
 - [DeleteFunction](#)
 - [GetFunction](#)
 - [Invoke](#)
 - [ListFunctions](#)
 - [UpdateFunctionCode](#)
 - [UpdateFunctionConfiguration](#)

SAP ABAP

SDK for SAP ABAP

Note

This documentation is for an SDK in developer preview release. The SDK is subject to change and is not recommended for use in production.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
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'
    ).
    MESSAGE 'IAM role created.' TYPE 'I'.
    WAIT UP TO 10 SECONDS.           " Make sure that the IAM role is
ready for use."
    CATCH /aws1/cx_iamentityalrdyexecex.
        MESSAGE 'IAM role already exists.' TYPE 'E'.
    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 = lo_create_role_output->get_role( )->get_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_lmdcodestorageexcex.
        MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
    CATCH /aws1/cx_lmdinvalparamvalueex.
        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_lmdinparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
    CATCH /aws1/cx_lmdinrequestcontex.
        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_lmdcodestorageexcex.
        MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
    CATCH /aws1/cx_lmdinparamvalueex.
        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_lmdinparamvalueex.
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_lmdinparamvalueex.
MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
CATCH /aws1/cx_lmdinrequestcontex.
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_lmdinparamvalueex.
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_lmdinparamvalueex.
MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
CATCH /aws1/cx_lmdresourcenotfoundex.
MESSAGE 'The requested resource does not exist.' TYPE 'E'.
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 'E'.
    CATCH /aws1/cx_iamplynottachableex.
        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 'E'.
    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.

```

- For API details, see the following topics in *AWS SDK for SAP ABAP API reference*.
 - [CreateFunction](#)
 - [DeleteFunction](#)
 - [GetFunction](#)
 - [Invoke](#)
 - [ListFunctions](#)
 - [UpdateFunctionCode](#)
 - [UpdateFunctionConfiguration](#)

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Cross-service examples for Lambda using AWS SDKs

The following sample applications use AWS SDKs to combine Lambda with other AWS services. Each example includes a link to GitHub, where you can find instructions on how to set up and run the application.

Examples

- [Create an API Gateway REST API to track COVID-19 data \(p. 1123\)](#)
- [Create a lending library REST API \(p. 1123\)](#)
- [Create a messenger application with Step Functions \(p. 1124\)](#)
- [Create a photo asset management application that lets users manage photos using labels \(p. 1125\)](#)
- [Create a websocket chat application with API Gateway \(p. 1125\)](#)
- [Invoke a Lambda function from a browser \(p. 1126\)](#)
- [Use API Gateway to invoke a Lambda function \(p. 1126\)](#)
- [Use Step Functions to invoke Lambda functions \(p. 1128\)](#)
- [Use scheduled events to invoke a Lambda function \(p. 1129\)](#)

Create an API Gateway REST API to track COVID-19 data

The following code example shows how to create a REST API that simulates a system to track daily cases of COVID-19 in the United States, using fictional data.

Python

SDK for Python (Boto3)

Shows how to use AWS Chalice with the AWS SDK for Python (Boto3) to create a serverless REST API that uses Amazon API Gateway, AWS Lambda, and Amazon DynamoDB. The REST API simulates a system that tracks daily cases of COVID-19 in the United States, using fictional data. Learn how to:

- Use AWS Chalice to define routes in Lambda functions that are called to handle REST requests that come through API Gateway.
- Use Lambda functions to retrieve and store data in a DynamoDB table to serve REST requests.
- Define table structure and security role resources in an AWS CloudFormation template.
- Use AWS Chalice and CloudFormation to package and deploy all necessary resources.
- Use CloudFormation to clean up all created resources.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- API Gateway
- AWS CloudFormation
- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Create a lending library REST API

The following code example shows how to create a lending library where patrons can borrow and return books by using a REST API backed by an Amazon Aurora database.

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) with the Amazon Relational Database Service (Amazon RDS) API and AWS Chalice to create a REST API backed by an Amazon Aurora database. The web service is fully serverless and represents a simple lending library where patrons can borrow and return books. Learn how to:

- Create and manage a serverless Aurora database cluster.
- Use AWS Secrets Manager to manage database credentials.
- Implement a data storage layer that uses Amazon RDS to move data into and out of the database.
- Use AWS Chalice to deploy a serverless REST API to Amazon API Gateway and AWS Lambda.
- Use the Requests package to send requests to the web service.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- API Gateway
- Lambda
- Amazon RDS
- Secrets Manager

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Create a messenger application with Step Functions

The following code example shows how to create an AWS Step Functions messenger application that retrieves message records from a database table.

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) with AWS Step Functions to create a messenger application that retrieves message records from an Amazon DynamoDB table and sends them with Amazon Simple Queue Service (Amazon SQS). The state machine integrates with an AWS Lambda function to scan the database for unsent messages.

- Create a state machine that retrieves and updates message records from an Amazon DynamoDB table.
- Update the state machine definition to also send messages to Amazon Simple Queue Service (Amazon SQS).
- Start and stop state machine runs.
- Connect to Lambda, DynamoDB, and Amazon SQS from a state machine by using service integrations.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB

- Lambda
- Amazon SQS
- Step Functions

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Create a photo asset management application that lets users manage photos using labels

The following code example shows how to create a serverless application that lets users manage photos using labels.

Java

SDK for Java 2.x

Shows how to develop a photo asset management application that lets users manage photos using labels.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Create a websocket chat application with API Gateway

The following code example shows how to create a chat application that is served by a websocket API built on Amazon API Gateway.

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) with Amazon API Gateway V2 to create a websocket API that integrates with AWS Lambda and Amazon DynamoDB.

- Create a websocket API served by API Gateway.
- Define a Lambda handler that stores connections in DynamoDB and posts messages to other chat participants.

- Connect to the websocket chat application and send messages with the Websockets package.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Invoke a Lambda function from a browser

The following code example shows how to invoke an AWS Lambda function from a browser.

JavaScript

SDK for JavaScript (v2)

You can create a browser-based application that uses an AWS Lambda function to update an Amazon DynamoDB table with user selections.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Lambda

SDK for JavaScript (v3)

You can create a browser-based application that uses an AWS Lambda function to update an Amazon DynamoDB table with user selections. This app uses AWS SDK for JavaScript v3.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Use API Gateway to invoke a Lambda function

The following code examples show how to create an AWS Lambda function invoked by Amazon API Gateway.

Java

SDK for Java 2.x

Shows how to create an AWS Lambda function by using the Lambda Java runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create a Lambda function invoked by Amazon API Gateway that scans an Amazon DynamoDB table for work anniversaries and uses Amazon Simple Notification Service (Amazon SNS) to send a text message to your employees that congratulates them at their one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Shows how to create an AWS Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create a Lambda function invoked by Amazon API Gateway that scans an Amazon DynamoDB table for work anniversaries and uses Amazon Simple Notification Service (Amazon SNS) to send a text message to your employees that congratulates them at their one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

This example is also available in the [AWS SDK for JavaScript v3 developer guide](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

This example shows how to create and use an Amazon API Gateway REST API that targets an AWS Lambda function. The Lambda handler demonstrates how to route based on HTTP methods; how to get data from the query string, header, and body; and how to return a JSON response.

- Deploy a Lambda function.
- Create an API Gateway REST API.
- Create a REST resource that targets the Lambda function.
- Grant permission to let API Gateway invoke the Lambda function.

- Use the Requests package to send requests to the REST API.
- Clean up all resources created during the demo.

This example is best viewed on GitHub. For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- API Gateway
- Lambda

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Use Step Functions to invoke Lambda functions

The following code examples show how to create an AWS Step Functions state machine that invokes AWS Lambda functions in sequence.

Java

SDK for Java 2.x

Shows how to create an AWS serverless workflow by using AWS Step Functions and the AWS SDK for Java 2.x. Each workflow step is implemented using an AWS Lambda function.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Lambda
- Amazon SES
- Step Functions

JavaScript

SDK for JavaScript (v3)

Shows how to create an AWS serverless workflow by using AWS Step Functions and the AWS SDK for JavaScript. Each workflow step is implemented using an AWS Lambda function.

Lambda is a compute service that enables you to run code without provisioning or managing servers. Step Functions is a serverless orchestration service that lets you combine Lambda functions and other AWS services to build business-critical applications.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

This example is also available in the [AWS SDK for JavaScript v3 developer guide](#).

Services used in this example

- DynamoDB
- Lambda

- Amazon SES
- Step Functions

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Use scheduled events to invoke a Lambda function

The following code examples show how to create an AWS Lambda function invoked by an Amazon EventBridge scheduled event.

Java

SDK for Java 2.x

Shows how to create an Amazon EventBridge scheduled event that invokes an AWS Lambda function. Configure EventBridge to use a cron expression to schedule when the Lambda function is invoked. In this example, you create a Lambda function by using the Lambda Java runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create an app that sends a mobile text message to your employees that congratulates them at the one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Shows how to create an Amazon EventBridge scheduled event that invokes an AWS Lambda function. Configure EventBridge to use a cron expression to schedule when the Lambda function is invoked. In this example, you create a Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create an app that sends a mobile text message to your employees that congratulates them at the one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

This example is also available in the [AWS SDK for JavaScript v3 developer guide](#).

Services used in this example

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

This example shows how to register an AWS Lambda function as the target of a scheduled Amazon EventBridge event. The Lambda handler writes a friendly message and the full event data to Amazon CloudWatch Logs for later retrieval.

- Deploys a Lambda function.
- Creates an EventBridge scheduled event and makes the Lambda function the target.
- Grants permission to let EventBridge invoke the Lambda function.
- Prints the latest data from CloudWatch Logs to show the result of the scheduled invocations.
- Cleans up all resources created during the demo.

This example is best viewed on GitHub. For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- CloudWatch Logs
- EventBridge
- Lambda

For a complete list of AWS SDK developer guides and code examples, see [Using Lambda with an AWS SDK \(p. 1021\)](#). This topic also includes information about getting started and details about previous SDK versions.

Lambda quotas

Important

New AWS accounts have reduced concurrency and memory quotas. AWS raises these quotas automatically based on your usage. You can also [request a quota increase](#).

Compute and storage

Lambda sets quotas for the amount of compute and storage resources that you can use to run and store functions. Quotas for concurrent executions and storage apply per AWS Region. Elastic network interface (ENI) quotas apply per virtual private cloud (VPC), regardless of region. The following quotas can be increased from their default values. For more information, see [Requesting a quota increase](#) in the *Service Quotas User Guide*.

Resource	Default quota	Can be increased up to
Concurrent executions	1,000	Tens of thousands
Storage for uploaded functions (.zip file archives) and layers. Each function version and layer version consumes storage. For best practices on managing your code storage, see Monitoring Lambda code storage in the <i>Lambda Operator Guide</i> .	75 GB	Terabytes
Storage for functions defined as container images. These images are stored in Amazon ECR.	See Amazon ECR service quotas .	
Elastic network interfaces per virtual private cloud (VPC) (p. 222) Note This quota is shared with other services, such as Amazon Elastic File System (Amazon EFS). See Amazon VPC quotas .	250	Thousands

For details on concurrency and how Lambda scales your function concurrency in response to traffic, see [Lambda function scaling \(p. 197\)](#).

Function configuration, deployment, and execution

The following quotas apply to function configuration, deployment, and execution. They cannot be changed.

Note

The Lambda documentation, log messages, and console use the abbreviation MB (rather than MiB) to refer to 1024 KB.

Resource	Quota
Function memory allocation (p. 71)	128 MB to 10,240 MB, in 1-MB increments. Note: Lambda allocates CPU power in proportion to the amount of memory configured. You can increase or decrease the memory and CPU power allocated to your function using the Memory (MB) setting. At 1,769 MB, a function has the equivalent of one vCPU.
Function timeout	900 seconds (15 minutes)
Function environment variables (p. 76)	4 KB, for all environment variables associated with the function, in aggregate
Function resource-based policy (p. 832)	20 KB
Function layers (p. 93)	five layers
Function burst concurrency (p. 220)	500 - 3000 (varies per Region)
Invocation payload (p. 118) (request and response)	6 MB each for request and response (synchronous) 256 KB (asynchronous)
Deployment package (.zip file archive) (p. 18) size	50 MB (zipped, for direct upload) 250 MB (unzipped) This quota applies to all the files you upload, including layers and custom runtimes. 3 MB (console editor)
Container image settings (p. 888) size	16 KB
Container image (p. 881) code package size	10 GB (maximum uncompressed image size, including all layers)
Test events (console editor)	10
/tmp directory storage	Between 512 MB and 10,240 MB, in 1-MB increments
File descriptors	1,024
Execution processes/threads	1,024

Lambda API requests

The following quotas are associated with Lambda API requests.

Resource	Quota
Invocation requests per function per Region (synchronous)	Each instance of your execution environment can serve up to 10 requests per second. In other words, the total invocation limit is 10 times your concurrency limit. See Lambda function scaling (p. 197) .
Invocation requests per function per Region (asynchronous)	Each instance of your execution environment can serve an unlimited number of requests. In other words, the total invocation limit is based only on concurrency available to your function. See Lambda function scaling (p. 197) .
Invocation requests per function version or alias (requests per second)	10 x allocated provisioned concurrency (p. 210) Note This quota applies only to functions that use provisioned concurrency.
GetFunction (p. 1220) API requests	100 requests per second
GetPolicy (p. 1252) API requests	15 requests per second
Remainder of the control plane API requests (excludes invocation, GetFunction, and GetPolicy requests)	15 requests per second

Other services

Quotas for other services, such as AWS Identity and Access Management (IAM), Amazon CloudFront (Lambda@Edge), and Amazon Virtual Private Cloud (Amazon VPC), can impact your Lambda functions. For more information, see [AWS service quotas](#) in the *Amazon Web Services General Reference*, and [Using AWS Lambda with other services \(p. 556\)](#).

AWS glossary

For the latest AWS terminology, see the [AWS glossary](#) in the *AWS General Reference*.

API reference

This section contains the AWS Lambda API Reference documentation. When making the API calls, you will need to authenticate your request by providing a signature. AWS Lambda supports signature version 4. For more information, see [Signature Version 4 signing process](#) in the *Amazon Web Services General Reference*.

For an overview of the service, see [What is AWS Lambda? \(p. 1\)](#).

You can use the AWS CLI to explore the AWS Lambda API. This guide provides several tutorials that use the AWS CLI.

Topics

- [Actions \(p. 1135\)](#)
- [Data Types \(p. 1396\)](#)

Actions

The following actions are supported:

- [AddLayerVersionPermission \(p. 1137\)](#)
- [AddPermission \(p. 1141\)](#)
- [CreateAlias \(p. 1146\)](#)
- [CreateCodeSigningConfig \(p. 1150\)](#)
- [CreateEventSourceMapping \(p. 1153\)](#)
- [CreateFunction \(p. 1165\)](#)
- [CreateFunctionUrlConfig \(p. 1178\)](#)
- [DeleteAlias \(p. 1182\)](#)
- [DeleteCodeSigningConfig \(p. 1184\)](#)
- [DeleteEventSourceMapping \(p. 1186\)](#)
- [DeleteFunction \(p. 1193\)](#)
- [DeleteFunctionCodeSigningConfig \(p. 1195\)](#)
- [DeleteFunctionConcurrency \(p. 1197\)](#)
- [DeleteFunctionEventInvokeConfig \(p. 1199\)](#)
- [DeleteFunctionUrlConfig \(p. 1201\)](#)
- [DeleteLayerVersion \(p. 1203\)](#)
- [DeleteProvisionedConcurrencyConfig \(p. 1205\)](#)
- [GetAccountSettings \(p. 1207\)](#)
- [GetAlias \(p. 1209\)](#)
- [GetCodeSigningConfig \(p. 1212\)](#)
- [GetEventSourceMapping \(p. 1214\)](#)
- [GetFunction \(p. 1220\)](#)
- [GetFunctionCodeSigningConfig \(p. 1224\)](#)
- [GetFunctionConcurrency \(p. 1227\)](#)
- [GetFunctionConfiguration \(p. 1229\)](#)

- [GetFunctionEventInvokeConfig \(p. 1237\)](#)
- [GetFunctionUrlConfig \(p. 1240\)](#)
- [GetLayerVersion \(p. 1243\)](#)
- [GetLayerVersionByArn \(p. 1247\)](#)
- [GetLayerVersionPolicy \(p. 1250\)](#)
- [GetPolicy \(p. 1252\)](#)
- [GetProvisionedConcurrencyConfig \(p. 1254\)](#)
- [GetRuntimeManagementConfig \(p. 1257\)](#)
- [Invoke \(p. 1260\)](#)
- [InvokeAsync \(p. 1266\)](#)
- [InvokeWithResponseStream \(p. 1268\)](#)
- [ListAliases \(p. 1274\)](#)
- [ListCodeSigningConfigs \(p. 1277\)](#)
- [ListEventSourceMappings \(p. 1279\)](#)
- [ListFunctionEventInvokeConfigs \(p. 1283\)](#)
- [ListFunctions \(p. 1286\)](#)
- [ListFunctionsByCodeSigningConfig \(p. 1290\)](#)
- [ListFunctionUrlConfigs \(p. 1292\)](#)
- [ListLayers \(p. 1295\)](#)
- [ListLayerVersions \(p. 1298\)](#)
- [ListProvisionedConcurrencyConfigs \(p. 1301\)](#)
- [ListTags \(p. 1304\)](#)
- [ListVersionsByFunction \(p. 1306\)](#)
- [PublishLayerVersion \(p. 1310\)](#)
- [PublishVersion \(p. 1315\)](#)
- [PutFunctionCodeSigningConfig \(p. 1324\)](#)
- [PutFunctionConcurrency \(p. 1327\)](#)
- [PutFunctionEventInvokeConfig \(p. 1330\)](#)
- [PutProvisionedConcurrencyConfig \(p. 1334\)](#)
- [PutRuntimeManagementConfig \(p. 1337\)](#)
- [RemoveLayerVersionPermission \(p. 1341\)](#)
- [RemovePermission \(p. 1343\)](#)
- [TagResource \(p. 1345\)](#)
- [UntagResource \(p. 1347\)](#)
- [UpdateAlias \(p. 1349\)](#)
- [UpdateCodeSigningConfig \(p. 1353\)](#)
- [UpdateEventSourceMapping \(p. 1356\)](#)
- [UpdateFunctionCode \(p. 1367\)](#)
- [UpdateFunctionConfiguration \(p. 1377\)](#)
- [UpdateFunctionEventInvokeConfig \(p. 1389\)](#)
- [UpdateFunctionUrlConfig \(p. 1393\)](#)

AddLayerVersionPermission

Adds permissions to the resource-based policy of a version of an [AWS Lambda layer](#). Use this action to grant layer usage permission to other accounts. You can grant permission to a single account, all accounts in an organization, or all AWS accounts.

To revoke permission, call [RemoveLayerVersionPermission \(p. 1341\)](#) with the statement ID that you specified when you added it.

Request Syntax

```
POST /2018-10-31/layers/LayerName/versions/VersionNumber/policy?RevisionId=RevisionId
HTTP/1.1
Content-type: application/json

{
  "Action": "string",
  "OrganizationId": "string",
  "Principal": "string",
  "StatementId": "string"
}
```

URI Request Parameters

The request uses the following URI parameters.

[LayerName \(p. 1137\)](#)

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_+])|[a-zA-Z0-9-_+]

Required: Yes

[RevisionId \(p. 1137\)](#)

Only update the policy if the revision ID matches the ID specified. Use this option to avoid modifying a policy that has changed since you last read it.

[VersionNumber \(p. 1137\)](#)

The version number.

Required: Yes

Request Body

The request accepts the following data in JSON format.

[Action \(p. 1137\)](#)

The API action that grants access to the layer. For example, lambda:GetLayerVersion.

Type: String

Length Constraints: Maximum length of 22.

Pattern: lambda:GetLayerVersion

Required: Yes

[OrganizationId \(p. 1137\)](#)

With the principal set to *, grant permission to all accounts in the specified organization.

Type: String

Length Constraints: Maximum length of 34.

Pattern: o-[a-zA-Z0-9]{10,32}

Required: No

[Principal \(p. 1137\)](#)

An account ID, or * to grant layer usage permission to all accounts in an organization, or all AWS accounts (if organizationId is not specified). For the last case, make sure that you really do want all AWS accounts to have usage permission to this layer.

Type: String

Pattern: \d{12}|*|arn:(aws[a-zA-Z-]*):iam::\d{12}:root

Required: Yes

[StatementId \(p. 1137\)](#)

An identifier that distinguishes the policy from others on the same layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 100.

Pattern: ([a-zA-Z0-9-_]+)

Required: Yes

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
    "RevisionId": "string",
    "Statement": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

[RevisionId \(p. 1138\)](#)

A unique identifier for the current revision of the policy.

Type: String

[Statement \(p. 1138\)](#)

The permission statement.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

PolicyLengthExceededException

The permissions policy for the resource is too large. For more information, see [Lambda quotas](#).

HTTP Status Code: 400

PreconditionFailedException

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)

- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

AddPermission

Grants an AWS service, AWS account, or AWS organization permission to use a function. You can apply the policy at the function level, or specify a qualifier to restrict access to a single version or alias. If you use a qualifier, the invoker must use the full Amazon Resource Name (ARN) of that version or alias to invoke the function. Note: Lambda does not support adding policies to version \$LATEST.

To grant permission to another account, specify the account ID as the Principal. To grant permission to an organization defined in AWS Organizations, specify the organization ID as the PrincipalOrgID. For AWS services, the principal is a domain-style identifier that the service defines, such as s3.amazonaws.com or sns.amazonaws.com. For AWS services, you can also specify the ARN of the associated resource as the SourceArn. If you grant permission to a service principal without specifying the source, other accounts could potentially configure resources in their account to invoke your Lambda function.

This operation adds a statement to a resource-based permissions policy for the function. For more information about function policies, see [Using resource-based policies for Lambda](#).

Request Syntax

```
POST /2015-03-31/functions/FunctionName/policy?Qualifier=Qualifier HTTP/1.1
Content-type: application/json

{
  "Action": "string",
  "EventSourceToken": "string",
  "FunctionUrlAuthType": "string",
  "Principal": "string",
  "PrincipalOrgID": "string",
  "RevisionId": "string",
  "SourceAccount": "string",
  "SourceArn": "string",
  "StatementId": "string"
}
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1141\)](#)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** – my-function (name-only), my-function:v1 (with alias).
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$\LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1141\)](#)

Specify a version or alias to add permissions to a published version of the function.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (| [a-zA-Z0-9\$_-]+)

Request Body

The request accepts the following data in JSON format.

[Action \(p. 1141\)](#)

The action that the principal can use on the function. For example, `lambda:InvokeFunction` or `lambda:GetFunction`.

Type: String

Pattern: (`lambda:[*]` | `lambda:[a-zA-Z]+[*]`)

Required: Yes

[EventSourceToken \(p. 1141\)](#)

For Alexa Smart Home functions, a token that the invoker must supply.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Pattern: [a-zA-Z0-9._\-\-]+

Required: No

[FunctionUrlAuthType \(p. 1141\)](#)

The type of authentication that your function URL uses. Set to `AWS_IAM` if you want to restrict access to authenticated users only. Set to `NONE` if you want to bypass IAM authentication to create a public endpoint. For more information, see [Security and auth model for Lambda function URLs](#).

Type: String

Valid Values: `NONE` | `AWS_IAM`

Required: No

[Principal \(p. 1141\)](#)

The AWS service or AWS account that invokes the function. If you specify a service, use `SourceArn` or `SourceAccount` to limit who can invoke the function through that service.

Type: String

Pattern: [^\s]+

Required: Yes

[PrincipalOrgID \(p. 1141\)](#)

The identifier for your organization in AWS Organizations. Use this to grant permissions to all the AWS accounts under this organization.

Type: String

Length Constraints: Minimum length of 12. Maximum length of 34.

Pattern: ^o-[a-zA-Z0-9]{10,32}\$

Required: No

[RevisionId \(p. 1141\)](#)

Update the policy only if the revision ID matches the ID that's specified. Use this option to avoid modifying a policy that has changed since you last read it.

Type: String

Required: No

[SourceAccount \(p. 1141\)](#)

For AWS service, the ID of the AWS account that owns the resource. Use this together with `SourceArn` to ensure that the specified account owns the resource. It is possible for an Amazon S3 bucket to be deleted by its owner and recreated by another account.

Type: String

Length Constraints: Maximum length of 12.

Pattern: \d{12}

Required: No

[SourceArn \(p. 1141\)](#)

For AWS services, the ARN of the AWS resource that invokes the function. For example, an Amazon S3 bucket or Amazon SNS topic.

Note that Lambda configures the comparison using the `StringLike` operator.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-.])+:([a-z]{2}(-gov)?-[a-z]+\-\d{1})?:(\d{12})?:(.*)

Required: No

[StatementId \(p. 1141\)](#)

A statement identifier that differentiates the statement from others in the same policy.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 100.

Pattern: ([a-zA-Z0-9-_]+)

Required: Yes

Response Syntax

```
HTTP/1.1 201
Content-type: application/json
```

```
{
  "Statement": "string"
```

}

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

[Statement \(p. 1143\)](#)

The permission statement that's added to the function policy.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

PolicyLengthExceededException

The permissions policy for the resource is too large. For more information, see [Lambda quotas](#).

HTTP Status Code: 400

PreconditionFailedException

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

CreateAlias

Creates an [alias](#) for a Lambda function version. Use aliases to provide clients with a function identifier that you can update to invoke a different version.

You can also map an alias to split invocation requests between two versions. Use the `RoutingConfig` parameter to specify a second version and the percentage of invocation requests that it receives.

Request Syntax

```
POST /2015-03-31/functions/FunctionName/aliases HTTP/1.1
Content-type: application/json

{
  "Description": "string",
  "FunctionVersion": "string",
  "Name": "string",
  "RoutingConfig": {
    "AdditionalVersionWeights": {
      "string": number
    }
  }
}
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName (p. 1146)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

Request Body

The request accepts the following data in JSON format.

Description (p. 1146)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

[FunctionVersion \(p. 1146\)](#)

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST | [0-9]+)

Required: Yes

[Name \(p. 1146\)](#)

The name of the alias.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (?![0-9]+\$)([a-zA-Z0-9-_]+)

Required: Yes

[RoutingConfig \(p. 1146\)](#)

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 1403\)](#) object

Required: No

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
    "AliasArn": "string",
    "Description": "string",
    "FunctionVersion": "string",
    "Name": "string",
    "RevisionId": "string",
    "RoutingConfig": {
        "AdditionalVersionWeights": {
            "string" : number
        }
    }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

[AliasArn \(p. 1147\)](#)

The Amazon Resource Name (ARN) of the alias.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[Description \(p. 1147\)](#)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[FunctionVersion \(p. 1147\)](#)

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

[Name \(p. 1147\)](#)

The name of the alias.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(?!^[\d-]+)([a-zA-Z0-9-_]+)`

[RevisionId \(p. 1147\)](#)

A unique identifier that changes when you update the alias.

Type: String

[RoutingConfig \(p. 1147\)](#)

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 1403\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

CreateCodeSigningConfig

Creates a code signing configuration. A [code signing configuration](#) defines a list of allowed signing profiles and defines the code-signing validation policy (action to be taken if deployment validation checks fail).

Request Syntax

```
POST /2020-04-22/code-signing-configs/ HTTP/1.1
Content-type: application/json

{
  "AllowedPublishers": {
    "SigningProfileVersionArns": [ "string" ]
  },
  "CodeSigningPolicies": {
    "UntrustedArtifactOnDeployment": "string"
  },
  "Description": "string"
}
```

URI Request Parameters

The request does not use any URI parameters.

Request Body

The request accepts the following data in JSON format.

[AllowedPublishers \(p. 1150\)](#)

Signing profiles for this code signing configuration.

Type: [AllowedPublishers \(p. 1404\)](#) object

Required: Yes

[CodeSigningPolicies \(p. 1150\)](#)

The code signing policies define the actions to take if the validation checks fail.

Type: [CodeSigningPolicies \(p. 1408\)](#) object

Required: No

[Description \(p. 1150\)](#)

Descriptive name for this code signing configuration.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

Response Syntax

```
HTTP/1.1 201
```

```
Content-type: application/json

{
    "CodeSigningConfig": {
        "AllowedPublishers": {
            "SigningProfileVersionArns": [ "string" ]
        },
        "CodeSigningConfigArn": "string",
        "CodeSigningConfigId": "string",
        "CodeSigningPolicies": {
            "UntrustedArtifactOnDeployment": "string"
        },
        "Description": "string",
        "LastModified": "string"
    }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

[CodeSigningConfig \(p. 1150\)](#)

The code signing configuration.

Type: [CodeSigningConfig \(p. 1406\)](#) object

Errors

InvalidArgumentException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

CreateEventSourceMapping

Creates a mapping between an event source and an AWS Lambda function. Lambda reads items from the event source and invokes the function.

For details about how to configure different event sources, see the following topics.

- [Amazon DynamoDB Streams](#)
- [Amazon Kinesis](#)
- [Amazon SQS](#)
- [Amazon MQ and RabbitMQ](#)
- [Amazon MSK](#)
- [Apache Kafka](#)
- [Amazon DocumentDB](#)

The following error handling options are available only for stream sources (DynamoDB and Kinesis):

- `BisectBatchOnFunctionError` – If the function returns an error, split the batch in two and retry.
- `DestinationConfig` – Send discarded records to an Amazon SQS queue or Amazon SNS topic.
- `MaximumRecordAgeInSeconds` – Discard records older than the specified age. The default value is infinite (-1). When set to infinite (-1), failed records are retried until the record expires
- `MaximumRetryAttempts` – Discard records after the specified number of retries. The default value is infinite (-1). When set to infinite (-1), failed records are retried until the record expires.
- `ParallelizationFactor` – Process multiple batches from each shard concurrently.

For information about which configuration parameters apply to each event source, see the following topics.

- [Amazon DynamoDB Streams](#)
- [Amazon Kinesis](#)
- [Amazon SQS](#)
- [Amazon MQ and RabbitMQ](#)
- [Amazon MSK](#)
- [Apache Kafka](#)
- [Amazon DocumentDB](#)

Request Syntax

```
POST /2015-03-31/event-source-mappings/ HTTP/1.1
Content-type: application/json
```

```
{
  "AmazonManagedKafkaEventSourceConfig": {
    "ConsumerGroupIdBatchSize": number,
  "BisectBatchOnFunctionError": boolean,
  "DestinationConfig": {
    "OnFailure": {
      "Destination": "string"
    },
    "OnSuccess": {
```

```

        "Destination": "string"
    },
    "DocumentDBEventSourceConfig": {
        "CollectionName": "string",
        "DatabaseName": "string",
        "FullDocument": "string"
    },
    "Enabled": boolean,
    "EventSourceArn": "string",
    "FilterCriteria": {
        "Filters": [
            {
                "Pattern": "string"
            }
        ]
    },
    "FunctionName": "string",
    "FunctionResponseTypes": [ "string" ],
    "MaximumBatchingWindowInSeconds": number,
    "MaximumRecordAgeInSeconds": number,
    "MaximumRetryAttempts": number,
    "ParallelizationFactor": number,
    "Queues": [ "string" ],
    "ScalingConfig": {
        "MaximumConcurrency": number
    },
    "SelfManagedEventSource": {
        "Endpoints": {
            "string" : [ "string" ]
        }
    },
    "SelfManagedKafkaEventSourceConfig": {
        "ConsumerGroupId": "string"
    },
    "SourceAccessConfigurations": [
        {
            "Type": "string",
            "URI": "string"
        }
    ],
    "StartingPosition": "string",
    "StartingPositionTimestamp": number,
    "Topics": [ "string" ],
    "TumblingWindowInSeconds": number
}

```

URI Request Parameters

The request does not use any URI parameters.

Request Body

The request accepts the following data in JSON format.

[AmazonManagedKafkaEventSourceConfig \(p. 1153\)](#)

Specific configuration settings for an Amazon Managed Streaming for Apache Kafka (Amazon MSK) event source.

Type: [AmazonManagedKafkaEventSourceConfig \(p. 1405\)](#) object

Required: No

[BatchSize \(p. 1153\)](#)

The maximum number of records in each batch that Lambda pulls from your stream or queue and sends to your function. Lambda passes all of the records in the batch to the function in a single call, up to the payload limit for synchronous invocation (6 MB).

- **Amazon Kinesis** – Default 100. Max 10,000.
- **Amazon DynamoDB Streams** – Default 100. Max 10,000.
- **Amazon Simple Queue Service** – Default 10. For standard queues the max is 10,000. For FIFO queues the max is 10.
- **Amazon Managed Streaming for Apache Kafka** – Default 100. Max 10,000.
- **Self-managed Apache Kafka** – Default 100. Max 10,000.
- **Amazon MQ (ActiveMQ and RabbitMQ)** – Default 100. Max 10,000.
- **DocumentDB** – Default 100. Max 10,000.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

Required: No

[BisectBatchOnFunctionError \(p. 1153\)](#)

(Kinesis and DynamoDB Streams only) If the function returns an error, split the batch in two and retry.

Type: Boolean

Required: No

[DestinationConfig \(p. 1153\)](#)

(Kinesis and DynamoDB Streams only) A standard Amazon SQS queue or standard Amazon SNS topic destination for discarded records.

Type: [DestinationConfig \(p. 1413\)](#) object

Required: No

[DocumentDBEventSourceConfig \(p. 1153\)](#)

Specific configuration settings for a DocumentDB event source.

Type: [DocumentDBEventSourceConfig \(p. 1414\)](#) object

Required: No

[Enabled \(p. 1153\)](#)

When true, the event source mapping is active. When false, Lambda pauses polling and invocation.

Default: True

Type: Boolean

Required: No

[EventSourceArn \(p. 1153\)](#)

The Amazon Resource Name (ARN) of the event source.

- **Amazon Kinesis** – The ARN of the data stream or a stream consumer.
- **Amazon DynamoDB Streams** – The ARN of the stream.
- **Amazon Simple Queue Service** – The ARN of the queue.

- **Amazon Managed Streaming for Apache Kafka** – The ARN of the cluster.
- **Amazon MQ** – The ARN of the broker.
- **Amazon DocumentDB** – The ARN of the DocumentDB change stream.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-\-])+([a-z]{2}(-gov)?-[a-z]+\-\d{1}):(\d{12}):(.*)`

Required: No

[FilterCriteria \(p. 1153\)](#)

An object that defines the filter criteria that determine whether Lambda should process an event. For more information, see [Lambda event filtering](#).

Type: [FilterCriteria \(p. 1426\)](#) object

Required: No

[FunctionName \(p. 1153\)](#)

The name of the Lambda function.

Name formats

- **Function name** – MyFunction.
- **Function ARN** – `arn:aws:lambda:us-west-2:123456789012:function:MyFunction`.
- **Version or Alias ARN** – `arn:aws:lambda:us-west-2:123456789012:function:MyFunction:PROD`.
- **Partial ARN** – `123456789012:function:MyFunction`.

The length constraint applies only to the full ARN. If you specify only the function name, it's limited to 64 characters in length.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:(aws[a-zA-Z-]*):lambda:)([a-z]{2}(-gov)?-[a-z]+\-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: Yes

[FunctionResponseTypes \(p. 1153\)](#)

(Kinesis, DynamoDB Streams, and Amazon SQS) A list of current response type enums applied to the event source mapping.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 1 item.

Valid Values: ReportBatchItemFailures

Required: No

[MaximumBatchingWindowInSeconds \(p. 1153\)](#)

The maximum amount of time, in seconds, that Lambda spends gathering records before invoking the function. You can configure MaximumBatchingWindowInSeconds to any value from 0 seconds to 300 seconds in increments of seconds.

For streams and Amazon SQS event sources, the default batching window is 0 seconds. For Amazon MSK, Self-managed Apache Kafka, Amazon MQ, and DocumentDB event sources, the default batching window is 500 ms. Note that because you can only change MaximumBatchingWindowInSeconds in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

Related setting: For streams and Amazon SQS event sources, when you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

Required: No

MaximumRecordAgeInSeconds (p. 1153)

(Kinesis and DynamoDB Streams only) Discard records older than the specified age. The default value is infinite (-1).

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 604800.

Required: No

MaximumRetryAttempts (p. 1153)

(Kinesis and DynamoDB Streams only) Discard records after the specified number of retries. The default value is infinite (-1). When set to infinite (-1), failed records are retried until the record expires.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

Required: No

ParallelizationFactor (p. 1153)

(Kinesis and DynamoDB Streams only) The number of batches to process from each shard concurrently.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10.

Required: No

Queues (p. 1153)

(MQ) The name of the Amazon MQ broker destination queue to consume.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 1000.

Pattern: `[\s\S]*`

Required: No

[ScalingConfig \(p. 1153\)](#)

(Amazon SQS only) The scaling configuration for the event source. For more information, see [Configuring maximum concurrency for Amazon SQS event sources](#).

Type: [ScalingConfig \(p. 1458\)](#) object

Required: No

[SelfManagedEventSource \(p. 1153\)](#)

The self-managed Apache Kafka cluster to receive records from.

Type: [SelfManagedEventSource \(p. 1459\)](#) object

Required: No

[SelfManagedKafkaEventSourceConfig \(p. 1153\)](#)

Specific configuration settings for a self-managed Apache Kafka event source.

Type: [SelfManagedKafkaEventSourceConfig \(p. 1460\)](#) object

Required: No

[SourceAccessConfigurations \(p. 1153\)](#)

An array of authentication protocols or VPC components required to secure your event source.

Type: Array of [SourceAccessConfiguration \(p. 1463\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

Required: No

[StartingPosition \(p. 1153\)](#)

The position in a stream from which to start reading. Required for Amazon Kinesis, Amazon DynamoDB, and Amazon MSK Streams sources. AT_TIMESTAMP is supported only for Amazon Kinesis streams and Amazon DocumentDB.

Type: String

Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

Required: No

[StartingPositionTimestamp \(p. 1153\)](#)

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds.

Type: Timestamp

Required: No

[Topics \(p. 1153\)](#)

The name of the Kafka topic.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 249.

Pattern: ^[^.]([a-zA-Z0-9\-.]+)+

Required: No

[TumblingWindowInSeconds \(p. 1153\)](#)

(Kinesis and DynamoDB Streams only) The duration in seconds of a processing window for DynamoDB and Kinesis Streams event sources. A value of 0 seconds indicates no tumbling window.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

Required: No

Response Syntax

```
HTTP/1.1 202
Content-type: application/json

{
    "AmazonManagedKafkaEventSourceConfig": {
        "ConsumerGroupId": "string"
    },
    "BatchSize": number,
    "BisectBatchOnFunctionError": boolean,
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "DocumentDBEventSourceConfig": {
        "CollectionName": "string",
        "DatabaseName": "string",
        "FullDocument": "string"
    },
    "EventSourceArn": "string",
    "FilterCriteria": {
        "Filters": [
            {
                "Pattern": "string"
            }
        ]
    },
    "FunctionArn": "string",
    "FunctionResponseTypes": [ "string" ],
    "LastModified": number,
    "LastProcessingResult": "string",
    "MaximumBatchingWindowInSeconds": number,
    "MaximumRecordAgeInSeconds": number,
    "MaximumRetryAttempts": number,
    "ParallelizationFactor": number,
    "Queues": [ "string" ],
    "ScalingConfig": {
        "MaximumConcurrency": number
    },
    "SelfManagedEventSource": {
        "Endpoints": {
            "string" : [ "string" ]
        }
    }
}
```

```
        },
        "SelfManagedKafkaEventSourceConfig": {
            "ConsumerGroupId": "string"
        },
        "SourceAccessConfigurations": [
            {
                "Type": "string",
                "URI": "string"
            }
        ],
        "StartingPosition": "string",
        "StartingPositionTimestamp": number,
        "State": "string",
        "StateTransitionReason": "string",
        "Topics": [ "string" ],
        "TumblingWindowInSeconds": number,
        "UUID": "string"
    }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 202 response.

The following data is returned in JSON format by the service.

[AmazonManagedKafkaEventSourceConfig \(p. 1159\)](#)

Specific configuration settings for an Amazon Managed Streaming for Apache Kafka (Amazon MSK) event source.

Type: [AmazonManagedKafkaEventSourceConfig \(p. 1405\)](#) object

[BatchSize \(p. 1159\)](#)

The maximum number of records in each batch that Lambda pulls from your stream or queue and sends to your function. Lambda passes all of the records in the batch to the function in a single call, up to the payload limit for synchronous invocation (6 MB).

Default value: Varies by service. For Amazon SQS, the default is 10. For all other services, the default is 100.

Related setting: When you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

[BisectBatchOnFunctionError \(p. 1159\)](#)

(Kinesis and DynamoDB Streams only) If the function returns an error, split the batch in two and retry. The default value is false.

Type: Boolean

[DestinationConfig \(p. 1159\)](#)

(Kinesis and DynamoDB Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: [DestinationConfig \(p. 1413\)](#) object

[DocumentDBEventSourceConfig \(p. 1159\)](#)

Specific configuration settings for a DocumentDB event source.

Type: [DocumentDBEventSourceConfig \(p. 1414\)](#) object

[**EventSourceArn \(p. 1159\)**](#)

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-\-])+([a-z]{2}(-gov)?-[a-z]+\-\d{1})?:(\d{12})?:(.*?)`

[**FilterCriteria \(p. 1159\)**](#)

An object that defines the filter criteria that determine whether Lambda should process an event. For more information, see [Lambda event filtering](#).

Type: [FilterCriteria \(p. 1426\)](#) object

[**FunctionArn \(p. 1159\)**](#)

The ARN of the Lambda function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+\-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$\$LATEST|[a-zA-Z0-9-_]+))?`

[**FunctionResponseTypes \(p. 1159\)**](#)

(Kinesis, DynamoDB Streams, and Amazon SQS) A list of current response type enums applied to the event source mapping.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 1 item.

Valid Values: ReportBatchItemFailures

[**LastModified \(p. 1159\)**](#)

The date that the event source mapping was last updated or that its state changed, in Unix time seconds.

Type: Timestamp

[**LastProcessingResult \(p. 1159\)**](#)

The result of the last Lambda invocation of your function.

Type: String

[**MaximumBatchingWindowInSeconds \(p. 1159\)**](#)

The maximum amount of time, in seconds, that Lambda spends gathering records before invoking the function. You can configure MaximumBatchingWindowInSeconds to any value from 0 seconds to 300 seconds in increments of seconds.

For streams and Amazon SQS event sources, the default batching window is 0 seconds.

For Amazon MSK, Self-managed Apache Kafka, Amazon MQ, and DocumentDB event sources, the default batching window is 500 ms. Note that because you can only change

MaximumBatchingWindowInSeconds in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

Related setting: For streams and Amazon SQS event sources, when you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

[MaximumRecordAgeInSeconds \(p. 1159\)](#)

(Kinesis and DynamoDB Streams only) Discard records older than the specified age. The default value is -1, which sets the maximum age to infinite. When the value is set to infinite, Lambda never discards old records.

Note

The minimum valid value for maximum record age is 60s. Although values less than 60 and greater than -1 fall within the parameter's absolute range, they are not allowed

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 604800.

[MaximumRetryAttempts \(p. 1159\)](#)

(Kinesis and DynamoDB Streams only) Discard records after the specified number of retries. The default value is -1, which sets the maximum number of retries to infinite. When MaximumRetryAttempts is infinite, Lambda retries failed records until the record expires in the event source.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

[ParallelizationFactor \(p. 1159\)](#)

(Kinesis and DynamoDB Streams only) The number of batches to process concurrently from each shard. The default value is 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10.

[Queues \(p. 1159\)](#)

(Amazon MQ) The name of the Amazon MQ broker destination queue to consume.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 1000.

Pattern: [\s\S]*

[ScalingConfig \(p. 1159\)](#)

(Amazon SQS only) The scaling configuration for the event source. For more information, see [Configuring maximum concurrency for Amazon SQS event sources](#).

Type: [ScalingConfig \(p. 1458\)](#) object

[SelfManagedEventSource \(p. 1159\)](#)

The self-managed Apache Kafka cluster for your event source.

Type: [SelfManagedEventSource \(p. 1459\)](#) object

[SelfManagedKafkaEventSourceConfig \(p. 1159\)](#)

Specific configuration settings for a self-managed Apache Kafka event source.

Type: [SelfManagedKafkaEventSourceConfig \(p. 1460\)](#) object

[**SourceAccessConfigurations \(p. 1159\)**](#)

An array of the authentication protocol, VPC components, or virtual host to secure and define your event source.

Type: Array of [SourceAccessConfiguration \(p. 1463\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

[**StartingPosition \(p. 1159\)**](#)

The position in a stream from which to start reading. Required for Amazon Kinesis, Amazon DynamoDB, and Amazon MSK stream sources. AT_TIMESTAMP is supported only for Amazon Kinesis streams and Amazon DocumentDB.

Type: String

Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

[**StartingPositionTimestamp \(p. 1159\)**](#)

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds.

Type: Timestamp

[**State \(p. 1159\)**](#)

The state of the event source mapping. It can be one of the following: Creating, Enabling, Enabled, Disabling, Disabled, Updating, or Deleting.

Type: String

[**StateTransitionReason \(p. 1159\)**](#)

Indicates whether a user or Lambda made the last change to the event source mapping.

Type: String

[**Topics \(p. 1159\)**](#)

The name of the Kafka topic.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 249.

Pattern: ^[^.]([a-zA-Z0-9\-_\.]+)+

[**TumblingWindowInSeconds \(p. 1159\)**](#)

(Kinesis and DynamoDB Streams only) The duration in seconds of a processing window for DynamoDB and Kinesis Streams event sources. A value of 0 seconds indicates no tumbling window.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

[**UUID \(p. 1159\)**](#)

The identifier of the event source mapping.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

CreateFunction

Creates a Lambda function. To create a function, you need a [deployment package](#) and an [execution role](#). The deployment package is a .zip file archive or container image that contains your function code. The execution role grants the function permission to use AWS services, such as Amazon CloudWatch Logs for log streaming and AWS X-Ray for request tracing.

If the deployment package is a [container image](#), then you set the package type to `Image`. For a container image, the `Code` property must include the URI of a container image in the Amazon ECR registry. You do not need to specify the handler and runtime properties.

If the deployment package is a [zip file archive](#), then you set the package type to `Zip`. For a .zip file archive, the `Code` property specifies the location of the .zip file. You must also specify the handler and runtime properties. The code in the deployment package must be compatible with the target instruction set architecture of the function (x86-64 or arm64). If you do not specify the architecture, then the default value is x86-64.

When you create a function, Lambda provisions an instance of the function and its supporting resources. If your function connects to a VPC, this process can take a minute or so. During this time, you can't invoke or modify the function. The `State`, `StateReason`, and `StateReasonCode` fields in the response from [GetFunctionConfiguration \(p. 1229\)](#) indicate when the function is ready to invoke. For more information, see [Lambda function states](#).

A function has an unpublished version, and can have published versions and aliases. The unpublished version changes when you update your function's code and configuration. A published version is a snapshot of your function code and configuration that can't be changed. An alias is a named resource that maps to a version, and can be changed to map to a different version. Use the `Publish` parameter to create version 1 of your function from its initial configuration.

The other parameters let you configure version-specific and function-level settings. You can modify version-specific settings later with [UpdateFunctionConfiguration \(p. 1377\)](#). Function-level settings apply to both the unpublished and published versions of the function, and include tags ([TagResource \(p. 1345\)](#)) and per-function concurrency limits ([PutFunctionConcurrency \(p. 1327\)](#)).

You can use code signing if your deployment package is a .zip file archive. To enable code signing for this function, specify the ARN of a code-signing configuration. When a user attempts to deploy a code package with [UpdateFunctionCode \(p. 1367\)](#), Lambda checks that the code package has a valid signature from a trusted publisher. The code-signing configuration includes set of signing profiles, which define the trusted publishers for this function.

If another AWS account or an AWS service invokes your function, use [AddPermission \(p. 1141\)](#) to grant permission by creating a resource-based AWS Identity and Access Management (IAM) policy. You can grant permissions at the function level, on a version, or on an alias.

To invoke your function directly, use [Invoke \(p. 1260\)](#). To invoke your function in response to events in other AWS services, create an event source mapping ([CreateEventSourceMapping \(p. 1153\)](#)), or configure a function trigger in the other service. For more information, see [Invoking Lambda functions](#).

Request Syntax

```
POST /2015-03-31/functions HTTP/1.1
Content-type: application/json

{
    "Architectures": [ "string" ],
    "Code": {
        "ImageUri": "string",
        "S3Bucket": "string",
        "S3Key": "string",
        "S3ObjectVersion": "string",
    }
}
```

```
        "ZipFile": blob
    },
    "CodeSigningConfigArn": "string",
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
    "Description": "string",
    "Environment": {
        "Variables": {
            "string" : "string"
        }
    },
    "EphemeralStorage": {
        "Size": number
    },
    "FileSystemConfigs": [
        {
            "Arn": "string",
            "LocalMountPath": "string"
        }
    ],
    "FunctionName": "string",
    "Handler": "string",
    "ImageConfig": {
        "Command": [ "string" ],
        "EntryPoint": [ "string" ],
        "WorkingDirectory": "string"
    },
    "KMSKeyArn": "string",
    "Layers": [ "string" ],
    "MemorySize": number,
    "PackageType": "string",
    "Publish": boolean,
    "Role": "string",
    "Runtime": "string",
    "SnapStart": {
        "ApplyOn": "string"
    },
    "Tags": {
        "string" : "string"
    },
    "Timeout": number,
    "TracingConfig": {
        "Mode": "string"
    },
    "VpcConfig": {
        "SecurityGroupIds": [ "string" ],
        "SubnetIds": [ "string" ]
    }
}
```

URI Request Parameters

The request does not use any URI parameters.

Request Body

The request accepts the following data in JSON format.

[Architectures \(p. 1165\)](#)

The instruction set architecture that the function supports. Enter a string array with one of the valid values (arm64 or x86_64). The default value is x86_64.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: x86_64 | arm64

Required: No

[Code \(p. 1165\)](#)

The code for the function.

Type: [FunctionCode \(p. 1427\)](#) object

Required: Yes

[CodeSigningConfigArn \(p. 1165\)](#)

To enable code signing for this function, specify the ARN of a code-signing configuration. A code-signing configuration includes a set of signing profiles, which define the trusted publishers for this function.

Type: String

Length Constraints: Maximum length of 200.

Pattern: arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}((-gov)|(-iso(b?)))?-+[a-z]+-\d{1}:\d{12}:code-signing-config:csc-[a-z0-9]{17}

Required: No

[DeadLetterConfig \(p. 1165\)](#)

A dead-letter queue configuration that specifies the queue or topic where Lambda sends asynchronous events when they fail processing. For more information, see [Dead-letter queues](#).

Type: [DeadLetterConfig \(p. 1412\)](#) object

Required: No

[Description \(p. 1165\)](#)

A description of the function.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

[Environment \(p. 1165\)](#)

Environment variables that are accessible from function code during execution.

Type: [Environment \(p. 1415\)](#) object

Required: No

[EphemeralStorage \(p. 1165\)](#)

The size of the function's /tmp directory in MB. The default value is 512, but can be any whole number between 512 and 10,240 MB.

Type: [EphemeralStorage \(p. 1418\)](#) object

Required: No

[FileSystemConfigs \(p. 1165\)](#)

Connection settings for an Amazon EFS file system.

Type: Array of [FileSystemConfig \(p. 1424\)](#) objects

Array Members: Maximum number of 1 item.

Required: No

[FunctionName \(p. 1165\)](#)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Handler \(p. 1165\)](#)

The name of the method within your code that Lambda calls to run your function. Handler is required if the deployment package is a .zip file archive. The format includes the file name. It can also include namespaces and other qualifiers, depending on the runtime. For more information, see [Lambda programming model](#).

Type: String

Length Constraints: Maximum length of 128.

Pattern: [^\s]+

Required: No

[ImageConfig \(p. 1165\)](#)

Container image [configuration values](#) that override the values in the container image Dockerfile.

Type: [ImageConfig \(p. 1440\)](#) object

Required: No

[KMSKeyArn \(p. 1165\)](#)

The ARN of the AWS Key Management Service (AWS KMS) customer managed key that's used to encrypt your function's [environment variables](#). When [Lambda SnapStart](#) is activated, this key is also used to encrypt your function's snapshot. If you don't provide a customer managed key, Lambda uses a default service key.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-.]+:[.]*|()|)`

Required: No

[Layers \(p. 1165\)](#)

A list of [function layers](#) to add to the function's execution environment. Specify each layer by its ARN, including the version.

Type: Array of strings

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+`

Required: No

[MemorySize \(p. 1165\)](#)

The amount of [memory available to the function](#) at runtime. Increasing the function memory also increases its CPU allocation. The default value is 128 MB. The value can be any multiple of 1 MB.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 10240.

Required: No

[PackageType \(p. 1165\)](#)

The type of deployment package. Set to `Image` for container image and set to `Zip` for .zip file archive.

Type: String

Valid Values: Zip | Image

Required: No

[Publish \(p. 1165\)](#)

Set to true to publish the first version of the function during creation.

Type: Boolean

Required: No

[Role \(p. 1165\)](#)

The Amazon Resource Name (ARN) of the function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\\-/_]+`

Required: Yes

[Runtime \(p. 1165\)](#)

The identifier of the function's [runtime](#). Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

Required: No

[SnapStart \(p. 1165\)](#)

The function's [SnapStart](#) setting.

Type: [SnapStart \(p. 1461\)](#) object

Required: No

[Tags \(p. 1165\)](#)

A list of [tags](#) to apply to the function.

Type: String to string map

Required: No

[Timeout \(p. 1165\)](#)

The amount of time (in seconds) that Lambda allows a function to run before stopping it. The default is 3 seconds. The maximum allowed value is 900 seconds. For more information, see [Lambda execution environment](#).

Type: Integer

Valid Range: Minimum value of 1.

Required: No

[TracingConfig \(p. 1165\)](#)

Set Mode to Active to sample and trace a subset of incoming requests with [X-Ray](#).

Type: [TracingConfig \(p. 1465\)](#) object

Required: No

[VpcConfig \(p. 1165\)](#)

For network connectivity to AWS resources in a VPC, specify a list of security groups and subnets in the VPC. When you connect a function to a VPC, it can access resources and the internet only through that VPC. For more information, see [Configuring a Lambda function to access resources in a VPC](#).

Type: [VpcConfig \(p. 1467\)](#) object

Required: No

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
    "Architectures": [ "string" ],
```

```

"CodeSha256": "string",
"CodeSize": number,
"DeadLetterConfig": {
    "TargetArn": "string"
},
"Description": "string",
"Environment": {
    "Error": {
        "ErrorCode": "string",
        "Message": "string"
    },
    "Variables": {
        "string" : "string"
    }
},
"EphemeralStorage": {
    "Size": number
},
"FileSystemConfigs": [
    {
        "Arn": "string",
        "LocalMountPath": "string"
    }
],
"FunctionArn": "string",
"FunctionName": "string",
"Handler": "string",
"ImageConfigResponse": {
    "Error": {
        "ErrorCode": "string",
        "Message": "string"
    },
    "ImageConfig": {
        "Command": [ "string" ],
        "EntryPoint": [ "string" ],
        "WorkingDirectory": "string"
    }
},
"KMSKeyArn": "string",
"LastModified": "string",
"LastUpdateStatus": "string",
"LastUpdateStatusReason": "string",
"LastUpdateStatusReasonCode": "string",
"Layers": [
    {
        "Arn": "string",
        "CodeSize": number,
        "SigningJobArn": "string",
        "SigningProfileVersionArn": "string"
    }
],
"MasterArn": "string",
"MemorySize": number,
"PackageType": "string",
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"RuntimeVersionConfig": {
    "Error": {
        "ErrorCode": "string",
        "Message": "string"
    },
    "RuntimeVersionArn": "string"
},
"SigningJobArn": "string",
"SigningProfileVersionArn": "string",

```

```
"SnapStart": {  
    "ApplyOn": "string",  
    "OptimizationStatus": "string"  
},  
"State": "string",  
"StateReason": "string",  
"StateReasonCode": "string",  
"Timeout": number,  
"TracingConfig": {  
    "Mode": "string"  
},  
"Version": "string",  
"VpcConfig": {  
    "SecurityGroupIds": [ "string" ],  
    "SubnetIds": [ "string" ],  
    "VpcId": "string"  
}  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

[Architectures \(p. 1170\)](#)

The instruction set architecture that the function supports. Architecture is a string array with one of the valid values. The default architecture value is x86_64.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: x86_64 | arm64

[CodeSha256 \(p. 1170\)](#)

The SHA256 hash of the function's deployment package.

Type: String

[CodeSize \(p. 1170\)](#)

The size of the function's deployment package, in bytes.

Type: Long

[DeadLetterConfig \(p. 1170\)](#)

The function's dead letter queue.

Type: [DeadLetterConfig \(p. 1412\)](#) object

[Description \(p. 1170\)](#)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[Environment \(p. 1170\)](#)

The function's [environment variables](#). Omitted from AWS CloudTrail logs.

Type: [EnvironmentResponse \(p. 1417\)](#) object

[EphemeralStorage \(p. 1170\)](#)

The size of the function's /tmp directory in MB. The default value is 512, but it can be any whole number between 512 and 10,240 MB.

Type: [EphemeralStorage \(p. 1418\)](#) object

[FileSystemConfigs \(p. 1170\)](#)

Connection settings for an [Amazon EFS file system](#).

Type: Array of [FileSystemConfig \(p. 1424\)](#) objects

Array Members: Maximum number of 1 item.

[FunctionArn \(p. 1170\)](#)

The function's Amazon Resource Name (ARN).

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:(\\$LATEST|[a-zA-Z0-9-_]+))?

[FunctionName \(p. 1170\)](#)

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?\d{12}:?(function:)?([a-zA-Z0-9-_\.]+)(:\\$LATEST|[a-zA-Z0-9-_]+)?

[Handler \(p. 1170\)](#)

The function that Lambda calls to begin running your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: [^\s]+

[ImageConfigResponse \(p. 1170\)](#)

The function's image configuration values.

Type: [ImageConfigResponse \(p. 1442\)](#) object

[KMSKeyArn \(p. 1170\)](#)

The AWS KMS key that's used to encrypt the function's [environment variables](#). When [Lambda SnapStart](#) is activated, this key is also used to encrypt the function's snapshot. This key is returned only if you've configured a customer managed key.

Type: String

Pattern: (arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-_\.]+\.:*)|()

[LastModified \(p. 1170\)](#)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[LastUpdateStatus \(p. 1170\)](#)

The status of the last update that was performed on the function. This is first set to Successful after function creation completes.

Type: String

Valid Values: Successful | Failed | InProgress

[LastUpdateStatusReason \(p. 1170\)](#)

The reason for the last update that was performed on the function.

Type: String

[LastUpdateStatusReasonCode \(p. 1170\)](#)

The reason code for the last update that was performed on the function.

Type: String

Valid Values: EniLimitExceeded | InsufficientRolePermissions | InvalidConfiguration | InternalError | SubnetOutOfIPAddresses | InvalidSubnet | InvalidSecurityGroup | ImageDeleted | ImageAccessDenied | InvalidImage | KMSKeyAccessDenied | KMSKeyNotFound | InvalidStateKMSKey | DisabledKMSKey | EFSIOError | EFSMountConnectivityError | EFSMountFailure | EFSMountTimeout | InvalidRuntime | InvalidZipFileException | FunctionError

[Layers \(p. 1170\)](#)

The function's [layers](#).

Type: Array of [Layer \(p. 1446\)](#) objects

[MasterArn \(p. 1170\)](#)

For Lambda@Edge functions, the ARN of the main function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?

[MemorySize \(p. 1170\)](#)

The amount of memory available to the function at runtime.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 10240.

[PackageType \(p. 1170\)](#)

The type of deployment package. Set to Image for container image and set Zip for .zip file archive.

Type: String

Valid Values: Zip | Image

[RevisionId \(p. 1170\)](#)

The latest updated revision of the function or alias.

Type: String

[Role \(p. 1170\)](#)

The function's execution role.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@-_/]+

[Runtime \(p. 1170\)](#)

The identifier of the function's [runtime](#). Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

[RuntimeVersionConfig \(p. 1170\)](#)

The ARN of the runtime and any errors that occurred.

Type: [RuntimeVersionConfig \(p. 1456\)](#) object

[SigningJobArn \(p. 1170\)](#)

The ARN of the signing job.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-\-]+)([a-z]{2}(-gov)?-[a-z]+\-\d{1}):(\d{12}):(.*)

[SigningProfileVersionArn \(p. 1170\)](#)

The ARN of the signing profile version.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-\-]+)([a-z]{2}(-gov)?-[a-z]+\-\d{1}):(\d{12}):(.*)

[SnapStart \(p. 1170\)](#)

Set `ApplyOn` to `PublishedVersions` to create a snapshot of the initialized execution environment when you publish a function version. For more information, see [Improving startup performance with Lambda SnapStart](#).

Type: [SnapStartResponse \(p. 1462\)](#) object

[State \(p. 1170\)](#)

The current state of the function. When the state is `Inactive`, you can reactivate the function by invoking it.

Type: String

Valid Values: Pending | Active | Inactive | Failed

[StateReason \(p. 1170\)](#)

The reason for the function's current state.

Type: String

[StateReasonCode \(p. 1170\)](#)

The reason code for the function's current state. When the code is `Creating`, you can't invoke or modify the function.

Type: String

Valid Values: `Idle` | `Creating` | `Restoring` | `EniLimitExceeded` | `InsufficientRolePermissions` | `InvalidConfiguration` | `InternalError` | `SubnetOutOfIPAddresses` | `InvalidSubnet` | `InvalidSecurityGroup` | `ImageDeleted` | `ImageAccessDenied` | `InvalidImage` | `KMSKeyAccessDenied` | `KMSKeyNotFound` | `InvalidStateKMSKey` | `DisabledKMSKey` | `EFSIOError` | `EFSMountConnectivityError` | `EFSMountFailure` | `EFSMountTimeout` | `InvalidRuntime` | `InvalidZipFileException` | `FunctionError`

[Timeout \(p. 1170\)](#)

The amount of time in seconds that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

[TracingConfig \(p. 1170\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 1466\)](#) object

[Version \(p. 1170\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST | [0-9]+)`

[VpcConfig \(p. 1170\)](#)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 1468\)](#) object

Errors

CodeSigningConfigNotFoundException

The specified code signing configuration does not exist.

HTTP Status Code: 404

CodeStorageExceededException

Your AWS account has exceeded its maximum total code size. For more information, see [Lambda quotas](#).

HTTP Status Code: 400

CodeVerificationFailedException

The code signature failed one or more of the validation checks for signature mismatch or expiry, and the code signing policy is set to ENFORCE. Lambda blocks the deployment.

HTTP Status Code: 400

InvalidCodeSignatureException

The code signature failed the integrity check. If the integrity check fails, then Lambda blocks deployment, even if the code signing policy is set to WARN.

HTTP Status Code: 400

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

CreateFunctionUrlConfig

Creates a Lambda function URL with the specified configuration parameters. A function URL is a dedicated HTTP(S) endpoint that you can use to invoke your function.

Request Syntax

```
POST /2021-10-31/functions/FunctionName/url?Qualifier=Qualifier HTTP/1.1
Content-type: application/json

{
  "AuthTypeCors": {
    "AllowCredentials": boolean,
    "AllowHeaders": [ "string" ],
    "AllowMethods": [ "string" ],
    "AllowOrigins": [ "string" ],
    "ExposeHeaders": [ "string" ],
    "MaxAge": number
  },
  "InvokeMode": "string"
}
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName (p. 1178)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

Qualifier (p. 1178)

The alias name.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (^\\$LATEST\$)|((?!^[\d-]).+)([\d-]+)

Request Body

The request accepts the following data in JSON format.

[AuthType \(p. 1178\)](#)

The type of authentication that your function URL uses. Set to AWS_IAM if you want to restrict access to authenticated users only. Set to NONE if you want to bypass IAM authentication to create a public endpoint. For more information, see [Security and auth model for Lambda function URLs](#).

Type: String

Valid Values: NONE | AWS_IAM

Required: Yes

[Cors \(p. 1178\)](#)

The [cross-origin resource sharing \(CORS\)](#) settings for your function URL.

Type: [Cors \(p. 1410\)](#) object

Required: No

[InvokeMode \(p. 1178\)](#)

Use one of the following options:

- BUFFERED – This is the default option. Lambda invokes your function using the Invoke API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.
- RESPONSE_STREAM – Your function streams payload results as they become available. Lambda invokes your function using the InvokeWithResponseStream API operation. The maximum response payload size is 20 MB, however, you can [request a quota increase](#).

Type: String

Valid Values: BUFFERED | RESPONSE_STREAM

Required: No

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
  "AuthType": "string",
  "Cors": {
    "AllowCredentials": boolean,
    "AllowHeaders": [ "string" ],
    "AllowMethods": [ "string" ],
    "AllowOrigins": [ "string" ],
    "ExposeHeaders": [ "string" ],
    "MaxAge": number
  },
  "CreationTime": "string",
  "FunctionArn": "string",
  "FunctionUrl": "string",
  "InvokeMode": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

[AuthType \(p. 1179\)](#)

The type of authentication that your function URL uses. Set to AWS_IAM if you want to restrict access to authenticated users only. Set to NONE if you want to bypass IAM authentication to create a public endpoint. For more information, see [Security and auth model for Lambda function URLs](#).

Type: String

Valid Values: NONE | AWS_IAM

[Cors \(p. 1179\)](#)

The [cross-origin resource sharing \(CORS\)](#) settings for your function URL.

Type: [Cors \(p. 1410\)](#) object

[CreationTime \(p. 1179\)](#)

When the function URL was created, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[FunctionArn \(p. 1179\)](#)

The Amazon Resource Name (ARN) of your function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$LATEST|[a-zA-Z0-9-_]+))?

[FunctionUrl \(p. 1179\)](#)

The HTTP URL endpoint for your function.

Type: String

Length Constraints: Minimum length of 40. Maximum length of 100.

[InvokeMode \(p. 1179\)](#)

Use one of the following options:

- BUFFERED – This is the default option. Lambda invokes your function using the Invoke API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.
- RESPONSE_STREAM – Your function streams payload results as they become available. Lambda invokes your function using the InvokeWithResponseStream API operation. The maximum response payload size is 20 MB, however, you can [request a quota increase](#).

Type: String

Valid Values: BUFFERED | RESPONSE_STREAM

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DeleteAlias

Deletes a Lambda function [alias](#).

Request Syntax

```
DELETE /2015-03-31/functions/FunctionName/aliases/Name HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1182\)](#)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Name \(p. 1182\)](#)

The name of the alias.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (?![0-9]+\\$)([a-zA-Z0-9-_]+)

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DeleteCodeSigningConfig

Deletes the code signing configuration. You can delete the code signing configuration only if no function is using it.

Request Syntax

```
DELETE /2020-04-22/code-signing-configs/CodeSigningConfigArn HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[CodeSigningConfigArn \(p. 1184\)](#)

The The Amazon Resource Name (ARN) of the code signing configuration.

Length Constraints: Maximum length of 200.

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}((-gov)|(-iso(b?)))?- [a-z]+-\d{1}:\d{12}:code-signing-config:csc-[a-z0-9]{17}

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DeleteEventSourceMapping

Deletes an [event source mapping](#). You can get the identifier of a mapping from the output of [ListEventSourceMappings \(p. 1279\)](#).

When you delete an event source mapping, it enters a Deleting state and might not be completely deleted for several seconds.

Request Syntax

```
DELETE /2015-03-31/event-source-mappings/UUID HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[UUID \(p. 1186\)](#)

The identifier of the event source mapping.

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 202
Content-type: application/json

{
    "AmazonManagedKafkaEventSourceConfig": {
        "ConsumerGroupIdBatchSize": number,
    "BisectBatchOnFunctionError": boolean,
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "DocumentDBEventSourceConfig": {
        "CollectionName": "string",
        "DatabaseName": "string",
        "FullDocument": "string"
    },
    "EventSourceArn": "string",
    "FilterCriteria": {
        "Filters": [
            {
                "Pattern": "string"
            }
        ]
    },
}
```

```

"FunctionArn": "string",
"FunctionResponseTypes": [ "string" ],
"LastModified": number,
"LastProcessingResult": "string",
"MaximumBatchingWindowInSeconds": number,
"MaximumRecordAgeInSeconds": number,
"MaximumRetryAttempts": number,
"ParallelizationFactor": number,
"Queues": [ "string" ],
"ScalingConfig": {
    "MaximumConcurrency": number
},
"SelfManagedEventSource": {
    "Endpoints": {
        "string" : [ "string" ]
    }
},
"SelfManagedKafkaEventSourceConfig": {
    "ConsumerGroupId": "string"
},
"SourceAccessConfigurations": [
    {
        "Type": "string",
        "URI": "string"
    }
],
"StartingPosition": "string",
"StartingPositionTimestamp": number,
"State": "string",
"StateTransitionReason": "string",
"Topics": [ "string" ],
" TumblingWindowInSeconds": number,
"UUID": "string"
}

```

Response Elements

If the action is successful, the service sends back an HTTP 202 response.

The following data is returned in JSON format by the service.

[AmazonManagedKafkaEventSourceConfig \(p. 1186\)](#)

Specific configuration settings for an Amazon Managed Streaming for Apache Kafka (Amazon MSK) event source.

Type: [AmazonManagedKafkaEventSourceConfig \(p. 1405\)](#) object

[BatchSize \(p. 1186\)](#)

The maximum number of records in each batch that Lambda pulls from your stream or queue and sends to your function. Lambda passes all of the records in the batch to the function in a single call, up to the payload limit for synchronous invocation (6 MB).

Default value: Varies by service. For Amazon SQS, the default is 10. For all other services, the default is 100.

Related setting: When you set `BatchSize` to a value greater than 10, you must set `MaximumBatchingWindowInSeconds` to at least 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

[BisectBatchOnFunctionError \(p. 1186\)](#)

(Kinesis and DynamoDB Streams only) If the function returns an error, split the batch in two and retry. The default value is false.

Type: Boolean

[DestinationConfig \(p. 1186\)](#)

(Kinesis and DynamoDB Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: [DestinationConfig \(p. 1413\)](#) object

[DocumentDBEventSourceConfig \(p. 1186\)](#)

Specific configuration settings for a DocumentDB event source.

Type: [DocumentDBEventSourceConfig \(p. 1414\)](#) object

[EventSourceArn \(p. 1186\)](#)

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-.])+:(a-z){2}(-gov)?-[a-zA-Z]+\d{1})?:(\d{12})?:(.*?)

[FilterCriteria \(p. 1186\)](#)

An object that defines the filter criteria that determine whether Lambda should process an event. For more information, see [Lambda event filtering](#).

Type: [FilterCriteria \(p. 1426\)](#) object

[FunctionArn \(p. 1186\)](#)

The ARN of the Lambda function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-zA-Z]{2}(-gov)?-[a-zA-Z]+\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$\\$LATEST|[a-zA-Z0-9-_]+))?

[FunctionResponseTypes \(p. 1186\)](#)

(Kinesis, DynamoDB Streams, and Amazon SQS) A list of current response type enums applied to the event source mapping.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 1 item.

Valid Values: ReportBatchItemFailures

[LastModified \(p. 1186\)](#)

The date that the event source mapping was last updated or that its state changed, in Unix time seconds.

Type: Timestamp

[LastProcessingResult \(p. 1186\)](#)

The result of the last Lambda invocation of your function.

Type: String

MaximumBatchingWindowInSeconds (p. 1186)

The maximum amount of time, in seconds, that Lambda spends gathering records before invoking the function. You can configure MaximumBatchingWindowInSeconds to any value from 0 seconds to 300 seconds in increments of seconds.

For streams and Amazon SQS event sources, the default batching window is 0 seconds.

For Amazon MSK, Self-managed Apache Kafka, Amazon MQ, and DocumentDB event sources, the default batching window is 500 ms. Note that because you can only change MaximumBatchingWindowInSeconds in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

Related setting: For streams and Amazon SQS event sources, when you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

MaximumRecordAgeInSeconds (p. 1186)

(Kinesis and DynamoDB Streams only) Discard records older than the specified age. The default value is -1, which sets the maximum age to infinite. When the value is set to infinite, Lambda never discards old records.

Note

The minimum valid value for maximum record age is 60s. Although values less than 60 and greater than -1 fall within the parameter's absolute range, they are not allowed

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 604800.

MaximumRetryAttempts (p. 1186)

(Kinesis and DynamoDB Streams only) Discard records after the specified number of retries. The default value is -1, which sets the maximum number of retries to infinite. When MaximumRetryAttempts is infinite, Lambda retries failed records until the record expires in the event source.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

ParallelizationFactor (p. 1186)

(Kinesis and DynamoDB Streams only) The number of batches to process concurrently from each shard. The default value is 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10.

Queues (p. 1186)

(Amazon MQ) The name of the Amazon MQ broker destination queue to consume.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 1000.

Pattern: [\\s\\S]*

[ScalingConfig \(p. 1186\)](#)

(Amazon SQS only) The scaling configuration for the event source. For more information, see [Configuring maximum concurrency for Amazon SQS event sources](#).

Type: [ScalingConfig \(p. 1458\)](#) object

[SelfManagedEventSource \(p. 1186\)](#)

The self-managed Apache Kafka cluster for your event source.

Type: [SelfManagedEventSource \(p. 1459\)](#) object

[SelfManagedKafkaEventSourceConfig \(p. 1186\)](#)

Specific configuration settings for a self-managed Apache Kafka event source.

Type: [SelfManagedKafkaEventSourceConfig \(p. 1460\)](#) object

[SourceAccessConfigurations \(p. 1186\)](#)

An array of the authentication protocol, VPC components, or virtual host to secure and define your event source.

Type: Array of [SourceAccessConfiguration \(p. 1463\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

[StartingPosition \(p. 1186\)](#)

The position in a stream from which to start reading. Required for Amazon Kinesis, Amazon DynamoDB, and Amazon MSK stream sources. AT_TIMESTAMP is supported only for Amazon Kinesis streams and Amazon DocumentDB.

Type: String

Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

[StartingPositionTimestamp \(p. 1186\)](#)

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds.

Type: Timestamp

[State \(p. 1186\)](#)

The state of the event source mapping. It can be one of the following: Creating, Enabling, Enabled, Disabling, Disabled, Updating, or Deleting.

Type: String

[StateTransitionReason \(p. 1186\)](#)

Indicates whether a user or Lambda made the last change to the event source mapping.

Type: String

[Topics \(p. 1186\)](#)

The name of the Kafka topic.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 249.

Pattern: ^[^.]([a-zA-Z0-9\-.]+)+

TumblingWindowInSeconds (p. 1186)

(Kinesis and DynamoDB Streams only) The duration in seconds of a processing window for DynamoDB and Kinesis Streams event sources. A value of 0 seconds indicates no tumbling window.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

UUID (p. 1186)

The identifier of the event source mapping.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceInUseException

The operation conflicts with the resource's availability. For example, you tried to update an event source mapping in the CREATING state, or you tried to delete an event source mapping currently UPDATING.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)

- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DeleteFunction

Deletes a Lambda function. To delete a specific function version, use the `Qualifier` parameter. Otherwise, all versions and aliases are deleted.

To delete Lambda event source mappings that invoke a function, use [DeleteEventSourceMapping \(p. 1186\)](#). For AWS services and resources that invoke your function directly, delete the trigger in the service where you originally configured it.

Request Syntax

```
DELETE /2015-03-31/functions/FunctionName?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1193\)](#)

The name of the Lambda function or version.

Name formats

- **Function name** – my-function (name-only), my-function:1 (with version).
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1193\)](#)

Specify a version to delete. You can't delete a version that an alias references.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$-_]+)

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DeleteFunctionCodeSigningConfig

Removes the code signing configuration from the function.

Request Syntax

```
DELETE /2020-06-30/functions/FunctionName/code-signing-config HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName (p. 1195)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

CodeSigningConfigNotFoundException

The specified code signing configuration does not exist.

HTTP Status Code: 404

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DeleteFunctionConcurrency

Removes a concurrent execution limit from a function.

Request Syntax

```
DELETE /2017-10-31/functions/FunctionName/concurrency HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName (p. 1197)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DeleteFunctionEventInvokeConfig

Deletes the configuration for asynchronous invocation for a function, version, or alias.

To configure options for asynchronous invocation, use [PutFunctionEventInvokeConfig \(p. 1330\)](#).

Request Syntax

```
DELETE /2019-09-25/functions/FunctionName/event-invoke-config?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1199\)](#)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** - my-function (name-only), my-function:v1 (with alias).
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1199\)](#)

A version number or alias name.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$-_]+)

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DeleteFunctionUrlConfig

Deletes a Lambda function URL. When you delete a function URL, you can't recover it. Creating a new function URL results in a different URL address.

Request Syntax

```
DELETE /2021-10-31/functions/FunctionName/url?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1201\)](#)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1201\)](#)

The alias name.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (^\\$LATEST\$)|((?!^[\d-]+\$)([a-zA-Z0-9-_]+))

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DeleteLayerVersion

Deletes a version of an [AWS Lambda layer](#). Deleted versions can no longer be viewed or added to functions. To avoid breaking functions, a copy of the version remains in Lambda until no functions refer to it.

Request Syntax

```
DELETE /2018-10-31/layers/LayerName/versions/VersionNumber HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[LayerName \(p. 1203\)](#)

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_+])|[a-zA-Z0-9-_]+

Required: Yes

[VersionNumber \(p. 1203\)](#)

The version number.

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

DeleteProvisionedConcurrencyConfig

Deletes the provisioned concurrency configuration for a function.

Request Syntax

```
DELETE /2019-09-30/functions/FunctionName/provisioned-concurrency?Qualifier=Qualifier
HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1205\)](#)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1205\)](#)

The version number or alias name.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$-_]+)

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetAccountSettings

Retrieves details about your account's [limits](#) and usage in an AWS Region.

Request Syntax

```
GET /2016-08-19/account-settings/ HTTP/1.1
```

URI Request Parameters

The request does not use any URI parameters.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "AccountLimit": {
    "CodeSizeUnzipped": number,
    "CodeSizeZipped": number,
    "ConcurrentExecutions": number,
    "TotalCodeSize": number,
    "UnreservedConcurrentExecutions": number
  },
  "AccountUsage": {
    "FunctionCount": number,
    "TotalCodeSize": number
  }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[AccountLimit \(p. 1207\)](#)

Limits that are related to concurrency and code storage.

Type: [AccountLimit \(p. 1399\)](#) object

[AccountUsage \(p. 1207\)](#)

The number of functions and amount of storage in use.

Type: [AccountUsage \(p. 1400\)](#) object

Errors

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetAlias

Returns details about a Lambda function [alias](#).

Request Syntax

```
GET /2015-03-31/functions/FunctionName/aliases/Name HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1209\)](#)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_+]):(\$LATEST|[a-zA-Z0-9-_+]))?

Required: Yes

[Name \(p. 1209\)](#)

The name of the alias.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (?![0-9]+\$)([a-zA-Z0-9-_]+)

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "AliasArn": "string",
  "Description": "string",
  "FunctionVersion": "string",
```

```
"Name": "string",
"RevisionId": "string",
"RoutingConfig": {
    "AdditionalVersionWeights": {
        "string" : number
    }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[AliasArn \(p. 1209\)](#)

The Amazon Resource Name (ARN) of the alias.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$\\$LATEST|[a-zA-Z0-9-_]+))?

[Description \(p. 1209\)](#)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[FunctionVersion \(p. 1209\)](#)

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$\\$LATEST|[0-9]+)

[Name \(p. 1209\)](#)

The name of the alias.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (?!^[\d-]+)([a-zA-Z0-9-_]+)

[RevisionId \(p. 1209\)](#)

A unique identifier that changes when you update the alias.

Type: String

[RoutingConfig \(p. 1209\)](#)

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 1403\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetCodeSigningConfig

Returns information about the specified code signing configuration.

Request Syntax

```
GET /2020-04-22/code-signing-configs/CodeSigningConfigArn HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[CodeSigningConfigArn \(p. 1212\)](#)

The Amazon Resource Name (ARN) of the code signing configuration.

Length Constraints: Maximum length of 200.

Pattern: arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}((-gov)|(-iso(b?)))?-([a-z]+-\d{1}):\d{12}:code-signing-config:csc-[a-zA-Z0-9]{17}

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "CodeSigningConfigAllowedPublishersSigningProfileVersionArnsCodeSigningConfigArnCodeSigningConfigIdCodeSigningPoliciesUntrustedArtifactOnDeploymentDescriptionLastModified
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[CodeSigningConfig \(p. 1212\)](#)

The code signing configuration

Type: [CodeSigningConfig \(p. 1406\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetEventSourceMapping

Returns details about an event source mapping. You can get the identifier of a mapping from the output of [ListEventSourceMappings \(p. 1279\)](#).

Request Syntax

```
GET /2015-03-31/event-source-mappings/UUID HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[UUID \(p. 1214\)](#)

The identifier of the event source mapping.

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "AmazonManagedKafkaEventSourceConfig": {
        "ConsumerGroupIdBatchSize": number,
    "BisectBatchOnFunctionError": boolean,
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "DocumentDBEventSourceConfig": {
        "CollectionName": "string",
        "DatabaseName": "string",
        "FullDocument": "string"
    },
    "EventSourceArn": "string",
    "FilterCriteria": {
        "Filters": [
            {
                "Pattern": "string"
            }
        ]
    },
    "FunctionArn": "string",
    "FunctionResponseTypes": [ "string" ],
    "LastModified": number,
    "LastProcessingResult": "string",
}
```

```

    "MaximumBatchingWindowInSeconds": number,
    "MaximumRecordAgeInSeconds": number,
    "MaximumRetryAttempts": number,
    "ParallelizationFactor": number,
    "Queues": [ "string" ],
    "ScalingConfig": {
        "MaximumConcurrency": number
    },
    "SelfManagedEventSource": {
        "Endpoints": {
            "string": [ "string" ]
        }
    },
    "SelfManagedKafkaEventSourceConfig": {
        "ConsumerGroupId": "string"
    },
    "SourceAccessConfigurations": [
        {
            "Type": "string",
            "URI": "string"
        }
    ],
    "StartingPosition": "string",
    "StartingPositionTimestamp": number,
    "State": "string",
    "StateTransitionReason": "string",
    "Topics": [ "string" ],
    "TumblingWindowInSeconds": number,
    "UUID": "string"
}

```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[AmazonManagedKafkaEventSourceConfig \(p. 1214\)](#)

Specific configuration settings for an Amazon Managed Streaming for Apache Kafka (Amazon MSK) event source.

Type: [AmazonManagedKafkaEventSourceConfig \(p. 1405\)](#) object

[BatchSize \(p. 1214\)](#)

The maximum number of records in each batch that Lambda pulls from your stream or queue and sends to your function. Lambda passes all of the records in the batch to the function in a single call, up to the payload limit for synchronous invocation (6 MB).

Default value: Varies by service. For Amazon SQS, the default is 10. For all other services, the default is 100.

Related setting: When you set `BatchSize` to a value greater than 10, you must set `MaximumBatchingWindowInSeconds` to at least 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

[BisectBatchOnFunctionError \(p. 1214\)](#)

(Kinesis and DynamoDB Streams only) If the function returns an error, split the batch in two and retry. The default value is false.

Type: Boolean

[DestinationConfig \(p. 1214\)](#)

(Kinesis and DynamoDB Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: [DestinationConfig \(p. 1413\)](#) object

[DocumentDBEventSourceConfig \(p. 1214\)](#)

Specific configuration settings for a DocumentDB event source.

Type: [DocumentDBEventSourceConfig \(p. 1414\)](#) object

[EventSourceArn \(p. 1214\)](#)

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-.])+:(a-z{2}(-gov)?-[a-z]+\d{1})?:(\d{12})?:(.*?)`

[FilterCriteria \(p. 1214\)](#)

An object that defines the filter criteria that determine whether Lambda should process an event.

For more information, see [Lambda event filtering](#).

Type: [FilterCriteria \(p. 1426\)](#) object

[FunctionArn \(p. 1214\)](#)

The ARN of the Lambda function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$\$LATEST|[a-zA-Z0-9-_]+))?`

[FunctionResponseTypes \(p. 1214\)](#)

(Kinesis, DynamoDB Streams, and Amazon SQS) A list of current response type enums applied to the event source mapping.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 1 item.

Valid Values: ReportBatchItemFailures

[LastModified \(p. 1214\)](#)

The date that the event source mapping was last updated or that its state changed, in Unix time seconds.

Type: Timestamp

[LastProcessingResult \(p. 1214\)](#)

The result of the last Lambda invocation of your function.

Type: String

[MaximumBatchingWindowInSeconds \(p. 1214\)](#)

The maximum amount of time, in seconds, that Lambda spends gathering records before invoking the function. You can configure MaximumBatchingWindowInSeconds to any value from 0 seconds to 300 seconds in increments of seconds.

For streams and Amazon SQS event sources, the default batching window is 0 seconds. For Amazon MSK, Self-managed Apache Kafka, Amazon MQ, and DocumentDB event sources, the default batching window is 500 ms. Note that because you can only change MaximumBatchingWindowInSeconds in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

Related setting: For streams and Amazon SQS event sources, when you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

[MaximumRecordAgeInSeconds \(p. 1214\)](#)

(Kinesis and DynamoDB Streams only) Discard records older than the specified age. The default value is -1, which sets the maximum age to infinite. When the value is set to infinite, Lambda never discards old records.

Note

The minimum valid value for maximum record age is 60s. Although values less than 60 and greater than -1 fall within the parameter's absolute range, they are not allowed

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 604800.

[MaximumRetryAttempts \(p. 1214\)](#)

(Kinesis and DynamoDB Streams only) Discard records after the specified number of retries. The default value is -1, which sets the maximum number of retries to infinite. When MaximumRetryAttempts is infinite, Lambda retries failed records until the record expires in the event source.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

[ParallelizationFactor \(p. 1214\)](#)

(Kinesis and DynamoDB Streams only) The number of batches to process concurrently from each shard. The default value is 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10.

[Queues \(p. 1214\)](#)

(Amazon MQ) The name of the Amazon MQ broker destination queue to consume.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 1000.

Pattern: `[\s\S]*`

[ScalingConfig \(p. 1214\)](#)

(Amazon SQS only) The scaling configuration for the event source. For more information, see [Configuring maximum concurrency for Amazon SQS event sources](#).

Type: [ScalingConfig \(p. 1458\)](#) object

[**SelfManagedEventSource \(p. 1214\)**](#)

The self-managed Apache Kafka cluster for your event source.

Type: [SelfManagedEventSource \(p. 1459\)](#) object

[**SelfManagedKafkaEventSourceConfig \(p. 1214\)**](#)

Specific configuration settings for a self-managed Apache Kafka event source.

Type: [SelfManagedKafkaEventSourceConfig \(p. 1460\)](#) object

[**SourceAccessConfigurations \(p. 1214\)**](#)

An array of the authentication protocol, VPC components, or virtual host to secure and define your event source.

Type: Array of [SourceAccessConfiguration \(p. 1463\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

[**StartingPosition \(p. 1214\)**](#)

The position in a stream from which to start reading. Required for Amazon Kinesis, Amazon DynamoDB, and Amazon MSK stream sources. AT_TIMESTAMP is supported only for Amazon Kinesis streams and Amazon DocumentDB.

Type: String

Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

[**StartingPositionTimestamp \(p. 1214\)**](#)

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds.

Type: Timestamp

[**State \(p. 1214\)**](#)

The state of the event source mapping. It can be one of the following: Creating, Enabling, Enabled, Disabling, Disabled, Updating, or Deleting.

Type: String

[**StateTransitionReason \(p. 1214\)**](#)

Indicates whether a user or Lambda made the last change to the event source mapping.

Type: String

[**Topics \(p. 1214\)**](#)

The name of the Kafka topic.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 249.

Pattern: ^[^.]([a-zA-Z0-9\-_\.]+)+

[**TumblingWindowInSeconds \(p. 1214\)**](#)

(Kinesis and DynamoDB Streams only) The duration in seconds of a processing window for DynamoDB and Kinesis Streams event sources. A value of 0 seconds indicates no tumbling window.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

UUID (p. 1214)

The identifier of the event source mapping.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetFunction

Returns information about the function or function version, with a link to download the deployment package that's valid for 10 minutes. If you specify a function version, only details that are specific to that version are returned.

Request Syntax

```
GET /2015-03-31/functions/FunctionName?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1220\)](#)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** – my-function (name-only), my-function:v1 (with alias).
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1220\)](#)

Specify a version or alias to get details about a published version of the function.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$-_]+)

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "Code": {
        "ImageUriLocationRepositoryTypeResolvedImageUri
```

```
},
"Concurrency": {
    "ReservedConcurrentExecutions": number
},
"Configuration": {
    "Architectures": [ "string" ],
    "CodeSha256": "string",
    "CodeSize": number,
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
    "Description": "string",
    "Environment": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "Variables": {
            "string" : "string"
        }
    },
    "EphemeralStorage": {
        "Size": number
}
},
"FileSystemConfigs": [
    {
        "Arn": "string",
        "LocalMountPath": "string"
    }
],
"FunctionArn": "string",
"FunctionName": "string",
"Handler": "string",
"ImageConfigResponse": {
    "Error": {
        "ErrorCode": "string",
        "Message": "string"
    },
    "ImageConfig": {
        "Command": [ "string" ],
        "EntryPoint": [ "string" ],
        "WorkingDirectory": "string"
    }
},
"KMSKeyArn": "string",
"LastModified": "string",
"LastUpdateStatus": "string",
"LastUpdateStatusReason": "string",
"LastUpdateStatusReasonCode": "string",
"Layers": [
    {
        "Arn": "string",
        "CodeSize": number,
        "SigningJobArn": "string",
        "SigningProfileVersionArn": "string"
    }
],
"MasterArn": "string",
"MemorySize": number,
"PackageType": "string",
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"RuntimeVersionConfig": {
    "Error": {
        "ErrorCode": "string",
        "Message": "string"
    }
}
```

```
        "Message": "string"
    },
    "RuntimeVersionArn": "string"
},
"SigningJobArn": "string",
"SigningProfileVersionArn": "string",
"SnapStart": {
    "ApplyOn": "string",
    "OptimizationStatus": "string"
},
"State": "string",
"StateReason": "string",
"StateReasonCode": "string",
"Timeout": number,
"TracingConfig": {
    "Mode": "string"
},
"Version": "string",
"VpcConfig": {
    "SecurityGroupIds": [ "string" ],
    "SubnetIds": [ "string" ],
    "VpcId": "string"
}
},
"Tags": {
    "string" : "string"
}
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Code \(p. 1220\)](#)

The deployment package of the function or version.

Type: [FunctionCodeLocation \(p. 1429\)](#) object

[Concurrency \(p. 1220\)](#)

The function's [reserved concurrency](#).

Type: [Concurrency \(p. 1409\)](#) object

[Configuration \(p. 1220\)](#)

The configuration of the function or version.

Type: [FunctionConfiguration \(p. 1430\)](#) object

[Tags \(p. 1220\)](#)

The function's [tags](#).

Type: String to string map

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetFunctionCodeSigningConfig

Returns the code signing configuration for the specified function.

Request Syntax

```
GET /2020-06-30/functions/FunctionName/code-signing-config HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName (p. 1224)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "CodeSigningConfigArn": "string",
  "FunctionName": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

CodeSigningConfigArn (p. 1224)

The The Amazon Resource Name (ARN) of the code signing configuration.

Type: String

Length Constraints: Maximum length of 200.

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}((-gov)|(-iso(b?)))?- [a-z]+-\d{1}:\d{12}:code-signing-config:csc-[a-zA-Z0-9]{17}

[FunctionName \(p. 1224\)](#)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)

- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetFunctionConcurrency

Returns details about the reserved concurrency configuration for a function. To set a concurrency limit for a function, use [PutFunctionConcurrency \(p. 1327\)](#).

Request Syntax

```
GET /2019-09-30/functions/FunctionName/concurrency HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName (p. 1227)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "ReservedConcurrentExecutions": number
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

ReservedConcurrentExecutions (p. 1227)

The number of simultaneous executions that are reserved for the function.

Type: Integer

Valid Range: Minimum value of 0.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetFunctionConfiguration

Returns the version-specific settings of a Lambda function or version. The output includes only options that can vary between versions of a function. To modify these settings, use [UpdateFunctionConfiguration \(p. 1377\)](#).

To get all of a function's details, including function-level settings, use [GetFunction \(p. 1220\)](#).

Request Syntax

```
GET /2015-03-31/functions/FunctionName/configuration?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1229\)](#)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** – my-function (name-only), my-function:v1 (with alias).
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1229\)](#)

Specify a version or alias to get details about a published version of the function.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$-_]+)

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "Architectures": [ "string" ],
    "CodeSha256": "string",
    "CodeSize": number,
```

```
        "DeadLetterConfig": {
            "TargetArn": "string"
        },
        "Description": "string",
        "Environment": {
            "Error": {
                "ErrorCode": "string",
                "Message": "string"
            },
            "Variables": {
                "string" : "string"
            }
        },
        "EphemeralStorage": {
            "Size": number
        },
        "FileSystemConfigs": [
            {
                "Arn": "string",
                "LocalMountPath": "string"
            }
        ],
        "FunctionArn": "string",
        "FunctionName": "string",
        "Handler": "string",
        "ImageConfigResponse": {
            "Error": {
                "ErrorCode": "string",
                "Message": "string"
            },
            "ImageConfig": {
                "Command": [ "string" ],
                "EntryPoint": [ "string" ],
                "WorkingDirectory": "string"
            }
        },
        "KMSKeyArn": "string",
        "LastModified": "string",
        "LastUpdateStatus": "string",
        "LastUpdateStatusReason": "string",
        "LastUpdateStatusReasonCode": "string",
        "Layers": [
            {
                "Arn": "string",
                "CodeSize": number,
                "SigningJobArn": "string",
                "SigningProfileVersionArn": "string"
            }
        ],
        "MasterArn": "string",
        "MemorySize": number,
        "PackageType": "string",
        "RevisionId": "string",
        "Role": "string",
        "Runtime": "string",
        "RuntimeVersionConfig": {
            "Error": {
                "ErrorCode": "string",
                "Message": "string"
            },
            "RuntimeVersionArn": "string"
        },
        "SigningJobArn": "string",
        "SigningProfileVersionArn": "string",
        "SnapStart": {
            "ApplvOn": "string"
        }
    }
}
```

```
        "OptimizationStatus": "string"
    },
    "State": "string",
    "StateReason": "string",
    "StateReasonCode": "string",
    "Timeout": number,
    "TracingConfig": {
        "Mode": "string"
    },
    "Version": "string",
    "VpcConfig": {
        "SecurityGroupIds": [ "string" ],
        "SubnetIds": [ "string" ],
        "VpcId": "string"
    }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Architectures \(p. 1229\)](#)

The instruction set architecture that the function supports. Architecture is a string array with one of the valid values. The default architecture value is x86_64.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: x86_64 | arm64

[CodeSha256 \(p. 1229\)](#)

The SHA256 hash of the function's deployment package.

Type: String

[CodeSize \(p. 1229\)](#)

The size of the function's deployment package, in bytes.

Type: Long

[DeadLetterConfig \(p. 1412\)](#) object

[Description \(p. 1229\)](#)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[Environment \(p. 1229\)](#)

The function's [environment variables](#). Omitted from AWS CloudTrail logs.

Type: [EnvironmentResponse \(p. 1417\)](#) object

[EphemeralStorage \(p. 1229\)](#)

The size of the function's /tmp directory in MB. The default value is 512, but it can be any whole number between 512 and 10,240 MB.

Type: [EphemeralStorage \(p. 1418\)](#) object

[FileSystemConfigs \(p. 1229\)](#)

Connection settings for an [Amazon EFS file system](#).

Type: Array of [FileSystemConfig \(p. 1424\)](#) objects

Array Members: Maximum number of 1 item.

[FunctionArn \(p. 1229\)](#)

The function's Amazon Resource Name (ARN).

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[FunctionName \(p. 1229\)](#)

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[Handler \(p. 1229\)](#)

The function that Lambda calls to begin running your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

[ImageConfigResponse \(p. 1229\)](#)

The function's image configuration values.

Type: [ImageConfigResponse \(p. 1442\)](#) object

[KMSKeyArn \(p. 1229\)](#)

The AWS KMS key that's used to encrypt the function's [environment variables](#). When [Lambda SnapStart](#) is activated, this key is also used to encrypt the function's snapshot. This key is returned only if you've configured a customer managed key.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-_\.]+\.:*)|()`

[LastModified \(p. 1229\)](#)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[LastUpdateStatus \(p. 1229\)](#)

The status of the last update that was performed on the function. This is first set to Successful after function creation completes.

Type: String

Valid Values: Successful | Failed | InProgress

[LastUpdateStatusReason \(p. 1229\)](#)

The reason for the last update that was performed on the function.

Type: String

[LastUpdateStatusReasonCode \(p. 1229\)](#)

The reason code for the last update that was performed on the function.

Type: String

Valid Values: EniLimitExceeded | InsufficientRolePermissions | InvalidConfiguration | InternalError | SubnetOutOfIPAddresses | InvalidSubnet | InvalidSecurityGroup | ImageDeleted | ImageAccessDenied | InvalidImage | KMSKeyAccessDenied | KMSKeyNotFound | InvalidStateKMSKey | DisabledKMSKey | EFSIOError | EFSMountConnectivityError | EFSMountFailure | EFSMountTimeout | InvalidRuntime | InvalidZipFileException | FunctionError

[Layers \(p. 1229\)](#)

The function's [layers](#).

Type: Array of [Layer \(p. 1446\)](#) objects

[MasterArn \(p. 1229\)](#)

For Lambda@Edge functions, the ARN of the main function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?

[MemorySize \(p. 1229\)](#)

The amount of memory available to the function at runtime.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 10240.

[PackageType \(p. 1229\)](#)

The type of deployment package. Set to Image for container image and set Zip for .zip file archive.

Type: String

Valid Values: Zip | Image

[RevisionId \(p. 1229\)](#)

The latest updated revision of the function or alias.

Type: String

[Role \(p. 1229\)](#)

The function's execution role.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@-_/]+

[Runtime \(p. 1229\)](#)

The identifier of the function's [runtime](#). Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

[RuntimeVersionConfig \(p. 1229\)](#)

The ARN of the runtime and any errors that occurred.

Type: [RuntimeVersionConfig \(p. 1456\)](#) object

[SigningJobArn \(p. 1229\)](#)

The ARN of the signing job.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-\-]+)([a-z]{2}(-gov)?-[a-z]+\-\d{1})?:(\d{12})?:(.*?)

[SigningProfileVersionArn \(p. 1229\)](#)

The ARN of the signing profile version.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-\-]+)([a-z]{2}(-gov)?-[a-z]+\-\d{1})?:(\d{12})?:(.*?)

[SnapStart \(p. 1229\)](#)

Set `ApplyOn` to `PublishedVersions` to create a snapshot of the initialized execution environment when you publish a function version. For more information, see [Improving startup performance with Lambda SnapStart](#).

Type: [SnapStartResponse \(p. 1462\)](#) object

[State \(p. 1229\)](#)

The current state of the function. When the state is `Inactive`, you can reactivate the function by invoking it.

Type: String

Valid Values: Pending | Active | Inactive | Failed

[StateReason \(p. 1229\)](#)

The reason for the function's current state.

Type: String

[StateReasonCode \(p. 1229\)](#)

The reason code for the function's current state. When the code is `Creating`, you can't invoke or modify the function.

Type: String

Valid Values: `Idle` | `Creating` | `Restoring` | `EniLimitExceeded` | `InsufficientRolePermissions` | `InvalidConfiguration` | `InternalError` | `SubnetOutOfIPAddresses` | `InvalidSubnet` | `InvalidSecurityGroup` | `ImageDeleted` | `ImageAccessDenied` | `InvalidImage` | `KMSKeyAccessDenied` | `KMSKeyNotFound` | `InvalidStateKMSKey` | `DisabledKMSKey` | `EFSIOError` | `EFSMountConnectivityError` | `EFSMountFailure` | `EFSMountTimeout` | `InvalidRuntime` | `InvalidZipFileException` | `FunctionError`

[Timeout \(p. 1229\)](#)

The amount of time in seconds that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

[TracingConfig \(p. 1229\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 1466\)](#) object

[Version \(p. 1229\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST| [0-9]+)`

[VpcConfig \(p. 1229\)](#)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 1468\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetFunctionEventInvokeConfig

Retrieves the configuration for asynchronous invocation for a function, version, or alias.

To configure options for asynchronous invocation, use [PutFunctionEventInvokeConfig \(p. 1330\)](#).

Request Syntax

```
GET /2019-09-25/functions/FunctionName/event-invoke-config?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1237\)](#)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** - my-function (name-only), my-function:v1 (with alias).
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1237\)](#)

A version number or alias name.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$-_]+)

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "DestinationConfig": {
    "OnFailure": {
      "DestinationOnSuccess": {
      "Destination
```

```
        },
        "FunctionArn": "string",
        "LastModified": number,
        "MaximumEventAgeInSeconds": number,
        "MaximumRetryAttempts": number
    }
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[DestinationConfig \(p. 1237\)](#)

A destination for events after they have been sent to a function for processing.

Destinations

- **Function** - The Amazon Resource Name (ARN) of a Lambda function.
- **Queue** - The ARN of a standard SQS queue.
- **Topic** - The ARN of a standard SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: [DestinationConfig \(p. 1413\)](#) object

[FunctionArn \(p. 1237\)](#)

The Amazon Resource Name (ARN) of the function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+\-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$\$LATEST|[a-zA-Z0-9-_]+))?`

[LastModified \(p. 1237\)](#)

The date and time that the configuration was last updated, in Unix time seconds.

Type: Timestamp

[MaximumEventAgeInSeconds \(p. 1237\)](#)

The maximum age of a request that Lambda sends to a function for processing.

Type: Integer

Valid Range: Minimum value of 60. Maximum value of 21600.

[MaximumRetryAttempts \(p. 1237\)](#)

The maximum number of times to retry when the function returns an error.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 2.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetFunctionUrlConfig

Returns details about a Lambda function URL.

Request Syntax

```
GET /2021-10-31/functions/FunctionName/url?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1240\)](#)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?(([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1240\)](#)

The alias name.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (^\\$LATEST\$)|((?!^[\d-]+\$)([a-zA-Z0-9-_]+))

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "AuthType": "string",
  "Cors": {
    "AllowCredentials": boolean,
    "AllowHeaders": [ "string" ],
    "AllowMethods": [ "string" ],
    "AllowOrigins": [ "string" ],
    "ExposeHeaders": [ "string" ],
  }
}
```

```
        "MaxAge": number
    },
    "CreationTimeFunctionArnFunctionUrlInvokeModeLastModifiedTime
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

AuthType (p. 1240)

The type of authentication that your function URL uses. Set to AWS_IAM if you want to restrict access to authenticated users only. Set to NONE if you want to bypass IAM authentication to create a public endpoint. For more information, see [Security and auth model for Lambda function URLs](#).

Type: String

Valid Values: NONE | AWS_IAM

Cors (p. 1240)

The [cross-origin resource sharing \(CORS\)](#) settings for your function URL.

Type: [Cors \(p. 1410\)](#) object

CreationTime (p. 1240)

When the function URL was created, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

FunctionArn (p. 1240)

The Amazon Resource Name (ARN) of your function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$\\$LATEST|[a-zA-Z0-9-_+]))?

FunctionUrl (p. 1240)

The HTTP URL endpoint for your function.

Type: String

Length Constraints: Minimum length of 40. Maximum length of 100.

InvokeMode (p. 1240)

Use one of the following options:

- **BUFFERED** – This is the default option. Lambda invokes your function using the Invoke API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.
- **RESPONSE_STREAM** – Your function streams payload results as they become available. Lambda invokes your function using the InvokeWithResponseStream API operation. The maximum response payload size is 20 MB, however, you can [request a quota increase](#).

Type: String

Valid Values: BUFFERED | RESPONSE_STREAM

LastModifiedTime (p. 1240)

When the function URL configuration was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetLayerVersion

Returns information about a version of an [AWS Lambda layer](#), with a link to download the layer archive that's valid for 10 minutes.

Request Syntax

```
GET /2018-10-31/layers/LayerName/versions/VersionNumber HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[LayerName \(p. 1243\)](#)

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_+])|([a-zA-Z0-9-_+])

Required: Yes

[VersionNumber \(p. 1243\)](#)

The version number.

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "CompatibleArchitectures": [ "string" ],
    "CompatibleRuntimes": [ "string" ],
    "Content": {
        "CodeSha256": "string",
        "CodeSize": number,
        "Location": "string",
        "SigningJobArn": "string",
        "SigningProfileVersionArn": "string"
    },
    "CreatedDate": "string",
    "Description": "string",
    "LayerArn": "string",
    "LayerVersionArn": "string",
    "LicenseInfo": "string",
    "Version": number
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[CompatibleArchitectures \(p. 1243\)](#)

A list of compatible [instruction set architectures](#).

Type: Array of strings

Array Members: Maximum number of 2 items.

Valid Values: x86_64 | arm64

[CompatibleRuntimes \(p. 1243\)](#)

The layer's compatible runtimes.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: Array of strings

Array Members: Maximum number of 15 items.

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

[Content \(p. 1243\)](#)

Details about the layer version.

Type: [LayerVersionContentOutput \(p. 1449\)](#) object

[CreatedDate \(p. 1243\)](#)

The date that the layer version was created, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[Description \(p. 1243\)](#)

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[LayerArn \(p. 1243\)](#)

The ARN of the layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+

[LayerVersionArn \(p. 1243\)](#)

The ARN of the layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+

[LicenseInfo \(p. 1243\)](#)

The layer's software license.

Type: String

Length Constraints: Maximum length of 512.

[Version \(p. 1243\)](#)

The version number.

Type: Long

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)

- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetLayerVersionByArn

Returns information about a version of an [AWS Lambda layer](#), with a link to download the layer archive that's valid for 10 minutes.

Request Syntax

```
GET /2018-10-31/layers?find=LayerVersion&Arn=Arn HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[Arn](#) (p. 1247)

The ARN of the layer version.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "CompatibleArchitectures": [ "string" ],
    "CompatibleRuntimes": [ "string" ],
    "Content": {
        "CodeSha256": "string",
        "CodeSize": number,
        "Location": "string",
        "SigningJobArn": "string",
        "SigningProfileVersionArn": "string"
    },
    "CreatedDate": "string",
    "Description": "string",
    "LayerArn": "string",
    "LayerVersionArn": "string",
    "LicenseInfo": "string",
    "Version": number
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[CompatibleArchitectures \(p. 1247\)](#)

A list of compatible [instruction set architectures](#).

Type: Array of strings

Array Members: Maximum number of 2 items.

Valid Values: x86_64 | arm64

[CompatibleRuntimes \(p. 1247\)](#)

The layer's compatible runtimes.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: Array of strings

Array Members: Maximum number of 15 items.

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

[Content \(p. 1247\)](#)

Details about the layer version.

Type: [LayerVersionContentOutput \(p. 1449\)](#) object

[CreatedDate \(p. 1247\)](#)

The date that the layer version was created, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[Description \(p. 1247\)](#)

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[LayerArn \(p. 1247\)](#)

The ARN of the layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+

[LayerVersionArn \(p. 1247\)](#)

The ARN of the layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+

LicensesInfo (p. 1247)

The layer's software license.

Type: String

Length Constraints: Maximum length of 512.

Version (p. 1247)

The version number.

Type: Long

Errors

InvalidArgumentException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetLayerVersionPolicy

Returns the permission policy for a version of an [AWS Lambda layer](#). For more information, see [AddLayerVersionPermission \(p. 1137\)](#).

Request Syntax

```
GET /2018-10-31/layers/LayerName/versions/VersionNumber/policy HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[LayerName \(p. 1250\)](#)

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (`arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+\:\d{12}:layer:[a-zA-Z0-9-_+]|[a-zA-Z0-9-_+]`)

Required: Yes

[VersionNumber \(p. 1250\)](#)

The version number.

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "Policy": "string",
    "RevisionId": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Policy \(p. 1250\)](#)

The policy document.

Type: String

[RevisionId \(p. 1250\)](#)

A unique identifier for the current revision of the policy.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetPolicy

Returns the [resource-based IAM policy](#) for a function, version, or alias.

Request Syntax

```
GET /2015-03-31/functions/FunctionName/policy?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1252\)](#)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** – my-function (name-only), my-function:v1 (with alias).
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1252\)](#)

Specify a version or alias to get the policy for that resource.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (|[a-zA-Z0-9\$-_]+)

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "Policy": "string",
    "RevisionId": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Policy (p. 1252)

The resource-based policy.

Type: String

RevisionId (p. 1252)

A unique identifier for the current revision of the policy.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetProvisionedConcurrencyConfig

Retrieves the provisioned concurrency configuration for a function's alias or version.

Request Syntax

```
GET /2019-09-30/functions/FunctionName/provisioned-concurrency?Qualifier=Qualifier HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1254\)](#)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1254\)](#)

The version number or alias name.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$_.-]+)

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "AllocatedProvisionedConcurrentExecutions": number,
    "AvailableProvisionedConcurrentExecutions": number,
    "LastModified": "string",
    "RequestedProvisionedConcurrentExecutions": number,
```

```
    "Status": "string",
    "StatusReason": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[AllocatedProvisionedConcurrentExecutions \(p. 1254\)](#)

The amount of provisioned concurrency allocated. When a weighted alias is used during linear and canary deployments, this value fluctuates depending on the amount of concurrency that is provisioned for the function versions.

Type: Integer

Valid Range: Minimum value of 0.

[AvailableProvisionedConcurrentExecutions \(p. 1254\)](#)

The amount of provisioned concurrency available.

Type: Integer

Valid Range: Minimum value of 0.

[LastModified \(p. 1254\)](#)

The date and time that a user last updated the configuration, in [ISO 8601 format](#).

Type: String

[RequestedProvisionedConcurrentExecutions \(p. 1254\)](#)

The amount of provisioned concurrency requested.

Type: Integer

Valid Range: Minimum value of 1.

[Status \(p. 1254\)](#)

The status of the allocation process.

Type: String

Valid Values: IN_PROGRESS | READY | FAILED

[StatusReason \(p. 1254\)](#)

For failed allocations, the reason that provisioned concurrency could not be allocated.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ProvisionedConcurrencyConfigNotFoundException

The specified configuration does not exist.

HTTP Status Code: 404

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

GetRuntimeManagementConfig

Retrieves the runtime management configuration for a function's version. If the runtime update mode is **Manual**, this includes the ARN of the runtime version and the runtime update mode. If the runtime update mode is **Auto** or **Function update**, this includes the runtime update mode and null is returned for the ARN. For more information, see [Runtime updates](#).

Request Syntax

```
GET /2021-07-20/functions/FunctionName/runtime-management-config?Qualifier=Qualifier
HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1257\)](#)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1257\)](#)

Specify a version of the function. This can be \$LATEST or a published version number. If no value is specified, the configuration for the \$LATEST version is returned.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (| [a-zA-Z0-9\$-_]+)

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "FunctionArn": "string",
```

```
    "RuntimeVersionArn": "string",
    "UpdateRuntimeOn": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[FunctionArn \(p. 1257\)](#)

The Amazon Resource Name (ARN) of your function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:(\$\$LATEST|[a-zA-Z0-9-_]+))?`

[RuntimeVersionArn \(p. 1257\)](#)

The ARN of the runtime the function is configured to use. If the runtime update mode is **Manual**, the ARN is returned, otherwise `null` is returned.

Type: String

Length Constraints: Minimum length of 26. Maximum length of 2048.

Pattern: `^arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}((-gov)|(-iso(b?)))?-+[a-z]+\d{1}:\runtime:.`+\$

[UpdateRuntimeOn \(p. 1257\)](#)

The current runtime update mode of the function.

Type: String

Valid Values: Auto | Manual | FunctionUpdate

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

Invoke

Invokes a Lambda function. You can invoke a function synchronously (and wait for the response), or asynchronously. To invoke a function asynchronously, set `InvocationType` to `Event`.

For [synchronous invocation](#), details about the function response, including errors, are included in the response body and headers. For either invocation type, you can find more information in the [execution log](#) and [trace](#).

When an error occurs, your function may be invoked multiple times. Retry behavior varies by error type, client, event source, and invocation type. For example, if you invoke a function asynchronously and it returns an error, Lambda executes the function up to two more times. For more information, see [Error handling and automatic retries in Lambda](#).

For [asynchronous invocation](#), Lambda adds events to a queue before sending them to your function. If your function does not have enough capacity to keep up with the queue, events may be lost. Occasionally, your function may receive the same event multiple times, even if no error occurs. To retain events that were not processed, configure your function with a [dead-letter queue](#).

The status code in the API response doesn't reflect function errors. Error codes are reserved for errors that prevent your function from executing, such as permissions errors, [quota](#) errors, or issues with your function's code and configuration. For example, Lambda returns `TooManyRequestsException` if running the function would cause you to exceed a concurrency limit at either the account level (`ConcurrentInvocationLimitExceeded`) or function level (`ReservedFunctionConcurrentInvocationLimitExceeded`).

For functions with a long timeout, your client might disconnect during synchronous invocation while it waits for a response. Configure your HTTP client, SDK, firewall, proxy, or operating system to allow for long connections with timeout or keep-alive settings.

This operation requires permission for the [lambda:InvokeFunction](#) action. For details on how to set up permissions for cross-account invocations, see [Granting function access to other accounts](#).

Request Syntax

```
POST /2015-03-31/functions/FunctionName/invocations?Qualifier=Qualifier HTTP/1.1
X-Amz-Invocation-Type: InvocationType
X-Amz-Log-Type: LogType
X-Amz-Client-Context: ClientContext

Payload
```

URI Request Parameters

The request uses the following URI parameters.

[ClientContext \(p. 1260\)](#)

Up to 3,583 bytes of base64-encoded data about the invoking client to pass to the function in the context object.

[FunctionName \(p. 1260\)](#)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** – my-function (name-only), my-function:v1 (with alias).

- **Function ARN** – `arn:aws:lambda:us-west-2:123456789012:function:my-function`.
- **Partial ARN** – `123456789012:function:my-function`.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: Yes

[InvocationType \(p. 1260\)](#)

Choose from the following options.

- **RequestResponse** (default) – Invoke the function synchronously. Keep the connection open until the function returns a response or times out. The API response includes the function response and additional data.
- **Event** – Invoke the function asynchronously. Send events that fail multiple times to the function's dead-letter queue (if one is configured). The API response only includes a status code.
- **DryRun** – Validate parameter values and verify that the user or role has permission to invoke the function.

Valid Values: Event | RequestResponse | DryRun

[LogType \(p. 1260\)](#)

Set to Tail to include the execution log in the response. Applies to synchronously invoked functions only.

Valid Values: None | Tail

[Qualifier \(p. 1260\)](#)

Specify a version or alias to invoke a published version of the function.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(|[a-zA-Z0-9$-_]+)`

Request Body

The request accepts the following binary data.

[Payload \(p. 1260\)](#)

The JSON that you want to provide to your Lambda function as input.

You can enter the JSON directly. For example, `--payload '{ "key": "value" }'`. You can also specify a file path. For example, `--payload file://payload.json`.

Response Syntax

```
HTTP/1.1 $Status$Code
X-Amz-Function-Error: $FunctionError
X-Amz-Log-Result: $LogResult
X-Amz-Executed-Version: $ExecutedVersion
```

Payload

Response Elements

If the action is successful, the service sends back the following HTTP response.

[StatusCode \(p. 1261\)](#)

The HTTP status code is in the 200 range for a successful request. For the RequestResponse invocation type, this status code is 200. For the Event invocation type, this status code is 202. For the DryRun invocation type, the status code is 204.

The response returns the following HTTP headers.

[ExecutedVersion \(p. 1261\)](#)

The version of the function that executed. When you invoke a function with an alias, this indicates which version the alias resolved to.

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST | [0-9]+)

[FunctionError \(p. 1261\)](#)

If present, indicates that an error occurred during function execution. Details about the error are included in the response payload.

[LogResult \(p. 1261\)](#)

The last 4 KB of the execution log, which is base64-encoded.

The response returns the following as the HTTP body.

[Payload \(p. 1261\)](#)

The response from the function, or an error object.

Errors

EC2AccessDeniedException

Need additional permissions to configure VPC settings.

HTTP Status Code: 502

EC2ThrottledException

Amazon EC2 throttled AWS Lambda during Lambda function initialization using the execution role provided for the function.

HTTP Status Code: 502

EC2UnexpectedException

AWS Lambda received an unexpected Amazon EC2 client exception while setting up for the Lambda function.

HTTP Status Code: 502

EFSIOException

An error occurred when reading from or writing to a connected file system.

HTTP Status Code: 410

EFSMountConnectivityException

The Lambda function couldn't make a network connection to the configured file system.

HTTP Status Code: 408

EFSMountFailureException

The Lambda function couldn't mount the configured file system due to a permission or configuration issue.

HTTP Status Code: 403

EFSMountTimeoutException

The Lambda function made a network connection to the configured file system, but the mount operation timed out.

HTTP Status Code: 408

ENILimitReachedException

AWS Lambda couldn't create an elastic network interface in the VPC, specified as part of Lambda function configuration, because the limit for network interfaces has been reached. For more information, see [Lambda quotas](#).

HTTP Status Code: 502

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

InvalidRequestContentException

The request body could not be parsed as JSON.

HTTP Status Code: 400

InvalidRuntimeException

The runtime or runtime version specified is not supported.

HTTP Status Code: 502

InvalidSecurityGroupIDException

The security group ID provided in the Lambda function VPC configuration is not valid.

HTTP Status Code: 502

InvalidSubnetIDException

The subnet ID provided in the Lambda function VPC configuration is not valid.

HTTP Status Code: 502

InvalidZipFileException

AWS Lambda could not unzip the deployment package.

HTTP Status Code: 502

KMSAccessDeniedException

Lambda couldn't decrypt the environment variables because AWS KMS access was denied. Check the Lambda function's KMS permissions.

HTTP Status Code: 502

KMSDisabledException

Lambda couldn't decrypt the environment variables because the AWS KMS key used is disabled. Check the Lambda function's KMS key settings.

HTTP Status Code: 502

KMSInvalidStateException

Lambda couldn't decrypt the environment variables because the state of the AWS KMS key used is not valid for Decrypt. Check the function's KMS key settings.

HTTP Status Code: 502

KMSNotFoundException

Lambda couldn't decrypt the environment variables because the AWS KMS key was not found. Check the function's KMS key settings.

HTTP Status Code: 502

RequestTooLargeException

The request payload exceeded the Invoke request body JSON input quota. For more information, see [Lambda quotas](#).

HTTP Status Code: 413

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ResourceNotReadyException

The function is inactive and its VPC connection is no longer available. Wait for the VPC connection to reestablish and try again.

HTTP Status Code: 502

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

SnapStartException

The `afterRestore()` [runtime hook](#) encountered an error. For more information, check the Amazon CloudWatch logs.

HTTP Status Code: 400

SnapStartNotReadyException

Lambda is initializing your function. You can invoke the function when the [function state](#) becomes Active.

HTTP Status Code: 409

SnapStartTimeoutException

Lambda couldn't restore the snapshot within the timeout limit.

HTTP Status Code: 408

SubnetIPAddressLimitReachedException

AWS Lambda couldn't set up VPC access for the Lambda function because one or more configured subnets has no available IP addresses.

HTTP Status Code: 502

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

UnsupportedMediaTypeException

The content type of the Invoke request body is not JSON.

HTTP Status Code: 415

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

InvokeAsync

This action has been deprecated.

Important

For asynchronous function invocation, use [Invoke \(p. 1260\)](#).

Invokes a function asynchronously.

Request Syntax

```
POST /2014-11-13/functions/FunctionName/invoke-async/ HTTP/1.1
```

InvokeArgs

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1266\)](#)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

Request Body

The request accepts the following binary data.

[InvokeArgs \(p. 1266\)](#)

The JSON that you want to provide to your Lambda function as input.

Required: Yes

Response Syntax

```
HTTP/1.1 Status
```

Response Elements

If the action is successful, the service sends back the following HTTP response.

[Status \(p. 1266\)](#)

The status code.

Errors

InvalidRequestContentException

The request body could not be parsed as JSON.

HTTP Status Code: 400

InvalidRuntimeException

The runtime or runtime version specified is not supported.

HTTP Status Code: 502

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

InvokeWithResponseStream

Configure your Lambda functions to stream response payloads back to clients. For more information, see [Configuring a Lambda function to stream responses](#).

This operation requires permission for the [lambda:InvokeFunction](#) action. For details on how to set up permissions for cross-account invocations, see [Granting function access to other accounts](#).

Request Syntax

```
POST /2021-11-15/functions/FunctionName/response-streaming-invocations?Qualifier=Qualifier
HTTP/1.1
X-Amz-Invocation-Type: InvocationType
X-Amz-Log-Type: LogType
X-Amz-Client-Context: ClientContext

Payload
```

URI Request Parameters

The request uses the following URI parameters.

[ClientContext \(p. 1268\)](#)

Up to 3,583 bytes of base64-encoded data about the invoking client to pass to the function in the context object.

[FunctionName \(p. 1268\)](#)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[InvocationType \(p. 1268\)](#)

Use one of the following options:

- **RequestResponse** (default) – Invoke the function synchronously. Keep the connection open until the function returns a response or times out. The API operation response includes the function response and additional data.
- **DryRun** – Validate parameter values and verify that the IAM user or role has permission to invoke the function.

Valid Values: RequestResponse | DryRun

[LogType \(p. 1268\)](#)

Set to Tail to include the execution log in the response. Applies to synchronously invoked functions only.

Valid Values: None | Tail

[Qualifier \(p. 1268\)](#)

The alias name.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$_-]+)

Request Body

The request accepts the following binary data.

[Payload \(p. 1268\)](#)

The JSON that you want to provide to your Lambda function as input.

You can enter the JSON directly. For example, --payload '{ "key": "value" }'. You can also specify a file path. For example, --payload file://payload.json.

Response Syntax

```
HTTP/1.1 StatusCode
X-Amz-Executed-Version: ExecutedVersion
Content-Type: ResponseStreamContentType
Content-type: application/json

{
  "EventStreamInvokeCompletePayloadChunkPayloadblob
    }
  }
}
```

Response Elements

If the action is successful, the service sends back the following HTTP response.

[StatusCode \(p. 1269\)](#)

For a successful request, the HTTP status code is in the 200 range. For the RequestResponse invocation type, this status code is 200. For the DryRun invocation type, this status code is 204.

The response returns the following HTTP headers.

[ExecutedVersion \(p. 1269\)](#)

The version of the function that executed. When you invoke a function with an alias, this indicates which version the alias resolved to.

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST | [0-9]+)

[ResponseStreamContentType \(p. 1269\)](#)

The type of data the stream is returning.

The following data is returned in JSON format by the service.

[EventStream \(p. 1269\)](#)

The stream of response payloads.

Type: [InvokeWithResponseStreamResponseEvent \(p. 1445\)](#) object

Errors

EC2AccessDeniedException

Need additional permissions to configure VPC settings.

HTTP Status Code: 502

EC2ThrottledException

Amazon EC2 throttled AWS Lambda during Lambda function initialization using the execution role provided for the function.

HTTP Status Code: 502

EC2UnexpectedException

AWS Lambda received an unexpected Amazon EC2 client exception while setting up for the Lambda function.

HTTP Status Code: 502

EFSIOException

An error occurred when reading from or writing to a connected file system.

HTTP Status Code: 410

EFSMountConnectivityException

The Lambda function couldn't make a network connection to the configured file system.

HTTP Status Code: 408

EFSMountFailureException

The Lambda function couldn't mount the configured file system due to a permission or configuration issue.

HTTP Status Code: 403

EFSMountTimeoutException

The Lambda function made a network connection to the configured file system, but the mount operation timed out.

HTTP Status Code: 408

ENILimitReachedException

AWS Lambda couldn't create an elastic network interface in the VPC, specified as part of Lambda function configuration, because the limit for network interfaces has been reached. For more information, see [Lambda quotas](#).

HTTP Status Code: 502

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

InvalidRequestContentException

The request body could not be parsed as JSON.

HTTP Status Code: 400

InvalidRuntimeException

The runtime or runtime version specified is not supported.

HTTP Status Code: 502

InvalidSecurityGroupIDException

The security group ID provided in the Lambda function VPC configuration is not valid.

HTTP Status Code: 502

InvalidSubnetIDException

The subnet ID provided in the Lambda function VPC configuration is not valid.

HTTP Status Code: 502

InvalidZipFileException

AWS Lambda could not unzip the deployment package.

HTTP Status Code: 502

KMSAccessDeniedException

Lambda couldn't decrypt the environment variables because AWS KMS access was denied. Check the Lambda function's KMS permissions.

HTTP Status Code: 502

KMSDisabledException

Lambda couldn't decrypt the environment variables because the AWS KMS key used is disabled. Check the Lambda function's KMS key settings.

HTTP Status Code: 502

KMSInvalidStateException

Lambda couldn't decrypt the environment variables because the state of the AWS KMS key used is not valid for Decrypt. Check the function's KMS key settings.

HTTP Status Code: 502

KMSNotFoundException

Lambda couldn't decrypt the environment variables because the AWS KMS key was not found. Check the function's KMS key settings.

HTTP Status Code: 502

RequestTooLargeException

The request payload exceeded the Invoke request body JSON input quota. For more information, see [Lambda quotas](#).

HTTP Status Code: 413

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ResourceNotReadyException

The function is inactive and its VPC connection is no longer available. Wait for the VPC connection to reestablish and try again.

HTTP Status Code: 502

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

SnapStartException

The `afterRestore()` [runtime hook](#) encountered an error. For more information, check the Amazon CloudWatch logs.

HTTP Status Code: 400

SnapStartNotReadyException

Lambda is initializing your function. You can invoke the function when the [function state](#) becomes Active.

HTTP Status Code: 409

SnapStartTimeoutException

Lambda couldn't restore the snapshot within the timeout limit.

HTTP Status Code: 408

SubnetIPAddressLimitReachedException

AWS Lambda couldn't set up VPC access for the Lambda function because one or more configured subnets has no available IP addresses.

HTTP Status Code: 502

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

UnsupportedMediaTypeException

The content type of the Invoke request body is not JSON.

HTTP Status Code: 415

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListAliases

Returns a list of [aliases](#) for a Lambda function.

Request Syntax

```
GET /2015-03-31/functions/FunctionName/aliases?  
FunctionVersion=FunctionVersion&Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1274\)](#)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[FunctionVersion \(p. 1274\)](#)

Specify a function version to only list aliases that invoke that version.

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST|[0-9]+)

[Marker \(p. 1274\)](#)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MaxItems \(p. 1274\)](#)

Limit the number of aliases returned.

Valid Range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
```

```
Content-type: application/json

{
  "Aliases": [
    {
      "AliasArn": "string",
      "Description": "string",
      "FunctionVersion": "string",
      "Name": "string",
      "RevisionId": "string",
      "RoutingConfig": {
        "AdditionalVersionWeights": {
          "string" : number
        }
      }
    }
  ],
  "NextMarker": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Aliases \(p. 1274\)](#)

A list of aliases.

Type: Array of [AliasConfiguration \(p. 1401\)](#) objects

[NextMarker \(p. 1274\)](#)

The pagination token that's included if more results are available.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListCodeSigningConfigs

Returns a list of [code signing configurations](#). A request returns up to 10,000 configurations per call. You can use the `MaxItems` parameter to return fewer configurations per call.

Request Syntax

```
GET /2020-04-22/code-signing-configs/?Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[Marker \(p. 1277\)](#)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MaxItems \(p. 1277\)](#)

Maximum number of items to return.

Valid Range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "CodeSigningConfigs": [
        {
            "AllowedPublishers": {
                "SigningProfileVersionArns": [ "string" ]
            },
            "CodeSigningConfigArn": "string",
            "CodeSigningConfigId": "string",
            "CodeSigningPolicies": {
                "UntrustedArtifactOnDeployment": "string"
            },
            "Description": "string",
            "LastModified": "string"
        }
    ],
    "NextMarker": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[CodeSigningConfigs \(p. 1277\)](#)

The code signing configurations

Type: Array of [CodeSigningConfig \(p. 1406\)](#) objects

[NextMarker \(p. 1277\)](#)

The pagination token that's included if more results are available.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListEventSourceMappings

Lists event source mappings. Specify an EventSourceArn to show only event source mappings for a single event source.

Request Syntax

```
GET /2015-03-31/event-source-mappings/?  
EventSourceArn=EventSourceArn&FunctionName=FunctionName&Marker=Marker&MaxItems=MaxItems  
HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[EventSourceArn \(p. 1279\)](#)

The Amazon Resource Name (ARN) of the event source.

- **Amazon Kinesis** – The ARN of the data stream or a stream consumer.
- **Amazon DynamoDB Streams** – The ARN of the stream.
- **Amazon Simple Queue Service** – The ARN of the queue.
- **Amazon Managed Streaming for Apache Kafka** – The ARN of the cluster.
- **Amazon MQ** – The ARN of the broker.
- **Amazon DocumentDB** – The ARN of the DocumentDB change stream.

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-.])+([a-z]{2}(-gov)?-[a-z]+\-\d{1})?:(\d{12})?:(.*.)`

[FunctionName \(p. 1279\)](#)

The name of the Lambda function.

Name formats

- **Function name** – MyFunction.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Version or Alias ARN** – arn:aws:lambda:us-west-2:123456789012:function:MyFunction:PROD.
- **Partial ARN** – 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it's limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[Marker \(p. 1279\)](#)

A pagination token returned by a previous call.

[MaxItems \(p. 1279\)](#)

The maximum number of event source mappings to return. Note that ListEventSourceMappings returns a maximum of 100 items in each response, even if you set the number higher.

Valid Range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "EventSourceMappings": [
        {
            "AmazonManagedKafkaEventSourceConfig": {
                "ConsumerGroupId": "string"
            },
            "BatchSize": number,
            "BisectBatchOnFunctionError": boolean,
            "DestinationConfig": {
                "OnFailure": {
                    "Destination": "string"
                },
                "OnSuccess": {
                    "Destination": "string"
                }
            },
            "DocumentDBEventSourceConfig": {
                "CollectionName": "string",
                "DatabaseName": "string",
                "FullDocument": "string"
            },
            "EventSourceArn": "string",
            "FilterCriteria": {
                "Filters": [
                    {
                        "Pattern": "string"
                    }
                ],
                "FunctionArn": "string",
                "FunctionResponseTypes": [ "string" ],
                "LastModified": number,
                "LastProcessingResult": "string",
                "MaximumBatchingWindowInSeconds": number,
                "MaximumRecordAgeInSeconds": number,
                "MaximumRetryAttempts": number,
                "ParallelizationFactor": number,
                "Queues": [ "string" ],
                "ScalingConfig": {
                    "MaximumConcurrency": number
                },
                "SelfManagedEventSource": {
                    "Endpoints": {
                        "string" : [ "string" ]
                    }
                },
                "SelfManagedKafkaEventSourceConfig": {
                    "ConsumerGroupId": "string"
                },
                "SourceAccessConfigurations": [
                    {
                        "Type": "string",
                        "URI": "string"
                    }
                ],
            }
        }
    ]
}
```

```
    "StartingPosition": "string",
    "StartingPositionTimestamp": number,
    "State": "string",
    "StateTransitionReason": "string",
    "Topics": [ "string" ],
    "TumblingWindowInSeconds": number,
    "UUID": "string"
}
],
"NextMarker": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[EventSourceMappings \(p. 1280\)](#)

A list of event source mappings.

Type: Array of [EventSourceMappingConfiguration \(p. 1419\)](#) objects

[NextMarker \(p. 1280\)](#)

A pagination token that's returned when the response doesn't contain all event source mappings.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)

- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListFunctionEventInvokeConfigs

Retrieves a list of configurations for asynchronous invocation for a function.

To configure options for asynchronous invocation, use [PutFunctionEventInvokeConfig \(p. 1330\)](#).

Request Syntax

```
GET /2019-09-25/functions/FunctionName/event-invoke-config/list?  
Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1283\)](#)

The name of the Lambda function.

Name formats

- **Function name** - my-function.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Marker \(p. 1283\)](#)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MaxItems \(p. 1283\)](#)

The maximum number of configurations to return.

Valid Range: Minimum value of 1. Maximum value of 50.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200  
Content-type: application/json  
{
```

```
"FunctionEventInvokeConfigs": [  
    {  
        "DestinationConfig": {  
            "OnFailure": {  
                "Destination": "string"  
            },  
            "OnSuccess": {  
                "Destination": "string"  
            }  
        },  
        "FunctionArn": "string",  
        "LastModified": number,  
        "MaximumEventAgeInSeconds": number,  
        "MaximumRetryAttempts": number  
    }  
,  
    "NextMarkerstring"  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[FunctionEventInvokeConfigs \(p. 1283\)](#)

A list of configurations.

Type: Array of [FunctionEventInvokeConfig \(p. 1436\)](#) objects

[NextMarker \(p. 1283\)](#)

The pagination token that's included if more results are available.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListFunctions

Returns a list of Lambda functions, with the version-specific configuration of each. Lambda returns up to 50 functions per call.

Set FunctionVersion to ALL to include all published versions of each function in addition to the unpublished version.

Note

The ListFunctions operation returns a subset of the [FunctionConfiguration \(p. 1430\)](#) fields. To get the additional fields (State, StateReasonCode, StateReason, LastUpdateStatus, LastUpdateStatusReason, LastUpdateStatusReasonCode, RuntimeVersionConfig) for a function or version, use [GetFunction \(p. 1220\)](#).

Request Syntax

```
GET /2015-03-31/functions/?  
FunctionVersion=FunctionVersion&Marker=Marker&MasterRegion=MasterRegion&MaxItems=MaxItems  
HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionVersion \(p. 1286\)](#)

Set to ALL to include entries for all published versions of each function.

Valid Values: ALL

[Marker \(p. 1286\)](#)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MasterRegion \(p. 1286\)](#)

For Lambda@Edge functions, the AWS Region of the master function. For example, us-east-1 filters the list of functions to include only Lambda@Edge functions replicated from a master function in US East (N. Virginia). If specified, you must set FunctionVersion to ALL.

Pattern: ALL | [a-z]{2}(-gov)?-[a-z]+-\d{1}

[MaxItems \(p. 1286\)](#)

The maximum number of functions to return in the response. Note that ListFunctions returns a maximum of 50 items in each response, even if you set the number higher.

Valid Range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
```

Content-type: application/json

```
{
  "Functions": [
    {
      "Architectures": [ "string" ],
      "CodeSha256": "string",
      "CodeSize": number,
      "DeadLetterConfig": {
        "TargetArn": "string"
      },
      "Description": "string",
      "Environment": {
        "Error": {
          "ErrorCode": "string",
          "Message": "string"
        },
        "Variables": {
          "string" : "string"
        }
      },
      "EphemeralStorage": {
        "Size": number
      },
      "FileSystemConfigs": [
        {
          "Arn": "string",
          "LocalMountPath": "string"
        }
      ],
      "FunctionArn": "string",
      "FunctionName": "string",
      "Handler": "string",
      "ImageConfigResponse": {
        "Error": {
          "ErrorCode": "string",
          "Message": "string"
        },
        "ImageConfig": {
          "Command": [ "string" ],
          "EntryPoint": [ "string" ],
          "WorkingDirectory": "string"
        }
      },
      "KMSKeyArn": "string",
      "LastModified": "string",
      "LastUpdateStatus": "string",
      "LastUpdateStatusReason": "string",
      "LastUpdateStatusReasonCode": "string",
      "Layers": [
        {
          "Arn": "string",
          "CodeSize": number,
          "SigningJobArn": "string",
          "SigningProfileVersionArn": "string"
        }
      ],
      "MasterArn": "string",
      "MemorySize": number,
      "PackageType": "string",
      "RevisionId": "string",
      "Role": "string",
      "Runtime": "string",
      "RuntimeVersionConfig": {
        "Error": {
          "ErrorCode": "string",
          "Message": "string"
        }
      }
    }
  ]
}
```

```
        "Message": "string"
    },
    "RuntimeVersionArn": "string"
},
"SigningJobArn": "string",
"SigningProfileVersionArn": "string",
"SnapStart": {
    "ApplyOn": "string",
    "OptimizationStatus": "string"
},
"State": "string",
"StateReason": "string",
"StateReasonCode": "string",
"Timeout": number,
"TracingConfig": {
    "Mode": "string"
},
"Version": "string",
"VpcConfig": {
    "SecurityGroupIds": [ "string" ],
    "SubnetIds": [ "string" ],
    "VpcId": "string"
}
}
],
"NextMarker": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Functions \(p. 1286\)](#)

A list of Lambda functions.

Type: Array of [FunctionConfiguration \(p. 1430\)](#) objects

[NextMarker \(p. 1286\)](#)

The pagination token that's included if more results are available.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListFunctionsByCodeSigningConfig

List the functions that use the specified code signing configuration. You can use this method prior to deleting a code signing configuration, to verify that no functions are using it.

Request Syntax

```
GET /2020-04-22/code-signing-configs/CodeSigningConfigArn/functions?  
Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[CodeSigningConfigArn \(p. 1290\)](#)

The The Amazon Resource Name (ARN) of the code signing configuration.

Length Constraints: Maximum length of 200.

Pattern: arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}((-gov)|(-iso(b?)))?-([a-z]+-\d{1}:\d{12}):code-signing-config:csc-[a-zA-Z0-9]{17}

Required: Yes

[Marker \(p. 1290\)](#)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MaxItems \(p. 1290\)](#)

Maximum number of items to return.

Valid Range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200  
Content-type: application/json  
  
{  
    "FunctionArns": [ "string" ],  
    "NextMarker": "string"  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[FunctionArns \(p. 1290\)](#)

The function ARNs.

Type: Array of strings

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?

[NextMarker \(p. 1290\)](#)

The pagination token that's included if more results are available.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListFunctionUrlConfigs

Returns a list of Lambda function URLs for the specified function.

Request Syntax

```
GET /2021-10-31/functions/FunctionName/urls?Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1292\)](#)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Marker \(p. 1292\)](#)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MaxItems \(p. 1292\)](#)

The maximum number of function URLs to return in the response. Note that `ListFunctionUrlConfigs` returns a maximum of 50 items in each response, even if you set the number higher.

Valid Range: Minimum value of 1. Maximum value of 50.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "FunctionUrlConfigs": [
```

```
{  
    "AuthType": "string",  
    "Cors": {  
        "AllowCredentials": boolean,  
        "AllowHeaders": [ "string" ],  
        "AllowMethods": [ "string" ],  
        "AllowOrigins": [ "string" ],  
        "ExposeHeaders": [ "string" ],  
        "MaxAge": number  
    },  
    "CreationTime": "string",  
    "FunctionArn": "string",  
    "FunctionUrl": "string",  
    "InvokeMode": "string",  
    "LastModifiedTime": "string"  
},  
]  
,  
"NextMarker": "string"  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[FunctionUrlConfigs \(p. 1292\)](#)

A list of function URL configurations.

Type: Array of [FunctionUrlConfig \(p. 1438\)](#) objects

[NextMarker \(p. 1292\)](#)

The pagination token that's included if more results are available.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListLayers

Lists [AWS Lambda layers](#) and shows information about the latest version of each. Specify a [runtime identifier](#) to list only layers that indicate that they're compatible with that runtime. Specify a compatible architecture to include only layers that are compatible with that [instruction set architecture](#).

Request Syntax

```
GET /2018-10-31/layers?  
CompatibleArchitecture=CompatibleArchitecture&CompatibleRuntime=CompatibleRuntime&Marker=Marker&MaxItems=MaxItems  
HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[CompatibleArchitecture \(p. 1295\)](#)

The compatible [instruction set architecture](#).

Valid Values: x86_64 | arm64

[CompatibleRuntime \(p. 1295\)](#)

A runtime identifier. For example, go1.x.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

[Marker \(p. 1295\)](#)

A pagination token returned by a previous call.

[MaxItems \(p. 1295\)](#)

The maximum number of layers to return.

Valid Range: Minimum value of 1. Maximum value of 50.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200  
Content-type: application/json  
  
{  
  "Layers": [  
    {
```

```
    "LatestMatchingVersion": {  
        "CompatibleArchitectures": [ "string" ],  
        "CompatibleRuntimes": [ "string" ],  
        "CreatedDate": "string",  
        "Description": "string",  
        "LayerVersionArn": "string",  
        "LicenseInfo": "string",  
        "Version": number  
    },  
    "LayerArn": "string",  
    "LayerName": "string"  
},  
]  
,  
"NextMarker": "string"  
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Layers \(p. 1295\)](#)

A list of function layers.

Type: Array of [LayersListItem \(p. 1447\)](#) objects

[NextMarker \(p. 1295\)](#)

A pagination token returned when the response doesn't contain all layers.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListLayerVersions

Lists the versions of an [AWS Lambda layer](#). Versions that have been deleted aren't listed. Specify a [runtime identifier](#) to list only versions that indicate that they're compatible with that runtime. Specify a compatible architecture to include only layer versions that are compatible with that architecture.

Request Syntax

```
GET /2018-10-31/layers/LayerName/versions?
```

```
CompatibleArchitecture=CompatibleArchitecture&CompatibleRuntime=CompatibleRuntime&Marker=Marker&MaxItems=MaxItems  
HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[CompatibleArchitecture \(p. 1298\)](#)

The compatible [instruction set architecture](#).

Valid Values: x86_64 | arm64

[CompatibleRuntime \(p. 1298\)](#)

A runtime identifier. For example, go1.x.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

[LayerName \(p. 1298\)](#)

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_+])|[a-zA-Z0-9-_+]

Required: Yes

[Marker \(p. 1298\)](#)

A pagination token returned by a previous call.

[MaxItems \(p. 1298\)](#)

The maximum number of versions to return.

Valid Range: Minimum value of 1. Maximum value of 50.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "LayerVersions": [
        {
            "CompatibleArchitecturesCompatibleRuntimes": [ "string" ],
            "CreatedDate": "string",
            "Description": "string",
            "LayerVersionArn": "string",
            "LicenseInfo": "string",
            "Version": number
        }
    ],
    "NextMarker": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

LayerVersions (p. 1299)

A list of versions.

Type: Array of [LayerVersionsListItem \(p. 1450\)](#) objects

NextMarker (p. 1299)

A pagination token returned when the response doesn't contain all versions.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListProvisionedConcurrencyConfigs

Retrieves a list of provisioned concurrency configurations for a function.

Request Syntax

```
GET /2019-09-30/functions/FunctionName/provisioned-concurrency?  
List=ALL&Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1301\)](#)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Marker \(p. 1301\)](#)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MaxItems \(p. 1301\)](#)

Specify a number to limit the number of configurations returned.

Valid Range: Minimum value of 1. Maximum value of 50.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200  
Content-type: application/json  
  
{  
  "NextMarker": "string",  
  "ProvisionedConcurrencyConfigs": [  
    {
```

```
    "AllocatedProvisionedConcurrentExecutions": number,
    "AvailableProvisionedConcurrentExecutions": number,
    "FunctionArn": "string",
    "LastModified": "string",
    "RequestedProvisionedConcurrentExecutions": number,
    "Status": "string",
    "StatusReason": "string"
}
]
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[NextMarker \(p. 1301\)](#)

The pagination token that's included if more results are available.

Type: String

[ProvisionedConcurrencyConfigs \(p. 1301\)](#)

A list of provisioned concurrency configurations.

Type: Array of [ProvisionedConcurrencyConfigListItem \(p. 1454\)](#) objects

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListTags

Returns a function's [tags](#). You can also view tags with [GetFunction \(p. 1220\)](#).

Request Syntax

```
GET /2017-03-31/tags/ARN HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[ARN \(p. 1304\)](#)

The function's Amazon Resource Name (ARN). Note: Lambda does not support adding tags to aliases or versions.

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$\$LATEST|[a-zA-Z0-9-_]+))?`

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "Tags": [
    {
      "string": "string"
    }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Tags \(p. 1304\)](#)

The function's tags.

Type: String to string map

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

ListVersionsByFunction

Returns a list of [versions](#), with the version-specific configuration of each. Lambda returns up to 50 versions per call.

Request Syntax

```
GET /2015-03-31/functions/FunctionName/versions?Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1306\)](#)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\\$\LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Marker \(p. 1306\)](#)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MaxItems \(p. 1306\)](#)

The maximum number of versions to return. Note that `ListVersionsByFunction` returns a maximum of 50 items in each response, even if you set the number higher.

Valid Range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "NextMarker": "string",
  "Versions": [
```

```
{
    "Architectures": [ "string" ],
    "CodeSha256": "string",
    "CodeSize": number,
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
    "Description": "string",
    "Environment": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "Variables": {
            "string" : "string"
        }
    },
    "EphemeralStorage": {
        "Size": number
    },
    "FileSystemConfigs": [
        {
            "Arn": "string",
            "LocalMountPath": "string"
        }
    ],
    "FunctionArn": "string",
    "FunctionName": "string",
    "Handler": "string",
    "ImageConfigResponse": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "ImageConfig": {
            "Command": [ "string" ],
            "EntryPoint": [ "string" ],
            "WorkingDirectory": "string"
        }
    },
    "KMSKeyArn": "string",
    "LastModified": "string",
    "LastUpdateStatus": "string",
    "LastUpdateStatusReason": "string",
    "LastUpdateStatusReasonCode": "string",
    "Layers": [
        {
            "Arn": "string",
            "CodeSize": number,
            "SigningJobArn": "string",
            "SigningProfileVersionArn": "string"
        }
    ],
    "MasterArn": "string",
    "MemorySize": number,
    "PackageType": "string",
    "RevisionId": "string",
    "Role": "string",
    "Runtime": "string",
    "RuntimeVersionConfig": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "RuntimeVersionArn": "string"
    },
}
```

```
"SigningJobArn": "string",
"SigningProfileVersionArn": "string",
"SnapStart": {
    "ApplyOn": "string",
    "OptimizationStatus": "string"
},
"State": "string",
"StateReason": "string",
"StateReasonCode": "string",
"Timeout": number,
"TracingConfig": {
    "Mode": "string"
},
"Version": "string",
"VpcConfig": {
    "SecurityGroupIds": [ "string" ],
    "SubnetIds": [ "string" ],
    "VpcId": "string"
}
}
]
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

NextMarker (p. 1306)

The pagination token that's included if more results are available.

Type: String

Versions (p. 1306)

A list of Lambda function versions.

Type: Array of [FunctionConfiguration](#) (p. 1430) objects

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

PublishLayerVersion

Creates an [AWS Lambda layer](#) from a ZIP archive. Each time you call PublishLayerVersion with the same layer name, a new version is created.

Add layers to your function with [CreateFunction \(p. 1165\)](#) or [UpdateFunctionConfiguration \(p. 1377\)](#).

Request Syntax

```
POST /2018-10-31/layers/LayerName/versions HTTP/1.1
Content-type: application/json

{
    "CompatibleArchitecturesCompatibleRuntimesContentS3BucketS3KeyS3ObjectVersionZipFileDescriptionLicenseInfo
```

URI Request Parameters

The request uses the following URI parameters.

[LayerName \(p. 1310\)](#)

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (`arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_+]|[a-zA-Z0-9-_]+`)

Required: Yes

Request Body

The request accepts the following data in JSON format.

[CompatibleArchitectures \(p. 1310\)](#)

A list of compatible [instruction set architectures](#).

Type: Array of strings

Array Members: Maximum number of 2 items.

Valid Values: x86_64 | arm64

Required: No

[CompatibleRuntimes \(p. 1310\)](#)

A list of compatible [function runtimes](#). Used for filtering with [ListLayers \(p. 1295\)](#) and [ListLayerVersions \(p. 1298\)](#).

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: Array of strings

Array Members: Maximum number of 15 items.

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

Required: No

[Content \(p. 1310\)](#)

The function layer archive.

Type: [LayerVersionContentInput \(p. 1448\)](#) object

Required: Yes

[Description \(p. 1310\)](#)

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

[LicenseInfo \(p. 1310\)](#)

The layer's software license. It can be any of the following:

- An [SPDX license identifier](#). For example, MIT.
- The URL of a license hosted on the internet. For example, <https://opensource.org/licenses/MIT>.
- The full text of the license.

Type: String

Length Constraints: Maximum length of 512.

Required: No

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
    "CompatibleArchitectures": [ "string" ],
    "CompatibleRuntimes": [ "string" ],
    "Content": {
        "CodeSha256": "string",
        "CodeSize": number,
        "Location": "string",
        "SigningJobArn": "string",
    }
}
```

```
        "SigningProfileVersionArn": "string"
    },
    "CreatedDate": "string",
    "Description": "string",
    "LayerArn": "string",
    "LayerVersionArn": "string",
    "LicenseInfo": "string",
    "Version": number
}
```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

[CompatibleArchitectures \(p. 1311\)](#)

A list of compatible [instruction set architectures](#).

Type: Array of strings

Array Members: Maximum number of 2 items.

Valid Values: x86_64 | arm64

[CompatibleRuntimes \(p. 1311\)](#)

The layer's compatible runtimes.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: Array of strings

Array Members: Maximum number of 15 items.

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

[Content \(p. 1311\)](#)

Details about the layer version.

Type: [LayerVersionContentOutput \(p. 1449\)](#) object

[CreatedDate \(p. 1311\)](#)

The date that the layer version was created, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[Description \(p. 1311\)](#)

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[LayerArn \(p. 1311\)](#)

The ARN of the layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+

[LayerVersionArn \(p. 1311\)](#)

The ARN of the layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+

[LicenseInfo \(p. 1311\)](#)

The layer's software license.

Type: String

Length Constraints: Maximum length of 512.

[Version \(p. 1311\)](#)

The version number.

Type: Long

Errors

CodeStorageExceededException

Your AWS account has exceeded its maximum total code size. For more information, see [Lambda quotas](#).

HTTP Status Code: 400

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

PublishVersion

Creates a [version](#) from the current code and configuration of a function. Use versions to create a snapshot of your function code and configuration that doesn't change.

AWS Lambda doesn't publish a version if the function's configuration and code haven't changed since the last version. Use [UpdateFunctionCode \(p. 1367\)](#) or [UpdateFunctionConfiguration \(p. 1377\)](#) to update the function before publishing a version.

Clients can invoke versions directly or with an alias. To create an alias, use [CreateAlias \(p. 1146\)](#).

Request Syntax

```
POST /2015-03-31/functions/FunctionName/versions HTTP/1.1
Content-type: application/json

{
  "CodeSha256": "string",
  "Description": "string",
  "RevisionId": "string"
}
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1315\)](#)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

Request Body

The request accepts the following data in JSON format.

[CodeSha256 \(p. 1315\)](#)

Only publish a version if the hash value matches the value that's specified. Use this option to avoid publishing a version if the function code has changed since you last updated it. You can get the hash for the version that you uploaded from the output of [UpdateFunctionCode \(p. 1367\)](#).

Type: String

Required: No

[Description \(p. 1315\)](#)

A description for the version to override the description in the function configuration.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

[RevisionId \(p. 1315\)](#)

Only update the function if the revision ID matches the ID that's specified. Use this option to avoid publishing a version if the function configuration has changed since you last updated it.

Type: String

Required: No

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
    "Architectures": [ "string" ],
    "CodeSha256": "string",
    "CodeSizeDeadLetterConfig": {
        "TargetArn": "string"
    },
    "Description": "string",
    "Environment": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "Variables": {
            "string" : "string"
        }
    },
    "EphemeralStorage": {
        "Size": number
    },
    "FileSystemConfigs": [
        {
            "Arn": "string",
            "LocalMountPath": "string"
        }
    ],
    "FunctionArn": "string",
    "FunctionName": "string",
    "Handler": "string",
    "ImageConfigResponse": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "ImageConfig": {
            "Command": [ "string" ],
            "EntryPoint": [ "string" ],
            ...
        }
    }
}
```

```

        "WorkingDirectory": "string"
    },
    "KMSKeyArn": "string",
    "LastModified": "string",
    "LastUpdateStatus": "string",
    "LastUpdateStatusReason": "string",
    "LastUpdateStatusReasonCode": "string",
    "Layers": [
        {
            "Arn": "string",
            "CodeSize": number,
            "SigningJobArn": "string",
            "SigningProfileVersionArn": "string"
        }
    ],
    "MasterArn": "string",
    "MemorySize": number,
    "PackageType": "string",
    "RevisionId": "string",
    "Role": "string",
    "Runtime": "string",
    "RuntimeVersionConfig": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "RuntimeVersionArn": "string"
    },
    "SigningJobArn": "string",
    "SigningProfileVersionArn": "string",
    "SnapStart": {
        "ApplyOn": "string",
        "OptimizationStatus": "string"
    },
    "State": "string",
    "StateReason": "string",
    "StateReasonCode": "string",
    "Timeout": number,
    "TracingConfig": {
        "Mode": "string"
    },
    "Version": "string",
    "VpcConfig": {
        "SecurityGroupIds": [ "string" ],
        "SubnetIds": [ "string" ],
        "VpcId": "string"
    }
}

```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

[Architectures \(p. 1316\)](#)

The instruction set architecture that the function supports. Architecture is a string array with one of the valid values. The default architecture value is x86_64.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: x86_64 | arm64

CodeSha256 (p. 1316)

The SHA256 hash of the function's deployment package.

Type: String

CodeSize (p. 1316)

The size of the function's deployment package, in bytes.

Type: Long

DeadLetterConfig (p. 1316)

The function's dead letter queue.

Type: [DeadLetterConfig \(p. 1412\)](#) object

Description (p. 1316)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Environment (p. 1316)

The function's [environment variables](#). Omitted from AWS CloudTrail logs.

Type: [EnvironmentResponse \(p. 1417\)](#) object

EphemeralStorage (p. 1316)

The size of the function's /tmp directory in MB. The default value is 512, but it can be any whole number between 512 and 10,240 MB.

Type: [EphemeralStorage \(p. 1418\)](#) object

FileSystemConfigs (p. 1316)

Connection settings for an [Amazon EFS file system](#).

Type: Array of [FileSystemConfig \(p. 1424\)](#) objects

Array Members: Maximum number of 1 item.

FunctionArn (p. 1316)

The function's Amazon Resource Name (ARN).

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:(\\$\\$LATEST|[a-zA-Z0-9-_]+))?

FunctionName (p. 1316)

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?\d{12}:(function:)?([a-zA-Z0-9-_\.]+)(:(\\$\\$LATEST|[a-zA-Z0-9-_]+))?

[Handler \(p. 1316\)](#)

The function that Lambda calls to begin running your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: [^\s]+

[ImageConfigResponse \(p. 1316\)](#)

The function's image configuration values.

Type: [ImageConfigResponse \(p. 1442\)](#) object

[KMSKeyArn \(p. 1316\)](#)

The AWS KMS key that's used to encrypt the function's [environment variables](#). When [Lambda SnapStart](#) is activated, this key is also used to encrypt the function's snapshot. This key is returned only if you've configured a customer managed key.

Type: String

Pattern: (arn:(aws[a-zA-Z-]*)?:[a-z0-9-.]+:[^.]+|()

[LastModified \(p. 1316\)](#)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[LastUpdateStatus \(p. 1316\)](#)

The status of the last update that was performed on the function. This is first set to Successful after function creation completes.

Type: String

Valid Values: Successful | Failed | InProgress

[LastUpdateStatusReason \(p. 1316\)](#)

The reason for the last update that was performed on the function.

Type: String

[LastUpdateStatusReasonCode \(p. 1316\)](#)

The reason code for the last update that was performed on the function.

Type: String

Valid Values: EniLimitExceeded | InsufficientRolePermissions | InvalidConfiguration | InternalError | SubnetOutOfRange | InvalidSubnet | InvalidSecurityGroup | ImageDeleted | ImageAccessDenied | InvalidImage | KMSKeyAccessDenied | KMSKeyNotFound | InvalidStateKMSKey | DisabledKMSKey | EFSIOError | EFSMountConnectivityError | EFSMountFailure | EFSMountTimeout | InvalidRuntime | InvalidZipFileException | FunctionError

[Layers \(p. 1316\)](#)

The function's [layers](#).

Type: Array of [Layer \(p. 1446\)](#) objects

[MasterArn \(p. 1316\)](#)

For Lambda@Edge functions, the ARN of the main function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$\$LATEST|[a-zA-Z0-9-_]+))?`

[MemorySize \(p. 1316\)](#)

The amount of memory available to the function at runtime.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 10240.

[PackageType \(p. 1316\)](#)

The type of deployment package. Set to Image for container image and set Zip for .zip file archive.

Type: String

Valid Values: Zip | Image

[RevisionId \(p. 1316\)](#)

The latest updated revision of the function or alias.

Type: String

[Role \(p. 1316\)](#)

The function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\\-_/.]+`

[Runtime \(p. 1316\)](#)

The identifier of the function's [runtime](#). Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

[RuntimeVersionConfig \(p. 1316\)](#)

The ARN of the runtime and any errors that occurred.

Type: [RuntimeVersionConfig \(p. 1456\)](#) object

[SigningJobArn \(p. 1316\)](#)

The ARN of the signing job.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+([a-z]{2}(-gov)?-[a-z]+-\d{1})?:(\d{12})?:(.*)`

[SigningProfileVersionArn \(p. 1316\)](#)

The ARN of the signing profile version.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+([a-z]{2}(-gov)?-[a-z]+-\d{1})?:(\d{12})?:(.*)`

[SnapStart \(p. 1316\)](#)

Set `ApplyOn` to `PublishedVersions` to create a snapshot of the initialized execution environment when you publish a function version. For more information, see [Improving startup performance with Lambda SnapStart](#).

Type: [SnapStartResponse \(p. 1462\)](#) object

[State \(p. 1316\)](#)

The current state of the function. When the state is `Inactive`, you can reactivate the function by invoking it.

Type: String

Valid Values: `Pending` | `Active` | `Inactive` | `Failed`

[StateReason \(p. 1316\)](#)

The reason for the function's current state.

Type: String

[StateReasonCode \(p. 1316\)](#)

The reason code for the function's current state. When the code is `Creating`, you can't invoke or modify the function.

Type: String

Valid Values: `Idle` | `Creating` | `Restoring` | `EniLimitExceeded` | `InsufficientRolePermissions` | `InvalidConfiguration` | `InternalError` | `SubnetOutofIPAddresses` | `InvalidSubnet` | `InvalidSecurityGroup` | `ImageDeleted` | `ImageAccessDenied` | `InvalidImage` | `KMSKeyAccessDenied` | `KMSKeyNotFound` | `InvalidStateKMSKey` | `DisabledKMSKey` | `EFSIOError` | `EFSMountConnectivityError` | `EFSMountFailure` | `EFSMountTimeout` | `InvalidRuntime` | `InvalidZipFileException` | `FunctionError`

[Timeout \(p. 1316\)](#)

The amount of time in seconds that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

[TracingConfig \(p. 1316\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 1466\)](#) object

[Version \(p. 1316\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST | [0-9]+)

[VpcConfig \(p. 1316\)](#)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 1468\)](#) object

Errors

CodeStorageExceededException****

Your AWS account has exceeded its maximum total code size. For more information, see [Lambda quotas](#).

HTTP Status Code: 400

InvalidParameterValueException****

One of the parameters in the request is not valid.

HTTP Status Code: 400

PreconditionFailedException****

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceConflictException****

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException****

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException****

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)

- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

PutFunctionCodeSigningConfig

Update the code signing configuration for the function. Changes to the code signing configuration take effect the next time a user tries to deploy a code package to the function.

Request Syntax

```
PUT /2020-06-30/functions/FunctionName/code-signing-config HTTP/1.1
Content-type: application/json

{
    "CodeSigningConfigArn": "string"
}
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName (p. 1324)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

Request Body

The request accepts the following data in JSON format.

CodeSigningConfigArn (p. 1324)

The The Amazon Resource Name (ARN) of the code signing configuration.

Type: String

Length Constraints: Maximum length of 200.

Pattern: arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}((-gov)|(-iso(b?)))?-+[a-z]+\d{1}:\d{12}:code-signing-config:csc-[a-zA-Z0-9]{17}

Required: Yes

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "CodeSigningConfigArn": "string",
    "FunctionName": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[CodeSigningConfigArn](#) (p. 1325)

The The Amazon Resource Name (ARN) of the code signing configuration.

Type: String

Length Constraints: Maximum length of 200.

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}((-gov)|(-iso(b?)))?- [a-z]+-\d{1}:\d{12}:code-signing-config:csc-[a-zA-Z0-9]{17}

[FunctionName](#) (p. 1325)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-zA-Z]{2}(-gov)?-[a-zA-Z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Errors

CodeSigningConfigNotFoundException

The specified code signing configuration does not exist.

HTTP Status Code: 404

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

PutFunctionConcurrency

Sets the maximum number of simultaneous executions for a function, and reserves capacity for that concurrency level.

Concurrency settings apply to the function as a whole, including all published versions and the unpublished version. Reserving concurrency both ensures that your function has capacity to process the specified number of events simultaneously, and prevents it from scaling beyond that level. Use [GetFunction \(p. 1220\)](#) to see the current setting for a function.

Use [GetAccountSettings \(p. 1207\)](#) to see your Regional concurrency limit. You can reserve concurrency for as many functions as you like, as long as you leave at least 100 simultaneous executions unreserved for functions that aren't configured with a per-function limit. For more information, see [Lambda function scaling](#).

Request Syntax

```
PUT /2017-10-31/functions/FunctionName/concurrency HTTP/1.1
Content-type: application/json

{
    "ReservedConcurrentExecutions": number
}
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1327\)](#)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

Request Body

The request accepts the following data in JSON format.

[ReservedConcurrentExecutions \(p. 1327\)](#)

The number of simultaneous executions to reserve for the function.

Type: Integer

Valid Range: Minimum value of 0.

Required: Yes

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "ReservedConcurrentExecutions": number
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

ReservedConcurrentExecutions (p. 1328)

The number of concurrent executions that are reserved for this function. For more information, see [Managing Lambda reserved concurrency](#).

Type: Integer

Valid Range: Minimum value of 0.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

PutFunctionEventInvokeConfig

Configures options for [asynchronous invocation](#) on a function, version, or alias. If a configuration already exists for a function, version, or alias, this operation overwrites it. If you exclude any settings, they are removed. To set one option without affecting existing settings for other options, use [UpdateFunctionEventInvokeConfig \(p. 1389\)](#).

By default, Lambda retries an asynchronous invocation twice if the function returns an error. It retains events in a queue for up to six hours. When an event fails all processing attempts or stays in the asynchronous invocation queue for too long, Lambda discards it. To retain discarded events, configure a dead-letter queue with [UpdateFunctionConfiguration \(p. 1377\)](#).

To send an invocation record to a queue, topic, function, or event bus, specify a [destination](#). You can configure separate destinations for successful invocations (on-success) and events that fail all processing attempts (on-failure). You can configure destinations in addition to or instead of a dead-letter queue.

Request Syntax

```
PUT /2019-09-25/functions/FunctionName/event-invoke-config?Qualifier=Qualifier HTTP/1.1
Content-type: application/json

{
  "DestinationConfigOnFailureDestinationOnSuccessDestinationMaximumEventAgeInSecondsnumber,
  "MaximumRetryAttemptsnumber
}
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1330\)](#)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** - my-function (name-only), my-function:v1 (with alias).
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_+]):(\\$\LATEST|[a-zA-Z0-9-_+]))?

Required: Yes

[Qualifier \(p. 1330\)](#)

A version number or alias name.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (| [a-zA-Z0-9\$_-]+)

Request Body

The request accepts the following data in JSON format.

[DestinationConfig \(p. 1330\)](#)

A destination for events after they have been sent to a function for processing.

Destinations

- **Function** - The Amazon Resource Name (ARN) of a Lambda function.
- **Queue** - The ARN of a standard SQS queue.
- **Topic** - The ARN of a standard SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: [DestinationConfig \(p. 1413\)](#) object

Required: No

[MaximumEventAgeInSeconds \(p. 1330\)](#)

The maximum age of a request that Lambda sends to a function for processing.

Type: Integer

Valid Range: Minimum value of 60. Maximum value of 21600.

Required: No

[MaximumRetryAttempts \(p. 1330\)](#)

The maximum number of times to retry when the function returns an error.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 2.

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "DestinationConfig": {
    "OnFailure": {
      "DestinationOnSuccess": {
      "Destination
```

```
        },
        "FunctionArn": "string",
        "LastModifiedMaximumEventAgeInSeconds": "number",
        "MaximumRetryAttempts": "number"
    }
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[DestinationConfig \(p. 1331\)](#)

A destination for events after they have been sent to a function for processing.

Destinations

- **Function** - The Amazon Resource Name (ARN) of a Lambda function.
- **Queue** - The ARN of a standard SQS queue.
- **Topic** - The ARN of a standard SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: [DestinationConfig \(p. 1413\)](#) object

[FunctionArn \(p. 1331\)](#)

The Amazon Resource Name (ARN) of the function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$\\$LATEST|[a-zA-Z0-9-_]+))?

[LastModified \(p. 1331\)](#)

The date and time that the configuration was last updated, in Unix time seconds.

Type: Timestamp

[MaximumEventAgeInSeconds \(p. 1331\)](#)

The maximum age of a request that Lambda sends to a function for processing.

Type: Integer

Valid Range: Minimum value of 60. Maximum value of 21600.

[MaximumRetryAttempts \(p. 1331\)](#)

The maximum number of times to retry when the function returns an error.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 2.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

PutProvisionedConcurrencyConfig

Adds a provisioned concurrency configuration to a function's alias or version.

Request Syntax

```
PUT /2019-09-30/functions/FunctionName/provisioned-concurrency?Qualifier=Qualifier HTTP/1.1
Content-type: application/json

{
    "ProvisionedConcurrentExecutions": number
}
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName (p. 1334)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

Qualifier (p. 1334)

The version number or alias name.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$-_]+)

Required: Yes

Request Body

The request accepts the following data in JSON format.

ProvisionedConcurrentExecutions (p. 1334)

The amount of provisioned concurrency to allocate for the version or alias.

Type: Integer

Valid Range: Minimum value of 1.

Required: Yes

Response Syntax

```
HTTP/1.1 202
Content-type: application/json

{
    "AllocatedProvisionedConcurrentExecutions": number,
    "AvailableProvisionedConcurrentExecutions": number,
    "LastModified": "string",
    "RequestedProvisionedConcurrentExecutions": number,
    "Status": "string",
    "StatusReason": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 202 response.

The following data is returned in JSON format by the service.

[AllocatedProvisionedConcurrentExecutions \(p. 1335\)](#)

The amount of provisioned concurrency allocated. When a weighted alias is used during linear and canary deployments, this value fluctuates depending on the amount of concurrency that is provisioned for the function versions.

Type: Integer

Valid Range: Minimum value of 0.

[AvailableProvisionedConcurrentExecutions \(p. 1335\)](#)

The amount of provisioned concurrency available.

Type: Integer

Valid Range: Minimum value of 0.

[LastModified \(p. 1335\)](#)

The date and time that a user last updated the configuration, in [ISO 8601 format](#).

Type: String

[RequestedProvisionedConcurrentExecutions \(p. 1335\)](#)

The amount of provisioned concurrency requested.

Type: Integer

Valid Range: Minimum value of 1.

[Status \(p. 1335\)](#)

The status of the allocation process.

Type: String

Valid Values: IN_PROGRESS | READY | FAILED

[StatusReason \(p. 1335\)](#)

For failed allocations, the reason that provisioned concurrency could not be allocated.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

PutRuntimeManagementConfig

Sets the runtime management configuration for a function's version. For more information, see [Runtime updates](#).

Request Syntax

```
PUT /2021-07-20/functions/FunctionName/runtime-management-config?Qualifier=Qualifier
HTTP/1.1
Content-type: application/json

{
  "RuntimeVersionArn": "string",
  "UpdateRuntimeOn": "string"
}
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1337\)](#)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1337\)](#)

Specify a version of the function. This can be \$LATEST or a published version number. If no value is specified, the configuration for the \$LATEST version is returned.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$-_]+)

Request Body

The request accepts the following data in JSON format.

[RuntimeVersionArn \(p. 1337\)](#)

The ARN of the runtime version you want the function to use.

Note

This is only required if you're using the **Manual** runtime update mode.

Type: String

Length Constraints: Minimum length of 26. Maximum length of 2048.

Pattern: ^arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}((-gov)|(-iso(b?)))?-([a-z]+-\d{1}):runtime:.+\$

Required: No

[UpdateRuntimeOn \(p. 1337\)](#)

Specify the runtime update mode.

- **Auto (default)** - Automatically update to the most recent and secure runtime version using a [Two-phase runtime version rollout](#). This is the best choice for most customers to ensure they always benefit from runtime updates.
- **Function update** - Lambda updates the runtime of your function to the most recent and secure runtime version when you update your function. This approach synchronizes runtime updates with function deployments, giving you control over when runtime updates are applied and allowing you to detect and mitigate rare runtime update incompatibilities early. When using this setting, you need to regularly update your functions to keep their runtime up-to-date.
- **Manual** - You specify a runtime version in your function configuration. The function will use this runtime version indefinitely. In the rare case where a new runtime version is incompatible with an existing function, this allows you to roll back your function to an earlier runtime version. For more information, see [Roll back a runtime version](#).

Type: String

Valid Values: Auto | Manual | FunctionUpdate

Required: Yes

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "FunctionArn": "string",
  "RuntimeVersionArn": "string",
  "UpdateRuntimeOn": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[FunctionArn \(p. 1338\)](#)

The ARN of the function

Type: String

Pattern: arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}((-gov)?-[a-z]+\d{1}:\d{12}):function:[a-zA-Z0-9-_]+(:(\\$LATEST|[a-zA-Z0-9-_]+))?

[RuntimeVersionArn \(p. 1338\)](#)

The ARN of the runtime the function is configured to use. If the runtime update mode is **manual**, the ARN is returned, otherwise null is returned.

Type: String

Length Constraints: Minimum length of 26. Maximum length of 2048.

Pattern: ^arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}((-gov)|(-iso(b?)))?-([a-z]+-\d{1}):runtime:.+\$

[UpdateRuntimeOn \(p. 1338\)](#)

The runtime update mode.

Type: String

Valid Values: Auto | Manual | FunctionUpdate

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)

- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

RemoveLayerVersionPermission

Removes a statement from the permissions policy for a version of an [AWS Lambda layer](#). For more information, see [AddLayerVersionPermission \(p. 1137\)](#).

Request Syntax

```
DELETE /2018-10-31/layers/LayerName/versions/VersionNumber/policy/StatementId?  
RevisionId=RevisionId HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[LayerName \(p. 1341\)](#)

The name or Amazon Resource Name (ARN) of the layer.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_+]) | [a-zA-Z0-9-_+]

Required: Yes

[RevisionId \(p. 1341\)](#)

Only update the policy if the revision ID matches the ID specified. Use this option to avoid modifying a policy that has changed since you last read it.

[StatementId \(p. 1341\)](#)

The identifier that was specified when the statement was added.

Length Constraints: Minimum length of 1. Maximum length of 100.

Pattern: ([a-zA-Z0-9-_]+)

Required: Yes

[VersionNumber \(p. 1341\)](#)

The version number.

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

PreconditionFailedException

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

RemovePermission

Revokes function-use permission from an AWS service or another AWS account. You can get the ID of the statement from the output of [GetPolicy \(p. 1252\)](#).

Request Syntax

```
DELETE /2015-03-31/functions/FunctionName/policy/StatementId?  
Qualifier=Qualifier&RevisionId=RevisionId HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1343\)](#)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** – my-function (name-only), my-function:v1 (with alias).
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1343\)](#)

Specify a version or alias to remove permissions from a published version of the function.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$-_]+)

[RevisionId \(p. 1343\)](#)

Update the policy only if the revision ID matches the ID that's specified. Use this option to avoid modifying a policy that has changed since you last read it.

[StatementId \(p. 1343\)](#)

Statement ID of the permission to remove.

Length Constraints: Minimum length of 1. Maximum length of 100.

Pattern: ([a-zA-Z0-9-_\.]+)

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

PreconditionFailedException

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

TagResource

Adds [tags](#) to a function.

Request Syntax

```
POST /2017-03-31/tags/ARN HTTP/1.1
Content-type: application/json

{
  "Tags": {
    "string" : "string"
  }
}
```

URI Request Parameters

The request uses the following URI parameters.

[ARN \(p. 1345\)](#)

The function's Amazon Resource Name (ARN).

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

Request Body

The request accepts the following data in JSON format.

[Tags \(p. 1345\)](#)

A list of tags to apply to the function.

Type: String to string map

Required: Yes

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

UntagResource

Removes [tags](#) from a function.

Request Syntax

```
DELETE /2017-03-31/tags/ARN?tagKeys=TagKeys HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[ARN \(p. 1347\)](#)

The function's Amazon Resource Name (ARN).

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_+]))?

Required: Yes

[TagKeys \(p. 1347\)](#)

A list of tag keys to remove from the function.

Required: Yes

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

UpdateAlias

Updates the configuration of a Lambda function [alias](#).

Request Syntax

```
PUT /2015-03-31/functions/FunctionName/aliases/Name HTTP/1.1
Content-type: application/json

{
  "Description": "string",
  "FunctionVersion": "string",
  "RevisionId": "string",
  "RoutingConfig": {
    "AdditionalVersionWeights": {
      "string" : number
    }
  }
}
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1349\)](#)

The name of the Lambda function.

Name formats

- **Function name** - MyFunction.
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Partial ARN** - 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Name \(p. 1349\)](#)

The name of the alias.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (?![0-9]+\$)([a-zA-Z0-9-_]+)

Required: Yes

Request Body

The request accepts the following data in JSON format.

[Description \(p. 1349\)](#)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

[FunctionVersion \(p. 1349\)](#)

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST | [0-9]+)

Required: No

[RevisionId \(p. 1349\)](#)

Only update the alias if the revision ID matches the ID that's specified. Use this option to avoid modifying an alias that has changed since you last read it.

Type: String

Required: No

[RoutingConfig \(p. 1349\)](#)

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 1403\)](#) object

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "AliasArn": "string",
    "Description": "string",
    "FunctionVersion": "string",
    "Name": "string",
    "RevisionId": "string",
    "RoutingConfig": {
        "AdditionalVersionWeights": {
            "string" : number
        }
    }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[AliasArn \(p. 1350\)](#)

The Amazon Resource Name (ARN) of the alias.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[Description \(p. 1350\)](#)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[FunctionVersion \(p. 1350\)](#)

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

[Name \(p. 1350\)](#)

The name of the alias.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(?!^[\d-]+)$)([a-zA-Z0-9-_]+)`

[RevisionId \(p. 1350\)](#)

A unique identifier that changes when you update the alias.

Type: String

[RoutingConfig \(p. 1350\)](#)

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 1403\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

PreconditionFailedException

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

UpdateCodeSigningConfig

Update the code signing configuration. Changes to the code signing configuration take effect the next time a user tries to deploy a code package to the function.

Request Syntax

```
PUT /2020-04-22/code-signing-configs/CodeSigningConfigArn HTTP/1.1
Content-type: application/json

{
  "AllowedPublishersSigningProfileVersionArnsCodeSigningPoliciesUntrustedArtifactOnDeploymentDescription
```

URI Request Parameters

The request uses the following URI parameters.

CodeSigningConfigArn (p. 1353)

The The Amazon Resource Name (ARN) of the code signing configuration.

Length Constraints: Maximum length of 200.

Pattern: arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}((-gov)|(-iso(b?)))?-+[a-z]+-\d{1}:\d{12}:code-signing-config:csc-[a-z0-9]{17}

Required: Yes

Request Body

The request accepts the following data in JSON format.

AllowedPublishers (p. 1353)

Signing profiles for this code signing configuration.

Type: [AllowedPublishers \(p. 1404\)](#) object

Required: No

CodeSigningPolicies (p. 1353)

The code signing policy.

Type: [CodeSigningPolicies \(p. 1408\)](#) object

Required: No

Description (p. 1353)

Descriptive name for this code signing configuration.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "CodeSigningConfig": {
        "AllowedPublishers": {
            "SigningProfileVersionArns": [ "string" ]
        },
        "CodeSigningConfigArn": "string",
        "CodeSigningConfigId": "string",
        "CodeSigningPolicies": {
            "UntrustedArtifactOnDeployment": "string"
        },
        "Description": "string",
        "LastModified": "string"
    }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[CodeSigningConfig \(p. 1354\)](#)

The code signing configuration

Type: [CodeSigningConfig \(p. 1406\)](#) object

Errors

InvalidArgumentException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

UpdateEventSourceMapping

Updates an event source mapping. You can change the function that AWS Lambda invokes, or pause invocation and resume later from the same location.

For details about how to configure different event sources, see the following topics.

- [Amazon DynamoDB Streams](#)
- [Amazon Kinesis](#)
- [Amazon SQS](#)
- [Amazon MQ and RabbitMQ](#)
- [Amazon MSK](#)
- [Apache Kafka](#)
- [Amazon DocumentDB](#)

The following error handling options are available only for stream sources (DynamoDB and Kinesis):

- **BisectBatchOnFunctionError** – If the function returns an error, split the batch in two and retry.
- **DestinationConfig** – Send discarded records to an Amazon SQS queue or Amazon SNS topic.
- **MaximumRecordAgeInSeconds** – Discard records older than the specified age. The default value is infinite (-1). When set to infinite (-1), failed records are retried until the record expires
- **MaximumRetryAttempts** – Discard records after the specified number of retries. The default value is infinite (-1). When set to infinite (-1), failed records are retried until the record expires.
- **ParallelizationFactor** – Process multiple batches from each shard concurrently.

For information about which configuration parameters apply to each event source, see the following topics.

- [Amazon DynamoDB Streams](#)
- [Amazon Kinesis](#)
- [Amazon SQS](#)
- [Amazon MQ and RabbitMQ](#)
- [Amazon MSK](#)
- [Apache Kafka](#)
- [Amazon DocumentDB](#)

Request Syntax

```
PUT /2015-03-31/event-source-mappings/UUID HTTP/1.1
Content-type: application/json
```

```
{
  "BatchSize": number,
  "BisectBatchOnFunctionError": boolean,
  "DestinationConfig": {
    "OnFailure": {
      "Destinationstring"
    },
    "OnSuccess": {
      "Destinationstring"
    }
  },
}
```

```
"DocumentDBEventSourceConfig": {  
    "CollectionName": "string",  
    "DatabaseName": "string",  
    "FullDocument": "string"  
},  
"Enabled": boolean,  
"FilterCriteria": {  
    "Filters": [  
        {  
            "Pattern": "string"  
        }  
    ]  
},  
"FunctionName": "string",  
"FunctionResponseTypes": [ "string" ],  
"MaximumBatchingWindowInSeconds": number,  
"MaximumRecordAgeInSeconds": number,  
"MaximumRetryAttempts": number,  
"ParallelizationFactor": number,  
"ScalingConfig": {  
    "MaximumConcurrency": number  
},  
"SourceAccessConfigurations": [  
    {  
        "Type": "string",  
        "URI": "string"  
    }  
],  
"TumblingWindowInSeconds": number  
}
```

URI Request Parameters

The request uses the following URI parameters.

[UUID \(p. 1356\)](#)

The identifier of the event source mapping.

Required: Yes

Request Body

The request accepts the following data in JSON format.

[BatchSize \(p. 1356\)](#)

The maximum number of records in each batch that Lambda pulls from your stream or queue and sends to your function. Lambda passes all of the records in the batch to the function in a single call, up to the payload limit for synchronous invocation (6 MB).

- **Amazon Kinesis** – Default 100. Max 10,000.
- **Amazon DynamoDB Streams** – Default 100. Max 10,000.
- **Amazon Simple Queue Service** – Default 10. For standard queues the max is 10,000. For FIFO queues the max is 10.
- **Amazon Managed Streaming for Apache Kafka** – Default 100. Max 10,000.
- **Self-managed Apache Kafka** – Default 100. Max 10,000.
- **Amazon MQ (ActiveMQ and RabbitMQ)** – Default 100. Max 10,000.
- **DocumentDB** – Default 100. Max 10,000.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

Required: No

BisectBatchOnFunctionError (p. 1356)

(Kinesis and DynamoDB Streams only) If the function returns an error, split the batch in two and retry.

Type: Boolean

Required: No

DestinationConfig (p. 1356)

(Kinesis and DynamoDB Streams only) A standard Amazon SQS queue or standard Amazon SNS topic destination for discarded records.

Type: [DestinationConfig \(p. 1413\)](#) object

Required: No

DocumentDBEventSourceConfig (p. 1356)

Specific configuration settings for a DocumentDB event source.

Type: [DocumentDBEventSourceConfig \(p. 1414\)](#) object

Required: No

Enabled (p. 1356)

When true, the event source mapping is active. When false, Lambda pauses polling and invocation.

Default: True

Type: Boolean

Required: No

FilterCriteria (p. 1356)

An object that defines the filter criteria that determine whether Lambda should process an event. For more information, see [Lambda event filtering](#).

Type: [FilterCriteria \(p. 1426\)](#) object

Required: No

FunctionName (p. 1356)

The name of the Lambda function.

Name formats

- **Function name** – MyFunction.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:MyFunction.
- **Version or Alias ARN** – arn:aws:lambda:us-west-2:123456789012:function:MyFunction:PROD.
- **Partial ARN** – 123456789012:function:MyFunction.

The length constraint applies only to the full ARN. If you specify only the function name, it's limited to 64 characters in length.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (`arn:(aws[a-zA-Z-]*)::lambda:`)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: No

[FunctionResponseTypes \(p. 1356\)](#)

(Kinesis, DynamoDB Streams, and Amazon SQS) A list of current response type enums applied to the event source mapping.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 1 item.

Valid Values: ReportBatchItemFailures

Required: No

[MaximumBatchingWindowInSeconds \(p. 1356\)](#)

The maximum amount of time, in seconds, that Lambda spends gathering records before invoking the function. You can configure MaximumBatchingWindowInSeconds to any value from 0 seconds to 300 seconds in increments of seconds.

For streams and Amazon SQS event sources, the default batching window is 0 seconds.

For Amazon MSK, Self-managed Apache Kafka, Amazon MQ, and DocumentDB event sources, the default batching window is 500 ms. Note that because you can only change MaximumBatchingWindowInSeconds in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

Related setting: For streams and Amazon SQS event sources, when you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

Required: No

[MaximumRecordAgeInSeconds \(p. 1356\)](#)

(Kinesis and DynamoDB Streams only) Discard records older than the specified age. The default value is infinite (-1).

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 604800.

Required: No

[MaximumRetryAttempts \(p. 1356\)](#)

(Kinesis and DynamoDB Streams only) Discard records after the specified number of retries. The default value is infinite (-1). When set to infinite (-1), failed records are retried until the record expires.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

Required: No

[ParallelizationFactor \(p. 1356\)](#)

(Kinesis and DynamoDB Streams only) The number of batches to process from each shard concurrently.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10.

Required: No

[ScalingConfig \(p. 1356\)](#)

(Amazon SQS only) The scaling configuration for the event source. For more information, see [Configuring maximum concurrency for Amazon SQS event sources](#).

Type: [ScalingConfig \(p. 1458\)](#) object

Required: No

[SourceAccessConfigurations \(p. 1356\)](#)

An array of authentication protocols or VPC components required to secure your event source.

Type: Array of [SourceAccessConfiguration \(p. 1463\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

Required: No

[TumblingWindowInSeconds \(p. 1356\)](#)

(Kinesis and DynamoDB Streams only) The duration in seconds of a processing window for DynamoDB and Kinesis Streams event sources. A value of 0 seconds indicates no tumbling window.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

Required: No

Response Syntax

```
HTTP/1.1 202
Content-type: application/json

{
    "AmazonManagedKafkaEventSourceConfig": {
        "ConsumerGroupId": "string"
    },
    "BatchSize": number,
    "BisectBatchOnFunctionError": boolean,
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "DocumentDBEventSourceConfig": {
        "CollectionName": "string",
        "ProjectionType": "string"
    }
}
```

```

        "DatabaseName": "string",
        "FullDocument": "string"
    },
    "EventSourceArn": "string",
    "FilterCriteria": {
        "Filters": [
            {
                "Pattern": "string"
            }
        ]
    },
    "FunctionArn": "string",
    "FunctionResponseTypes": [ "string" ],
    "LastModified": number,
    "LastProcessingResult": "string",
    "MaximumBatchingWindowInSeconds": number,
    "MaximumRecordAgeInSeconds": number,
    "MaximumRetryAttempts": number,
    "ParallelizationFactor": number,
    "Queues": [ "string" ],
    "ScalingConfig": {
        "MaximumConcurrency": number
    },
    "SelfManagedEventSource": {
        "Endpoints": {
            "string" : [ "string" ]
        }
    },
    "SelfManagedKafkaEventSourceConfig": {
        "ConsumerGroupId": "string"
    },
    "SourceAccessConfigurations": [
        {
            "Type": "string",
            "URI": "string"
        }
    ],
    "StartingPosition": "string",
    "StartingPositionTimestamp": number,
    "State": "string",
    "StateTransitionReason": "string",
    "Topics": [ "string" ],
    "TumblingWindowInSeconds": number,
    "UUID": "string"
}

```

Response Elements

If the action is successful, the service sends back an HTTP 202 response.

The following data is returned in JSON format by the service.

[AmazonManagedKafkaEventSourceConfig \(p. 1360\)](#)

Specific configuration settings for an Amazon Managed Streaming for Apache Kafka (Amazon MSK) event source.

Type: [AmazonManagedKafkaEventSourceConfig \(p. 1405\)](#) object

[BatchSize \(p. 1360\)](#)

The maximum number of records in each batch that Lambda pulls from your stream or queue and sends to your function. Lambda passes all of the records in the batch to the function in a single call, up to the payload limit for synchronous invocation (6 MB).

Default value: Varies by service. For Amazon SQS, the default is 10. For all other services, the default is 100.

Related setting: When you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

BisectBatchOnFunctionError (p. 1360)

(Kinesis and DynamoDB Streams only) If the function returns an error, split the batch in two and retry. The default value is false.

Type: Boolean

DestinationConfig (p. 1360)

(Kinesis and DynamoDB Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: [DestinationConfig \(p. 1413\)](#) object

DocumentDBEventSourceConfig (p. 1360)

Specific configuration settings for a DocumentDB event source.

Type: [DocumentDBEventSourceConfig \(p. 1414\)](#) object

EventSourceArn (p. 1360)

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-.])+([a-z]{2}(-gov)?-[a-z]+\-\d{1})?:(\d{12})?:(.*?)`

FilterCriteria (p. 1360)

An object that defines the filter criteria that determine whether Lambda should process an event. For more information, see [Lambda event filtering](#).

Type: [FilterCriteria \(p. 1426\)](#) object

FunctionArn (p. 1360)

The ARN of the Lambda function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+\-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$\$LATEST|[a-zA-Z0-9-_]+))?`

FunctionResponseTypes (p. 1360)

(Kinesis, DynamoDB Streams, and Amazon SQS) A list of current response type enums applied to the event source mapping.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 1 item.

Valid Values: ReportBatchItemFailures

[LastModified \(p. 1360\)](#)

The date that the event source mapping was last updated or that its state changed, in Unix time seconds.

Type: Timestamp

[LastProcessingResult \(p. 1360\)](#)

The result of the last Lambda invocation of your function.

Type: String

[MaximumBatchingWindowInSeconds \(p. 1360\)](#)

The maximum amount of time, in seconds, that Lambda spends gathering records before invoking the function. You can configure MaximumBatchingWindowInSeconds to any value from 0 seconds to 300 seconds in increments of seconds.

For streams and Amazon SQS event sources, the default batching window is 0 seconds. For Amazon MSK, Self-managed Apache Kafka, Amazon MQ, and DocumentDB event sources, the default batching window is 500 ms. Note that because you can only change MaximumBatchingWindowInSeconds in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

Related setting: For streams and Amazon SQS event sources, when you set BatchSize to a value greater than 10, you must set MaximumBatchingWindowInSeconds to at least 1.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

[MaximumRecordAgeInSeconds \(p. 1360\)](#)

(Kinesis and DynamoDB Streams only) Discard records older than the specified age. The default value is -1, which sets the maximum age to infinite. When the value is set to infinite, Lambda never discards old records.

Note

The minimum valid value for maximum record age is 60s. Although values less than 60 and greater than -1 fall within the parameter's absolute range, they are not allowed

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 604800.

[MaximumRetryAttempts \(p. 1360\)](#)

(Kinesis and DynamoDB Streams only) Discard records after the specified number of retries. The default value is -1, which sets the maximum number of retries to infinite. When MaximumRetryAttempts is infinite, Lambda retries failed records until the record expires in the event source.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

[ParallelizationFactor \(p. 1360\)](#)

(Kinesis and DynamoDB Streams only) The number of batches to process concurrently from each shard. The default value is 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10.

[Queues \(p. 1360\)](#)

(Amazon MQ) The name of the Amazon MQ broker destination queue to consume.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 1000.

Pattern: [\s\S]*

[ScalingConfig \(p. 1360\)](#)

(Amazon SQS only) The scaling configuration for the event source. For more information, see [Configuring maximum concurrency for Amazon SQS event sources](#).

Type: [ScalingConfig \(p. 1458\)](#) object

[SelfManagedEventSource \(p. 1360\)](#)

The self-managed Apache Kafka cluster for your event source.

Type: [SelfManagedEventSource \(p. 1459\)](#) object

[SelfManagedKafkaEventSourceConfig \(p. 1360\)](#)

Specific configuration settings for a self-managed Apache Kafka event source.

Type: [SelfManagedKafkaEventSourceConfig \(p. 1460\)](#) object

[SourceAccessConfigurations \(p. 1360\)](#)

An array of the authentication protocol, VPC components, or virtual host to secure and define your event source.

Type: Array of [SourceAccessConfiguration \(p. 1463\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

[StartingPosition \(p. 1360\)](#)

The position in a stream from which to start reading. Required for Amazon Kinesis, Amazon DynamoDB, and Amazon MSK stream sources. AT_TIMESTAMP is supported only for Amazon Kinesis streams and Amazon DocumentDB.

Type: String

Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

[StartingPositionTimestamp \(p. 1360\)](#)

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds.

Type: Timestamp

[State \(p. 1360\)](#)

The state of the event source mapping. It can be one of the following: Creating, Enabling, Enabled, Disabling, Disabled, Updating, or Deleting.

Type: String

[StateTransitionReason \(p. 1360\)](#)

Indicates whether a user or Lambda made the last change to the event source mapping.

Type: String

[Topics \(p. 1360\)](#)

The name of the Kafka topic.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 249.

Pattern: ^[^.]([a-zA-Z0-9\-.]+)+

[TumblingWindowInSeconds \(p. 1360\)](#)

(Kinesis and DynamoDB Streams only) The duration in seconds of a processing window for DynamoDB and Kinesis Streams event sources. A value of 0 seconds indicates no tumbling window.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

[UUID \(p. 1360\)](#)

The identifier of the event source mapping.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceInUseException

The operation conflicts with the resource's availability. For example, you tried to update an event source mapping in the CREATING state, or you tried to delete an event source mapping currently UPDATING.

HTTP Status Code: 400

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

UpdateFunctionCode

Updates a Lambda function's code. If code signing is enabled for the function, the code package must be signed by a trusted publisher. For more information, see [Configuring code signing for Lambda](#).

If the function's package type is Image, then you must specify the code package in ImageUri as the URI of a [container image](#) in the Amazon ECR registry.

If the function's package type is Zip, then you must specify the deployment package as a [zip file archive](#). Enter the Amazon S3 bucket and key of the code .zip file location. You can also provide the function code inline using the ZipFile field.

The code in the deployment package must be compatible with the target instruction set architecture of the function (x86-64 or arm64).

The function's code is locked when you publish a version. You can't modify the code of a published version, only the unpublished version.

Note

For a function defined as a container image, Lambda resolves the image tag to an image digest. In Amazon ECR, if you update the image tag to a new image, Lambda does not automatically update the function.

Request Syntax

```
PUT /2015-03-31/functions/FunctionName/code HTTP/1.1
Content-type: application/json

{
    "Architectures": [ "string" ],
    "DryRun": boolean,
    "ImageUri": "string",
    "Publish": boolean,
    "RevisionId": "string",
    "S3Bucket": "string",
    "S3Key": "string",
    "S3ObjectVersion": "string",
    "ZipFile": blob
}
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName (p. 1367)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: Yes

Request Body

The request accepts the following data in JSON format.

[Architectures \(p. 1367\)](#)

The instruction set architecture that the function supports. Enter a string array with one of the valid values (arm64 or x86_64). The default value is x86_64.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: x86_64 | arm64

Required: No

[DryRun \(p. 1367\)](#)

Set to true to validate the request parameters and access permissions without modifying the function code.

Type: Boolean

Required: No

[ImageUri \(p. 1367\)](#)

URI of a container image in the Amazon ECR registry. Do not use for a function defined with a .zip file archive.

Type: String

Required: No

[Publish \(p. 1367\)](#)

Set to true to publish a new version of the function after updating the code. This has the same effect as calling [PublishVersion \(p. 1315\)](#) separately.

Type: Boolean

Required: No

[RevisionId \(p. 1367\)](#)

Update the function only if the revision ID matches the ID that's specified. Use this option to avoid modifying a function that has changed since you last read it.

Type: String

Required: No

[S3Bucket \(p. 1367\)](#)

An Amazon S3 bucket in the same AWS Region as your function. The bucket can be in a different AWS account. Use only with a function defined with a .zip file archive deployment package.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 63.

Pattern: ^[0-9A-Za-z\._\-_]*(?<!\.).\$

Required: No

[S3Key \(p. 1367\)](#)

The Amazon S3 key of the deployment package. Use only with a function defined with a .zip file archive deployment package.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Required: No

[S3ObjectVersion \(p. 1367\)](#)

For versioned objects, the version of the deployment package object to use.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Required: No

[ZipFile \(p. 1367\)](#)

The base64-encoded contents of the deployment package. AWS SDK and AWS CLI clients handle the encoding for you. Use only with a function defined with a .zip file archive deployment package.

Type: Base64-encoded binary data object

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "Architectures": [ "string" ],
  "CodeSha256": "string",
  "CodeSize": number,
  "DeadLetterConfig": {
    "TargetArn": "string"
  },
  "Description": "string",
  "Environment": {
    "Error": {
      "ErrorCode": "string",
      "Message": "string"
    },
    "Variables": {
      "string" : "string"
    }
  },
  "EphemeralStorage": {
    "Size": number
  }
}
```

```

},
"FileSystemConfigs": [
  {
    "Arn": "string",
    "LocalMountPath": "string"
  }
],
"FunctionArn": "string",
"FunctionName": "string",
"Handler": "string",
"ImageConfigResponse": {
  "Error": {
    "ErrorCode": "string",
    "Message": "string"
  },
  "ImageConfig": {
    "Command": [ "string" ],
    "EntryPoint": [ "string" ],
    "WorkingDirectory": "string"
  }
},
"KMSKeyArn": "string",
"LastModified": "string",
"LastUpdateStatus": "string",
"LastUpdateStatusReason": "string",
"LastUpdateStatusReasonCode": "string",
"Layers": [
  {
    "Arn": "string",
    "CodeSize": number,
    "SigningJobArn": "string",
    "SigningProfileVersionArn": "string"
  }
],
"MasterArn": "string",
"MemorySize": number,
"PackageType": "string",
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"RuntimeVersionConfig": {
  "Error": {
    "ErrorCode": "string",
    "Message": "string"
  },
  "RuntimeVersionArn": "string"
},
"SigningJobArn": "string",
"SigningProfileVersionArn": "string",
"SnapStart": {
  "ApplyOn": "string",
  "OptimizationStatus": "string"
},
"State": "string",
"StateReason": "string",
"StateReasonCode": "string",
"Timeout": number,
"TracingConfig": {
  "Mode": "string"
},
"Version": "string",
"VpcConfig": {
  "SecurityGroupIds": [ "string" ],
  "SubnetIds": [ "string" ],
  "VpcId": "string"
}
}

```

}

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Architectures \(p. 1369\)](#)

The instruction set architecture that the function supports. Architecture is a string array with one of the valid values. The default architecture value is x86_64.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: x86_64 | arm64

[CodeSha256 \(p. 1369\)](#)

The SHA256 hash of the function's deployment package.

Type: String

[CodeSize \(p. 1369\)](#)

The size of the function's deployment package, in bytes.

Type: Long

[DeadLetterConfig \(p. 1369\)](#)

The function's dead letter queue.

Type: [DeadLetterConfig \(p. 1412\)](#) object

[Description \(p. 1369\)](#)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[Environment \(p. 1369\)](#)

The function's [environment variables](#). Omitted from AWS CloudTrail logs.

Type: [EnvironmentResponse \(p. 1417\)](#) object

[EphemeralStorage \(p. 1369\)](#)

The size of the function's /tmp directory in MB. The default value is 512, but it can be any whole number between 512 and 10,240 MB.

Type: [EphemeralStorage \(p. 1418\)](#) object

[FileSystemConfigs \(p. 1369\)](#)

Connection settings for an [Amazon EFS file system](#).

Type: Array of [FileSystemConfig \(p. 1424\)](#) objects

Array Members: Maximum number of 1 item.

[FunctionArn \(p. 1369\)](#)

The function's Amazon Resource Name (ARN).

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:(\$\$LATEST|[a-zA-Z0-9-_]+))?`

[FunctionName \(p. 1369\)](#)

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?\d{12}:?(function:)?([a-zA-Z0-9-_\.]+)(:(\$\$LATEST|[a-zA-Z0-9-_]+))?`

[Handler \(p. 1369\)](#)

The function that Lambda calls to begin running your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

[ImageConfigResponse \(p. 1369\)](#)

The function's image configuration values.

Type: [ImageConfigResponse \(p. 1442\)](#) object

[KMSKeyArn \(p. 1369\)](#)

The AWS KMS key that's used to encrypt the function's [environment variables](#). When [Lambda SnapStart](#) is activated, this key is also used to encrypt the function's snapshot. This key is returned only if you've configured a customer managed key.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-_\.]+\.:*)|()`

[LastModified \(p. 1369\)](#)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[LastUpdateStatus \(p. 1369\)](#)

The status of the last update that was performed on the function. This is first set to Successful after function creation completes.

Type: String

Valid Values: Successful | Failed | InProgress

[LastUpdateStatusReason \(p. 1369\)](#)

The reason for the last update that was performed on the function.

Type: String

[LastUpdateStatusReasonCode \(p. 1369\)](#)

The reason code for the last update that was performed on the function.

Type: String

Valid Values: `EniLimitExceeded` | `InsufficientRolePermissions` | `InvalidConfiguration` | `InternalError` | `SubnetOutOfIPAddresses` | `InvalidSubnet` | `InvalidSecurityGroup` | `ImageDeleted` | `ImageAccessDenied` | `InvalidImage` | `KMSKeyAccessDenied` | `KMSKeyNotFound` | `InvalidStateKMSKey` | `DisabledKMSKey` | `EFSIOError` | `EFSMountConnectivityError` | `EFSMountFailure` | `EFSMountTimeout` | `InvalidRuntime` | `InvalidZipFileException` | `FunctionError`

[Layers \(p. 1369\)](#)

The function's [layers](#).

Type: Array of [Layer \(p. 1446\)](#) objects

[MasterArn \(p. 1369\)](#)

For Lambda@Edge functions, the ARN of the main function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$\$LATEST|[a-zA-Z0-9-_]+))?`

[MemorySize \(p. 1369\)](#)

The amount of memory available to the function at runtime.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 10240.

[PackageType \(p. 1369\)](#)

The type of deployment package. Set to `Image` for container image and set `Zip` for .zip file archive.

Type: String

Valid Values: `Zip` | `Image`

[RevisionId \(p. 1369\)](#)

The latest updated revision of the function or alias.

Type: String

[Role \(p. 1369\)](#)

The function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z0-9+=,.@-_/.]+`

[Runtime \(p. 1369\)](#)

The identifier of the function's [runtime](#). Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

RuntimeVersionConfig (p. 1369)

The ARN of the runtime and any errors that occurred.

Type: [RuntimeVersionConfig \(p. 1456\)](#) object

SigningJobArn (p. 1369)

The ARN of the signing job.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*)([a-zA-Z0-9\-.]+)([a-z]{2}(-gov)?-[a-zA-Z]+\d{1})?:(\d{12})?:(.*?)

SigningProfileVersionArn (p. 1369)

The ARN of the signing profile version.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*)([a-zA-Z0-9\-.]+)([a-z]{2}(-gov)?-[a-zA-Z]+\d{1})?:(\d{12})?:(.*?)

SnapStart (p. 1369)

Set `ApplyOn` to `PublishedVersions` to create a snapshot of the initialized execution environment when you publish a function version. For more information, see [Improving startup performance with Lambda SnapStart](#).

Type: [SnapStartResponse \(p. 1462\)](#) object

State (p. 1369)

The current state of the function. When the state is `Inactive`, you can reactivate the function by invoking it.

Type: String

Valid Values: Pending | Active | Inactive | Failed

StateReason (p. 1369)

The reason for the function's current state.

Type: String

StateReasonCode (p. 1369)

The reason code for the function's current state. When the code is `Creating`, you can't invoke or modify the function.

Type: String

Valid Values: Idle | Creating | Restoring | EniLimitExceeded | InsufficientRolePermissions | InvalidConfiguration | InternalError

| SubnetOutofIPAddresses | InvalidSubnet | InvalidSecurityGroup |
ImageDeleted | ImageAccessDenied | InvalidImage | KMSKeyAccessDenied
| KMSKeyNotFound | InvalidStateKMSKey | DisabledKMSKey | EFSIOError
| EFSSMountConnectivityError | EFSSMountFailure | EFSSMountTimeout |
InvalidRuntime | InvalidZipFileException | FunctionError

[Timeout \(p. 1369\)](#)

The amount of time in seconds that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

[TracingConfig \(p. 1369\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 1466\)](#) object

[Version \(p. 1369\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST|[0-9]+)

[VpcConfig \(p. 1369\)](#)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 1468\)](#) object

Errors

CodeSigningConfigNotFoundException

The specified code signing configuration does not exist.

HTTP Status Code: 404

CodeStorageExceededException

Your AWS account has exceeded its maximum total code size. For more information, see [Lambda quotas](#).

HTTP Status Code: 400

CodeVerificationFailedException

The code signature failed one or more of the validation checks for signature mismatch or expiry, and the code signing policy is set to ENFORCE. Lambda blocks the deployment.

HTTP Status Code: 400

InvalidCodeSignatureException

The code signature failed the integrity check. If the integrity check fails, then Lambda blocks deployment, even if the code signing policy is set to WARN.

HTTP Status Code: 400

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

PreconditionFailedException

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

UpdateFunctionConfiguration

Modify the version-specific settings of a Lambda function.

When you update a function, Lambda provisions an instance of the function and its supporting resources. If your function connects to a VPC, this process can take a minute. During this time, you can't modify the function, but you can still invoke it. The `LastUpdateStatus`, `LastUpdateStatusReason`, and `LastUpdateStatusReasonCode` fields in the response from [GetFunctionConfiguration \(p. 1229\)](#) indicate when the update is complete and the function is processing events with the new configuration. For more information, see [Lambda function states](#).

These settings can vary between versions of a function and are locked when you publish a version. You can't modify the configuration of a published version, only the unpublished version.

To configure function concurrency, use [PutFunctionConcurrency \(p. 1327\)](#). To grant invoke permissions to an AWS account or AWS service, use [AddPermission \(p. 1141\)](#).

Request Syntax

```
PUT /2015-03-31/functions/FunctionName/configuration HTTP/1.1
```

```
Content-type: application/json
```

```
{  
    "DeadLetterConfig        "TargetArnstring"  
    },  
    "Descriptionstring",  
    "Environment        "Variables            "string": "string"  
        }  
    },  
    "EphemeralStorage        "Sizenumber  
    },  
    "FileSystemConfigs        {  
            "Arnstring",  
            "LocalMountPathstring"  
        }  
    ],  
    "Handlerstring",  
    "ImageConfig        "Commandstring "],  
        "EntryPointstring "],  
        "WorkingDirectorystring"  
    },  
    "KMSKeyArnstring",  
    "Layersstring "],  
    "MemorySizenumber,  
    "RevisionIdstring",  
    "Rolestring",  
    "Runtimestring",  
    "SnapStart        "ApplyOnstring"  
    },  
    "Timeoutnumber,  
    "TracingConfig        "Modestring"  
    },  
    "VpcConfig        "SecurityGroupIdsstring "],  
        "SubnetIdsstring "],  
        "AssociatePublicIpAddressboolean  
    }  
}
```

```
        "SubnetIds": [ "string" ]  
    }  
}
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1377\)](#)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*):lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

Request Body

The request accepts the following data in JSON format.

[DeadLetterConfig \(p. 1377\)](#)

A dead-letter queue configuration that specifies the queue or topic where Lambda sends asynchronous events when they fail processing. For more information, see [Dead-letter queues](#).

Type: [DeadLetterConfig \(p. 1412\)](#) object

Required: No

[Description \(p. 1377\)](#)

A description of the function.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

[Environment \(p. 1377\)](#)

Environment variables that are accessible from function code during execution.

Type: [Environment \(p. 1415\)](#) object

Required: No

[EphemeralStorage \(p. 1377\)](#)

The size of the function's /tmp directory in MB. The default value is 512, but can be any whole number between 512 and 10,240 MB.

Type: [EphemeralStorage \(p. 1418\)](#) object

Required: No

[FileSystemConfigs \(p. 1377\)](#)

Connection settings for an Amazon EFS file system.

Type: Array of [FileSystemConfig \(p. 1424\)](#) objects

Array Members: Maximum number of 1 item.

Required: No

[Handler \(p. 1377\)](#)

The name of the method within your code that Lambda calls to run your function. Handler is required if the deployment package is a .zip file archive. The format includes the file name. It can also include namespaces and other qualifiers, depending on the runtime. For more information, see [Lambda programming model](#).

Type: String

Length Constraints: Maximum length of 128.

Pattern: [^\s]+

Required: No

[ImageConfig \(p. 1377\)](#)

[Container image configuration values](#) that override the values in the container image Docker file.

Type: [ImageConfig \(p. 1440\)](#) object

Required: No

[KMSKeyArn \(p. 1377\)](#)

The ARN of the AWS Key Management Service (AWS KMS) customer managed key that's used to encrypt your function's [environment variables](#). When [Lambda SnapStart](#) is activated, this key is also used to encrypt your function's snapshot. If you don't provide a customer managed key, Lambda uses a default service key.

Type: String

Pattern: (arn:(aws[a-zA-Z-]*)?:[a-z0-9-.]+:[.]*|()

Required: No

[Layers \(p. 1377\)](#)

A list of [function layers](#) to add to the function's execution environment. Specify each layer by its ARN, including the version.

Type: Array of strings

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+

Required: No

[MemorySize \(p. 1377\)](#)

The amount of [memory available to the function](#) at runtime. Increasing the function memory also increases its CPU allocation. The default value is 128 MB. The value can be any multiple of 1 MB.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 10240.

Required: No

[RevisionId \(p. 1377\)](#)

Update the function only if the revision ID matches the ID that's specified. Use this option to avoid modifying a function that has changed since you last read it.

Type: String

Required: No

[Role \(p. 1377\)](#)

The Amazon Resource Name (ARN) of the function's execution role.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/[a-zA-Z_0-9+=,.@\\-_/.]+

Required: No

[Runtime \(p. 1377\)](#)

The identifier of the function's [runtime](#). Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

Required: No

[SnapStart \(p. 1377\)](#)

The function's [SnapStart](#) setting.

Type: [SnapStart \(p. 1461\)](#) object

Required: No

[Timeout \(p. 1377\)](#)

The amount of time (in seconds) that Lambda allows a function to run before stopping it. The default is 3 seconds. The maximum allowed value is 900 seconds. For more information, see [Lambda execution environment](#).

Type: Integer

Valid Range: Minimum value of 1.

Required: No

[TracingConfig \(p. 1377\)](#)

Set Mode to Active to sample and trace a subset of incoming requests with [X-Ray](#).

Type: [TracingConfig \(p. 1465\)](#) object

Required: No

[VpcConfig \(p. 1377\)](#)

For network connectivity to AWS resources in a VPC, specify a list of security groups and subnets in the VPC. When you connect a function to a VPC, it can access resources and the internet only through that VPC. For more information, see [Configuring a Lambda function to access resources in a VPC](#).

Type: [VpcConfig \(p. 1467\)](#) object

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "Architectures": [ "string" ],
    "CodeSha256": "string",
    "CodeSize": number,
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
    "Description": "string",
    "Environment": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "Variables": {
            "string" : "string"
        }
    },
    "EphemeralStorage": {
        "Size": number
    },
    "FileSystemConfigs": [
        {
            "Arn": "string",
            "LocalMountPath": "string"
        }
    ],
    "FunctionArn": "string",
    "FunctionName": "string",
    "Handler": "string",
    "ImageConfigResponse": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "ImageConfig": {

```

```

        "Command": [ "string" ],
        "EntryPoint": [ "string" ],
        "WorkingDirectory": "string"
    }
},
"KMSKeyArn": "string",
"LastModified": "string",
"LastUpdateStatus": "string",
"LastUpdateStatusReason": "string",
"LastUpdateStatusReasonCode": "string",
"Layers": [
    {
        "Arn": "string",
        "CodeSize": number,
        "SigningJobArn": "string",
        "SigningProfileVersionArn": "string"
    }
],
"MasterArn": "string",
"MemorySize": number,
"PackageType": "string",
"RevisionId": "string",
"Role": "string",
"Runtime": "string",
"RuntimeVersionConfig": {
    "Error": {
        "ErrorCode": "string",
        "Message": "string"
    },
    "RuntimeVersionArn": "string"
},
"SigningJobArn": "string",
"SigningProfileVersionArn": "string",
"SnapStart": {
    "ApplyOn": "string",
    "OptimizationStatus": "string"
},
"State": "string",
"StateReason": "string",
"StateReasonCode": "string",
"Timeout": number,
"TracingConfig": {
    "Mode": "string"
},
"Version": "string",
"VpcConfig": {
    "SecurityGroupIds": [ "string" ],
    "SubnetIds": [ "string" ],
    "VpcId": "string"
}
}
}

```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[Architectures \(p. 1381\)](#)

The instruction set architecture that the function supports. Architecture is a string array with one of the valid values. The default architecture value is x86_64.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: x86_64 | arm64

CodeSha256 (p. 1381)

The SHA256 hash of the function's deployment package.

Type: String

CodeSize (p. 1381)

The size of the function's deployment package, in bytes.

Type: Long

DeadLetterConfig (p. 1381)

The function's dead letter queue.

Type: [DeadLetterConfig \(p. 1412\)](#) object

Description (p. 1381)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Environment (p. 1381)

The function's [environment variables](#). Omitted from AWS CloudTrail logs.

Type: [EnvironmentResponse \(p. 1417\)](#) object

EphemeralStorage (p. 1381)

The size of the function's /tmp directory in MB. The default value is 512, but it can be any whole number between 512 and 10,240 MB.

Type: [EphemeralStorage \(p. 1418\)](#) object

FileSystemConfigs (p. 1381)

Connection settings for an [Amazon EFS file system](#).

Type: Array of [FileSystemConfig \(p. 1424\)](#) objects

Array Members: Maximum number of 1 item.

FunctionArn (p. 1381)

The function's Amazon Resource Name (ARN).

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:(\$LATEST|[a-zA-Z0-9-_]+))?

FunctionName (p. 1381)

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_\.]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

[Handler \(p. 1381\)](#)

The function that Lambda calls to begin running your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

[ImageConfigResponse \(p. 1381\)](#)

The function's image configuration values.

Type: [ImageConfigResponse \(p. 1442\)](#) object

[KMSKeyArn \(p. 1381\)](#)

The AWS KMS key that's used to encrypt the function's [environment variables](#). When [Lambda SnapStart](#) is activated, this key is also used to encrypt the function's snapshot. This key is returned only if you've configured a customer managed key.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-.]+:[^.]+|())`

[LastModified \(p. 1381\)](#)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[LastUpdateStatus \(p. 1381\)](#)

The status of the last update that was performed on the function. This is first set to Successful after function creation completes.

Type: String

Valid Values: Successful | Failed | InProgress

[LastUpdateStatusReason \(p. 1381\)](#)

The reason for the last update that was performed on the function.

Type: String

[LastUpdateStatusReasonCode \(p. 1381\)](#)

The reason code for the last update that was performed on the function.

Type: String

Valid Values: EniLimitExceeded | InsufficientRolePermissions | InvalidConfiguration | InternalError | SubnetOutOfRangeAddresses | InvalidSubnet | InvalidSecurityGroup | ImageDeleted | ImageAccessDenied | InvalidImage | KMSKeyAccessDenied | KMSKeyNotFound | InvalidStateKMSKey | DisabledKMSKey | EFSIOError | EFSMountConnectivityError | EFSMountFailure | EFSMountTimeout | InvalidRuntime | InvalidZipFileException | FunctionError

[Layers \(p. 1381\)](#)

The function's [layers](#).

Type: Array of [Layer \(p. 1446\)](#) objects

MasterArn (p. 1381)

For Lambda@Edge functions, the ARN of the main function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:($LATEST|[a-zA-Z0-9-_+]))?`

MemorySize (p. 1381)

The amount of memory available to the function at runtime.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 10240.

PackageType (p. 1381)

The type of deployment package. Set to Image for container image and set Zip for .zip file archive.

Type: String

Valid Values: Zip | Image

RevisionId (p. 1381)

The latest updated revision of the function or alias.

Type: String

Role (p. 1381)

The function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/[a-zA-Z0-9+=,.@-_/.]+`

Runtime (p. 1381)

The identifier of the function's [runtime](#). Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

RuntimeVersionConfig (p. 1381)

The ARN of the runtime and any errors that occurred.

Type: [RuntimeVersionConfig \(p. 1456\)](#) object

SigningJobArn (p. 1381)

The ARN of the signing job.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+:(a-z){2}(-gov)?-[a-z]+-\d{1}?:(\d{12})?:(.*)

[SigningProfileVersionArn \(p. 1381\)](#)

The ARN of the signing profile version.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+:(a-z){2}(-gov)?-[a-z]+-\d{1}?:(\d{12})?:(.*)

[SnapStart \(p. 1381\)](#)

Set ApplyOn to PublishedVersions to create a snapshot of the initialized execution environment when you publish a function version. For more information, see [Improving startup performance with Lambda SnapStart](#).

Type: [SnapStartResponse \(p. 1462\)](#) object

[State \(p. 1381\)](#)

The current state of the function. When the state is Inactive, you can reactivate the function by invoking it.

Type: String

Valid Values: Pending | Active | Inactive | Failed

[StateReason \(p. 1381\)](#)

The reason for the function's current state.

Type: String

[StateReasonCode \(p. 1381\)](#)

The reason code for the function's current state. When the code is Creating, you can't invoke or modify the function.

Type: String

Valid Values: Idle | Creating | Restoring | EniLimitExceeded | InsufficientRolePermissions | InvalidConfiguration | InternalError | SubnetOutOfRange | InvalidSubnet | InvalidSecurityGroup | ImageDeleted | ImageAccessDenied | InvalidImage | KMSKeyAccessDenied | KMSKeyNotFound | InvalidStateKMSKey | DisabledKMSKey | EFSIOError | EFSSMountConnectivityError | EFSSMountFailure | EFSSMountTimeout | InvalidRuntime | InvalidZipFileException | FunctionError

[Timeout \(p. 1381\)](#)

The amount of time in seconds that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

[TracingConfig \(p. 1381\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 1466\)](#) object

[Version \(p. 1381\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST | [0-9]+)

[VpcConfig \(p. 1381\)](#)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 1468\)](#) object

Errors

CodeSigningConfigNotFoundException

The specified code signing configuration does not exist.

HTTP Status Code: 404

CodeVerificationFailedException

The code signature failed one or more of the validation checks for signature mismatch or expiry, and the code signing policy is set to ENFORCE. Lambda blocks the deployment.

HTTP Status Code: 400

InvalidCodeSignatureException

The code signature failed the integrity check. If the integrity check fails, then Lambda blocks deployment, even if the code signing policy is set to WARN.

HTTP Status Code: 400

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

PreconditionFailedException

The RevisionId provided does not match the latest RevisionId for the Lambda function or alias. Call the GetFunction or the GetAlias API operation to retrieve the latest RevisionId for your resource.

HTTP Status Code: 412

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

UpdateFunctionEventInvokeConfig

Updates the configuration for asynchronous invocation for a function, version, or alias.

To configure options for asynchronous invocation, use [PutFunctionEventInvokeConfig \(p. 1330\)](#).

Request Syntax

```
POST /2019-09-25/functions/FunctionName/event-invoke-config?Qualifier=Qualifier HTTP/1.1
Content-type: application/json

{
  "DestinationConfigOnFailure": {
      "DestinationOnSuccess": {
      "DestinationMaximumEventAgeInSecondsMaximumRetryAttempts
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 1389\)](#)

The name of the Lambda function, version, or alias.

Name formats

- **Function name** - my-function (name-only), my-function:v1 (with alias).
- **Function ARN** - arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** - 123456789012:function:my-function.

You can append a version number or alias to any of the formats. The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

[Qualifier \(p. 1389\)](#)

A version number or alias name.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: ([a-zA-Z0-9\$-_]+)

Request Body

The request accepts the following data in JSON format.

[DestinationConfig \(p. 1389\)](#)

A destination for events after they have been sent to a function for processing.

Destinations

- **Function** - The Amazon Resource Name (ARN) of a Lambda function.
- **Queue** - The ARN of a standard SQS queue.
- **Topic** - The ARN of a standard SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: [DestinationConfig \(p. 1413\)](#) object

Required: No

[MaximumEventAgeInSeconds \(p. 1389\)](#)

The maximum age of a request that Lambda sends to a function for processing.

Type: Integer

Valid Range: Minimum value of 60. Maximum value of 21600.

Required: No

[MaximumRetryAttempts \(p. 1389\)](#)

The maximum number of times to retry when the function returns an error.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 2.

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "DestinationConfig": {
    "OnFailureDestinationOnSuccessDestinationFunctionArnLastModifiedMaximumEventAgeInSecondsMaximumRetryAttempts
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

[DestinationConfig \(p. 1390\)](#)

A destination for events after they have been sent to a function for processing.

Destinations

- **Function** - The Amazon Resource Name (ARN) of a Lambda function.
- **Queue** - The ARN of a standard SQS queue.
- **Topic** - The ARN of a standard SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: [DestinationConfig \(p. 1413\)](#) object

[FunctionArn \(p. 1390\)](#)

The Amazon Resource Name (ARN) of the function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?

[LastModified \(p. 1390\)](#)

The date and time that the configuration was last updated, in Unix time seconds.

Type: Timestamp

[MaximumEventAgeInSeconds \(p. 1390\)](#)

The maximum age of a request that Lambda sends to a function for processing.

Type: Integer

Valid Range: Minimum value of 60. Maximum value of 21600.

[MaximumRetryAttempts \(p. 1390\)](#)

The maximum number of times to retry when the function returns an error.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 2.

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

UpdateFunctionUrlConfig

Updates the configuration for a Lambda function URL.

Request Syntax

```
PUT /2021-10-31/functions/FunctionName/url?Qualifier=Qualifier HTTP/1.1
Content-type: application/json

{
  "AuthType": "string",
  "Cors: {
    "AllowCredentials": boolean,
    "AllowHeaders": [ "string" ],
    "AllowMethods": [ "string" ],
    "AllowOrigins": [ "string" ],
    "ExposeHeaders": [ "string" ],
    "MaxAge": number
  },
  "InvokeMode": "string"
}
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName (p. 1393)

The name of the Lambda function.

Name formats

- **Function name** – my-function.
- **Function ARN** – arn:aws:lambda:us-west-2:123456789012:function:my-function.
- **Partial ARN** – 123456789012:function:my-function.

The length constraint applies only to the full ARN. If you specify only the function name, it is limited to 64 characters in length.

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: (arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)(:(\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

Qualifier (p. 1393)

The alias name.

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (^\\$LATEST\$)|((?!^[\d-]).+)([\d-])

Request Body

The request accepts the following data in JSON format.

[AuthType \(p. 1393\)](#)

The type of authentication that your function URL uses. Set to AWS_IAM if you want to restrict access to authenticated users only. Set to NONE if you want to bypass IAM authentication to create a public endpoint. For more information, see [Security and auth model for Lambda function URLs](#).

Type: String

Valid Values: NONE | AWS_IAM

Required: No

[Cors \(p. 1393\)](#)

The [cross-origin resource sharing \(CORS\)](#) settings for your function URL.

Type: [Cors \(p. 1410\)](#) object

Required: No

[InvokeMode \(p. 1393\)](#)

Use one of the following options:

- BUFFERED – This is the default option. Lambda invokes your function using the Invoke API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.
- RESPONSE_STREAM – Your function streams payload results as they become available. Lambda invokes your function using the InvokeWithResponseStream API operation. The maximum response payload size is 20 MB, however, you can [request a quota increase](#).

Type: String

Valid Values: BUFFERED | RESPONSE_STREAM

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
  "AuthType": "string",
  "Cors": {
    "AllowCredentials": boolean,
    "AllowHeaders": [ "string" ],
    "AllowMethods": [ "string" ],
    "AllowOrigins": [ "string" ],
    "ExposeHeaders": [ "string" ],
    "MaxAge": number
  },
  "CreationTime": "string",
  "FunctionArn": "string",
  "FunctionUrl": "string",
  "InvokeMode": "string",
  "LastModifiedTime": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

AuthType (p. 1394)

The type of authentication that your function URL uses. Set to AWS_IAM if you want to restrict access to authenticated users only. Set to NONE if you want to bypass IAM authentication to create a public endpoint. For more information, see [Security and auth model for Lambda function URLs](#).

Type: String

Valid Values: NONE | AWS_IAM

Cors (p. 1394)

The [cross-origin resource sharing \(CORS\)](#) settings for your function URL.

Type: [Cors \(p. 1410\)](#) object

CreationTime (p. 1394)

When the function URL was created, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

FunctionArn (p. 1394)

The Amazon Resource Name (ARN) of your function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?

FunctionUrl (p. 1394)

The HTTP URL endpoint for your function.

Type: String

Length Constraints: Minimum length of 40. Maximum length of 100.

InvokeMode (p. 1394)

Use one of the following options:

- BUFFERED – This is the default option. Lambda invokes your function using the Invoke API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.
- RESPONSE_STREAM – Your function streams payload results as they become available. Lambda invokes your function using the InvokeWithResponseStream API operation. The maximum response payload size is 20 MB, however, you can [request a quota increase](#).

Type: String

Valid Values: BUFFERED | RESPONSE_STREAM

LastModifiedTime (p. 1394)

When the function URL configuration was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is not valid.

HTTP Status Code: 400

ResourceConflictException

The resource already exists, or another operation is in progress.

HTTP Status Code: 409

ResourceNotFoundException

The resource specified in the request does not exist.

HTTP Status Code: 404

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded. For more information, see [Lambda quotas](#).

HTTP Status Code: 429

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS Command Line Interface](#)
- [AWS SDK for .NET](#)
- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for JavaScript](#)
- [AWS SDK for PHP V3](#)
- [AWS SDK for Python](#)
- [AWS SDK for Ruby V3](#)

Data Types

The following data types are supported:

- [AccountLimit \(p. 1399\)](#)
- [AccountUsage \(p. 1400\)](#)
- [AliasConfiguration \(p. 1401\)](#)
- [AliasRoutingConfiguration \(p. 1403\)](#)
- [AllowedPublishers \(p. 1404\)](#)

- [AmazonManagedKafkaEventSourceConfig \(p. 1405\)](#)
- [CodeSigningConfig \(p. 1406\)](#)
- [CodeSigningPolicies \(p. 1408\)](#)
- [Concurrency \(p. 1409\)](#)
- [Cors \(p. 1410\)](#)
- [DeadLetterConfig \(p. 1412\)](#)
- [DestinationConfig \(p. 1413\)](#)
- [DocumentDBEventSourceConfig \(p. 1414\)](#)
- [Environment \(p. 1415\)](#)
- [EnvironmentError \(p. 1416\)](#)
- [EnvironmentResponse \(p. 1417\)](#)
- [EphemeralStorage \(p. 1418\)](#)
- [EventSourceMappingConfiguration \(p. 1419\)](#)
- [FileSystemConfig \(p. 1424\)](#)
- [Filter \(p. 1425\)](#)
- [FilterCriteria \(p. 1426\)](#)
- [FunctionCode \(p. 1427\)](#)
- [FunctionCodeLocation \(p. 1429\)](#)
- [FunctionConfiguration \(p. 1430\)](#)
- [FunctionEventInvokeConfig \(p. 1436\)](#)
- [FunctionUrlConfig \(p. 1438\)](#)
- [ImageConfig \(p. 1440\)](#)
- [ImageConfigError \(p. 1441\)](#)
- [ImageConfigResponse \(p. 1442\)](#)
- [InvokeResponseStreamUpdate \(p. 1443\)](#)
- [InvokeWithResponseStreamCompleteEvent \(p. 1444\)](#)
- [InvokeWithResponseStreamResponseEvent \(p. 1445\)](#)
- [Layer \(p. 1446\)](#)
- [LayersListItem \(p. 1447\)](#)
- [LayerVersionContentInput \(p. 1448\)](#)
- [LayerVersionContentOutput \(p. 1449\)](#)
- [LayerVersionsListItem \(p. 1450\)](#)
- [OnFailure \(p. 1452\)](#)
- [OnSuccess \(p. 1453\)](#)
- [ProvisionedConcurrencyConfigListItem \(p. 1454\)](#)
- [RuntimeVersionConfig \(p. 1456\)](#)
- [RuntimeVersionError \(p. 1457\)](#)
- [ScalingConfig \(p. 1458\)](#)
- [SelfManagedEventSource \(p. 1459\)](#)
- [SelfManagedKafkaEventSourceConfig \(p. 1460\)](#)
- [SnapStart \(p. 1461\)](#)
- [SnapStartResponse \(p. 1462\)](#)
- [SourceAccessConfiguration \(p. 1463\)](#)
- [TracingConfig \(p. 1465\)](#)
- [TracingConfigResponse \(p. 1466\)](#)
- [VpcConfig \(p. 1467\)](#)

- [VpcConfigResponse \(p. 1468\)](#)

AccountLimit

Limits that are related to concurrency and storage. All file and storage sizes are in bytes.

Contents

CodeSizeUnzipped

The maximum size of a function's deployment package and layers when they're extracted.

Type: Long

Required: No

CodeSizeZipped

The maximum size of a deployment package when it's uploaded directly to Lambda. Use Amazon S3 for larger files.

Type: Long

Required: No

ConcurrentExecutions

The maximum number of simultaneous function executions.

Type: Integer

Required: No

TotalCodeSize

The amount of storage space that you can use for all deployment packages and layer archives.

Type: Long

Required: No

UnreservedConcurrentExecutions

The maximum number of simultaneous function executions, minus the capacity that's reserved for individual functions with [PutFunctionConcurrency \(p. 1327\)](#).

Type: Integer

Valid Range: Minimum value of 0.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

AccountUsage

The number of functions and amount of storage in use.

Contents

FunctionCount

The number of Lambda functions.

Type: Long

Required: No

TotalCodeSize

The amount of storage space, in bytes, that's being used by deployment packages and layer archives.

Type: Long

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

AliasConfiguration

Provides configuration information about a Lambda function [alias](#).

Contents

AliasArn

The Amazon Resource Name (ARN) of the alias.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: No

Description

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

FunctionVersion

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

Required: No

Name

The name of the alias.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(?!^[\0-9]+$)([a-zA-Z0-9-_]+)`

Required: No

RevisionId

A unique identifier that changes when you update the alias.

Type: String

Required: No

RoutingConfig

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 1403\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

AliasRoutingConfiguration

The [traffic-shifting](#) configuration of a Lambda function alias.

Contents

AdditionalVersionWeights

The second version, and the percentage of traffic that's routed to it.

Type: String to double map

Key Length Constraints: Minimum length of 1. Maximum length of 1024.

Key Pattern: [0-9]+

Valid Range: Minimum value of 0.0. Maximum value of 1.0.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

AllowedPublishers

List of signing profiles that can sign a code package.

Contents

SigningProfileVersionArns

The Amazon Resource Name (ARN) for each of the signing profiles. A signing profile defines a trusted user who can sign a code package.

Type: Array of strings

Array Members: Minimum number of 1 item. Maximum number of 20 items.

Pattern: `arn:(aws[a-zA-Z0-9-]*)([a-zA-Z0-9\-.])+([a-z]{2}(-gov)?-[a-z]+\d{1})?:(\d{12})?:(.*)`

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

AmazonManagedKafkaEventSourceConfig

Specific configuration settings for an Amazon Managed Streaming for Apache Kafka (Amazon MSK) event source.

Contents

ConsumerGroupId

The identifier for the Kafka consumer group to join. The consumer group ID must be unique among all your Kafka event sources. After creating a Kafka event source mapping with the consumer group ID specified, you cannot update this value. For more information, see [Customizable consumer group ID](#).

Type: String

Length Constraints: Minimum length of 1. Maximum length of 200.

Pattern: [a-zA-Z0-9-\/*:_+=.@-]*

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

CodeSigningConfig

Details about a [Code signing configuration](#).

Contents

AllowedPublishers

List of allowed publishers.

Type: [AllowedPublishers \(p. 1404\)](#) object

Required: Yes

CodeSigningConfigArn

The Amazon Resource Name (ARN) of the Code signing configuration.

Type: String

Length Constraints: Maximum length of 200.

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}((-gov)|(-iso(b?)))?-([a-z]+-\d{1}:\d{12}):code-signing-config:csc-[a-zA-Z0-9]{17}

Required: Yes

CodeSigningConfigId

Unique identifier for the Code signing configuration.

Type: String

Pattern: csc-[a-zA-Z0-9-_\.]{17}

Required: Yes

CodeSigningPolicies

The code signing policy controls the validation failure action for signature mismatch or expiry.

Type: [CodeSigningPolicies \(p. 1408\)](#) object

Required: Yes

Description

Code signing configuration description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

LastModified

The date and time that the Code signing configuration was last modified, in ISO-8601 format (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

CodeSigningPolicies

Code signing configuration [policies](#) specify the validation failure action for signature mismatch or expiry.

Contents

UntrustedArtifactOnDeployment

Code signing configuration policy for deployment validation failure. If you set the policy to Enforce, Lambda blocks the deployment request if signature validation checks fail. If you set the policy to Warn, Lambda allows the deployment and creates a CloudWatch log.

Default value: Warn

Type: String

Valid Values: Warn | Enforce

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Concurrency

Contents

ReservedConcurrentExecutions

The number of concurrent executions that are reserved for this function. For more information, see [Managing Lambda reserved concurrency](#).

Type: Integer

Valid Range: Minimum value of 0.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Cors

The [cross-origin resource sharing \(CORS\)](#) settings for your Lambda function URL. Use CORS to grant access to your function URL from any origin. You can also use CORS to control access for specific HTTP headers and methods in requests to your function URL.

Contents

AllowCredentials

Whether to allow cookies or other credentials in requests to your function URL. The default is `false`.

Type: Boolean

Required: No

AllowHeaders

The HTTP headers that origins can include in requests to your function URL. For example: `Date`, `Keep-Alive`, `X-Custom-Header`.

Type: Array of strings

Array Members: Maximum number of 100 items.

Length Constraints: Maximum length of 1024.

Pattern: `.*`

Required: No

AllowMethods

The HTTP methods that are allowed when calling your function URL. For example: `GET`, `POST`, `DELETE`, or the wildcard character `(*)`.

Type: Array of strings

Array Members: Maximum number of 6 items.

Length Constraints: Maximum length of 6.

Pattern: `.*`

Required: No

AllowOrigins

The origins that can access your function URL. You can list any number of specific origins, separated by a comma. For example: `https://www.example.com`, `http://localhost:60905`.

Alternatively, you can grant access to all origins using the wildcard character `(*)`.

Type: Array of strings

Array Members: Maximum number of 100 items.

Length Constraints: Minimum length of 1. Maximum length of 253.

Pattern: `.*`

Required: No

ExposeHeaders

The HTTP headers in your function response that you want to expose to origins that call your function URL. For example: Date, Keep-Alive, X-Custom-Header.

Type: Array of strings

Array Members: Maximum number of 100 items.

Length Constraints: Maximum length of 1024.

Pattern: .*

Required: No

MaxAge

The maximum amount of time, in seconds, that web browsers can cache results of a preflight request. By default, this is set to 0, which means that the browser doesn't cache results.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 86400.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

DeadLetterConfig

The [dead-letter queue](#) for failed asynchronous invocations.

Contents

TargetArn

The Amazon Resource Name (ARN) of an Amazon SQS queue or Amazon SNS topic.

Type: String

Pattern: (`arn:(aws[a-zA-Z-]*):[a-zA-Z0-9-.]+:[.*]|()`)

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

DestinationConfig

A configuration object that specifies the destination of an event after Lambda processes it.

Contents

OnFailure

The destination configuration for failed invocations.

Type: [OnFailure \(p. 1452\)](#) object

Required: No

OnSuccess

The destination configuration for successful invocations.

Type: [OnSuccess \(p. 1453\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

DocumentDBEventSourceConfig

Specific configuration settings for a DocumentDB event source.

Contents

CollectionName

The name of the collection to consume within the database. If you do not specify a collection, Lambda consumes all collections.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 57.

Pattern: `(^(!system\x2e))(^[_a-zA-Z0-9])([^$]*)`

Required: No

DatabaseName

The name of the database to consume within the DocumentDB cluster.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 63.

Pattern: `[^ /.\$\x22]*`

Required: No

FullDocument

Determines what DocumentDB sends to your event stream during document update operations. If set to UpdateLookup, DocumentDB sends a delta describing the changes, along with a copy of the entire document. Otherwise, DocumentDB sends only a partial document that contains the changes.

Type: String

Valid Values: UpdateLookup | Default

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Environment

A function's environment variable settings. You can use environment variables to adjust your function's behavior without updating code. An environment variable is a pair of strings that are stored in a function's version-specific configuration.

Contents

Variables

Environment variable key-value pairs. For more information, see [Using Lambda environment variables](#).

Type: String to string map

Key Pattern: [a-zA-Z]([a-zA-Z0-9_])+

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

EnvironmentError

Error messages for environment variables that couldn't be applied.

Contents

ErrorCode

The error code.

Type: String

Required: No

Message

The error message.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

EnvironmentResponse

The results of an operation to update or read environment variables. If the operation succeeds, the response contains the environment variables. If it fails, the response contains details about the error.

Contents

Error

Error messages for environment variables that couldn't be applied.

Type: [EnvironmentError \(p. 1416\)](#) object

Required: No

Variables

Environment variable key-value pairs. Omitted from AWS CloudTrail logs.

Type: String to string map

Key Pattern: [a-zA-Z]([a-zA-Z0-9_])⁺

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

EphemeralStorage

The size of the function's /tmp directory in MB. The default value is 512, but it can be any whole number between 512 and 10,240 MB.

Contents

Size

The size of the function's /tmp directory.

Type: Integer

Valid Range: Minimum value of 512. Maximum value of 10240.

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

EventSourceMappingConfiguration

A mapping between an AWS resource and a Lambda function. For details, see [CreateEventSourceMapping \(p. 1153\)](#).

Contents

AmazonManagedKafkaEventSourceConfig

Specific configuration settings for an Amazon Managed Streaming for Apache Kafka (Amazon MSK) event source.

Type: [AmazonManagedKafkaEventSourceConfig \(p. 1405\)](#) object

Required: No

BatchSize

The maximum number of records in each batch that Lambda pulls from your stream or queue and sends to your function. Lambda passes all of the records in the batch to the function in a single call, up to the payload limit for synchronous invocation (6 MB).

Default value: Varies by service. For Amazon SQS, the default is 10. For all other services, the default is 100.

Related setting: When you set `BatchSize` to a value greater than 10, you must set `MaximumBatchingWindowInSeconds` to at least 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

Required: No

BisectBatchOnFunctionError

(Kinesis and DynamoDB Streams only) If the function returns an error, split the batch in two and retry. The default value is false.

Type: Boolean

Required: No

DestinationConfig

(Kinesis and DynamoDB Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: [DestinationConfig \(p. 1413\)](#) object

Required: No

DocumentDBEventSourceConfig

Specific configuration settings for a DocumentDB event source.

Type: [DocumentDBEventSourceConfig \(p. 1414\)](#) object

Required: No

EventSourceArn

The Amazon Resource Name (ARN) of the event source.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+:(a-z){2}(-gov)?-[a-z]+\d{1}:(\d{12})?:(.*?)

Required: No

FilterCriteria

An object that defines the filter criteria that determine whether Lambda should process an event.
For more information, see [Lambda event filtering](#).

Type: [FilterCriteria \(p. 1426\)](#) object

Required: No

FunctionArn

The ARN of the Lambda function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$LATEST|[a-zA-Z0-9-_+]))?

Required: No

FunctionResponseTypes

(Kinesis, DynamoDB Streams, and Amazon SQS) A list of current response type enums applied to the event source mapping.

Type: Array of strings

Array Members: Minimum number of 0 items. Maximum number of 1 item.

Valid Values: ReportBatchItemFailures

Required: No

LastModified

The date that the event source mapping was last updated or that its state changed, in Unix time seconds.

Type: Timestamp

Required: No

LastProcessingResult

The result of the last Lambda invocation of your function.

Type: String

Required: No

MaximumBatchingWindowInSeconds

The maximum amount of time, in seconds, that Lambda spends gathering records before invoking the function. You can configure MaximumBatchingWindowInSeconds to any value from 0 seconds to 300 seconds in increments of seconds.

For streams and Amazon SQS event sources, the default batching window is 0 seconds.

For Amazon MSK, Self-managed Apache Kafka, Amazon MQ, and DocumentDB event sources, the default batching window is 500 ms. Note that because you can only change

`MaximumBatchingWindowInSeconds` in increments of seconds, you cannot revert back to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

Related setting: For streams and Amazon SQS event sources, when you set `BatchSize` to a value greater than 10, you must set `MaximumBatchingWindowInSeconds` to at least 1.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

Required: No

MaximumRecordAgeInSeconds

(Kinesis and DynamoDB Streams only) Discard records older than the specified age. The default value is -1, which sets the maximum age to infinite. When the value is set to infinite, Lambda never discards old records.

Note

The minimum valid value for maximum record age is 60s. Although values less than 60 and greater than -1 fall within the parameter's absolute range, they are not allowed

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 604800.

Required: No

MaximumRetryAttempts

(Kinesis and DynamoDB Streams only) Discard records after the specified number of retries. The default value is -1, which sets the maximum number of retries to infinite. When `MaximumRetryAttempts` is infinite, Lambda retries failed records until the record expires in the event source.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

Required: No

ParallelizationFactor

(Kinesis and DynamoDB Streams only) The number of batches to process concurrently from each shard. The default value is 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10.

Required: No

Queues

(Amazon MQ) The name of the Amazon MQ broker destination queue to consume.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 1000.

Pattern: `[\s\S]*`

Required: No

ScalingConfig

(Amazon SQS only) The scaling configuration for the event source. For more information, see [Configuring maximum concurrency for Amazon SQS event sources](#).

Type: [ScalingConfig \(p. 1458\)](#) object

Required: No

SelfManagedEventSource

The self-managed Apache Kafka cluster for your event source.

Type: [SelfManagedEventSource \(p. 1459\)](#) object

Required: No

SelfManagedKafkaEventSourceConfig

Specific configuration settings for a self-managed Apache Kafka event source.

Type: [SelfManagedKafkaEventSourceConfig \(p. 1460\)](#) object

Required: No

SourceAccessConfigurations

An array of the authentication protocol, VPC components, or virtual host to secure and define your event source.

Type: Array of [SourceAccessConfiguration \(p. 1463\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

Required: No

StartingPosition

The position in a stream from which to start reading. Required for Amazon Kinesis, Amazon DynamoDB, and Amazon MSK stream sources. AT_TIMESTAMP is supported only for Amazon Kinesis streams and Amazon DocumentDB.

Type: String

Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

Required: No

StartingPositionTimestamp

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds.

Type: Timestamp

Required: No

State

The state of the event source mapping. It can be one of the following: Creating, Enabling, Enabled, Disabling, Disabled, Updating, or Deleting.

Type: String

Required: No

StateTransitionReason

Indicates whether a user or Lambda made the last change to the event source mapping.

Type: String

Required: No

Topics

The name of the Kafka topic.

Type: Array of strings

Array Members: Fixed number of 1 item.

Length Constraints: Minimum length of 1. Maximum length of 249.

Pattern: ^[^\n]([a-zA-Z0-9\\-_.]+)

Required: No

TumblingWindowInSeconds

(Kinesis and DynamoDB Streams only) The duration in seconds of a processing window for DynamoDB and Kinesis Streams event sources. A value of 0 seconds indicates no tumbling window.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

Required: No

UUID

The identifier of the event source mapping.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

FileSystemConfig

Details about the connection between a Lambda function and an [Amazon EFS file system](#).

Contents

Arn

The Amazon Resource Name (ARN) of the Amazon EFS access point that provides access to the file system.

Type: String

Length Constraints: Maximum length of 200.

Pattern: arn:aws[a-zA-Z-]*:elasticfilesystem:[a-z]{2}((-gov)|(-iso(b?)))?- [a-z]+-\d{1}:\d{12}:access-point/fsap-[a-f0-9]{17}

Required: Yes

LocalMountPath

The path where the function can access the file system, starting with /mnt/.

Type: String

Length Constraints: Maximum length of 160.

Pattern: ^/mnt/[a-zA-Z0-9-_\.]+\\$

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Filter

A structure within a `FilterCriteria` object that defines an event filtering pattern.

Contents

Pattern

A filter pattern. For more information on the syntax of a filter pattern, see [Filter rule syntax](#).

Type: String

Length Constraints: Minimum length of 0. Maximum length of 4096.

Pattern: `.*`

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

FilterCriteria

An object that contains the filters for an event source.

Contents

Filters

A list of filters.

Type: Array of [Filter \(p. 1425\)](#) objects

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

FunctionCode

The code for the Lambda function. You can either specify an object in Amazon S3, upload a .zip file archive deployment package directly, or specify the URI of a container image.

Contents

ImageUri

URI of a [container image](#) in the Amazon ECR registry.

Type: String

Required: No

S3Bucket

An Amazon S3 bucket in the same AWS Region as your function. The bucket can be in a different AWS account.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 63.

Pattern: ^[0-9A-Za-z\._\-_]*(?<!\.).\$

Required: No

S3Key

The Amazon S3 key of the deployment package.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Required: No

S3ObjectVersion

For versioned objects, the version of the deployment package object to use.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Required: No

ZipFile

The base64-encoded contents of the deployment package. AWS SDK and AWS CLI clients handle the encoding for you.

Type: Base64-encoded binary data object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)

- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

FunctionCodeLocation

Details about a function's deployment package.

Contents

ImageUri

URI of a container image in the Amazon ECR registry.

Type: String

Required: No

Location

A presigned URL that you can use to download the deployment package.

Type: String

Required: No

RepositoryType

The service that's hosting the file.

Type: String

Required: No

ResolvedImageUri

The resolved URI for the image.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

FunctionConfiguration

Details about a function's configuration.

Contents

Architectures

The instruction set architecture that the function supports. Architecture is a string array with one of the valid values. The default architecture value is x86_64.

Type: Array of strings

Array Members: Fixed number of 1 item.

Valid Values: x86_64 | arm64

Required: No

CodeSha256

The SHA256 hash of the function's deployment package.

Type: String

Required: No

CodeSize

The size of the function's deployment package, in bytes.

Type: Long

Required: No

DeadLetterConfig

The function's dead letter queue.

Type: [DeadLetterConfig \(p. 1412\)](#) object

Required: No

Description

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

Environment

The function's [environment variables](#). Omitted from AWS CloudTrail logs.

Type: [EnvironmentResponse \(p. 1417\)](#) object

Required: No

EphemeralStorage

The size of the function's /tmp directory in MB. The default value is 512, but it can be any whole number between 512 and 10,240 MB.

Type: [EphemeralStorage \(p. 1418\)](#) object

Required: No

FileSystemConfigs

Connection settings for an [Amazon EFS file system](#).

Type: Array of [FileSystemConfig \(p. 1424\)](#) objects

Array Members: Maximum number of 1 item.

Required: No

FunctionArn

The function's Amazon Resource Name (ARN).

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_\.]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: No

FunctionName

The name of the function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 170.

Pattern: `(arn:(aws[a-zA-Z-]*)?:lambda:)?([a-z]{2}(-gov)?-[a-z]+-\d{1}:)?\d{12}:?(function:)?([a-zA-Z0-9-_\.]+)(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: No

Handler

The function that Lambda calls to begin running your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

Required: No

ImageConfigResponse

The function's image configuration values.

Type: [ImageConfigResponse \(p. 1442\)](#) object

Required: No

KMSKeyArn

The AWS KMS key that's used to encrypt the function's [environment variables](#). When [Lambda SnapStart](#) is activated, this key is also used to encrypt the function's snapshot. This key is returned only if you've configured a customer managed key.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-z0-9-.]+:[.]*|()|)`

Required: No

LastModified

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Required: No

LastUpdateStatus

The status of the last update that was performed on the function. This is first set to Successful after function creation completes.

Type: String

Valid Values: Successful | Failed | InProgress

Required: No

LastUpdateStatusReason

The reason for the last update that was performed on the function.

Type: String

Required: No

LastUpdateStatusReasonCode

The reason code for the last update that was performed on the function.

Type: String

Valid Values: EniLimitExceeded | InsufficientRolePermissions | InvalidConfiguration | InternalError | SubnetOutOfRange | InvalidSubnet | InvalidSecurityGroup | ImageDeleted | ImageAccessDenied | InvalidImage | KMSKeyAccessDenied | KMSKeyNotFound | InvalidStateKMSKey | DisabledKMSKey | EFSIOError | EFSMountConnectivityError | EFSMountFailure | EFSMountTimeout | InvalidRuntime | InvalidZipFileException | FunctionError

Required: No

Layers

The function's [layers](#).

Type: Array of [Layer \(p. 1446\)](#) objects

Required: No

MasterArn

For Lambda@Edge functions, the ARN of the main function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: No

MemorySize

The amount of memory available to the function at runtime.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 10240.

Required: No

PackageType

The type of deployment package. Set to Image for container image and set Zip for .zip file archive.

Type: String

Valid Values: Zip | Image

Required: No

RevisionId

The latest updated revision of the function or alias.

Type: String

Required: No

Role

The function's execution role.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/[a-zA-Z_0-9+=,.@-_/.]+

Required: No

Runtime

The identifier of the function's [runtime](#). Runtime is required if the deployment package is a .zip file archive.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

Required: No

RuntimeVersionConfig

The ARN of the runtime and any errors that occurred.

Type: [RuntimeVersionConfig \(p. 1456\)](#) object

Required: No

SigningJobArn

The ARN of the signing job.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+:([a-z]{2}(-gov)?-[a-z]+-\d{1})?:(\d{12})?:(.*)

Required: No

SigningProfileVersionArn

The ARN of the signing profile version.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+:([a-z]{2}(-gov)?-[a-z]+-\d{1})?:(\d{12})?:(.*)

Required: No

SnapStart

Set ApplyOn to PublishedVersions to create a snapshot of the initialized execution environment when you publish a function version. For more information, see [Improving startup performance with Lambda SnapStart](#).

Type: [SnapStartResponse \(p. 1462\)](#) object

Required: No

State

The current state of the function. When the state is Inactive, you can reactivate the function by invoking it.

Type: String

Valid Values: Pending | Active | Inactive | Failed

Required: No

StateReason

The reason for the function's current state.

Type: String

Required: No

StateReasonCode

The reason code for the function's current state. When the code is Creating, you can't invoke or modify the function.

Type: String

Valid Values: Idle | Creating | Restoring | EniLimitExceeded | InsufficientRolePermissions | InvalidConfiguration | InternalError | SubnetOutofIPAddresses | InvalidSubnet | InvalidSecurityGroup | ImageDeleted | ImageAccessDenied | InvalidImage | KMSKeyAccessDenied | KMSKeyNotFound | InvalidStateKMSKey | DisabledKMSKey | EFSIOError | EFSSMountConnectivityError | EFSSMountFailure | EFSSMountTimeout | InvalidRuntime | InvalidZipFileException | FunctionError

Required: No

Timeout

The amount of time in seconds that Lambda allows a function to run before stopping it.

Type: Integer

Valid Range: Minimum value of 1.

Required: No

TracingConfig

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 1466\)](#) object

Required: No

Version

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST | [0-9]+)

Required: No

VpcConfig

The function's networking configuration.

Type: [VpcConfigResponse \(p. 1468\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

FunctionEventInvokeConfig

Contents

DestinationConfig

A destination for events after they have been sent to a function for processing.

Destinations

- **Function** - The Amazon Resource Name (ARN) of a Lambda function.
- **Queue** - The ARN of a standard SQS queue.
- **Topic** - The ARN of a standard SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: [DestinationConfig \(p. 1413\)](#) object

Required: No

FunctionArn

The Amazon Resource Name (ARN) of the function.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}(-gov)?-[a-z]+\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$\$LATEST|[a-zA-Z0-9-_]+))?`

Required: No

LastModified

The date and time that the configuration was last updated, in Unix time seconds.

Type: Timestamp

Required: No

MaximumEventAgeInSeconds

The maximum age of a request that Lambda sends to a function for processing.

Type: Integer

Valid Range: Minimum value of 60. Maximum value of 21600.

Required: No

MaximumRetryAttempts

The maximum number of times to retry when the function returns an error.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 2.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

FunctionUrlConfig

Details about a Lambda function URL.

Contents

AuthType

The type of authentication that your function URL uses. Set to AWS_IAM if you want to restrict access to authenticated users only. Set to NONE if you want to bypass IAM authentication to create a public endpoint. For more information, see [Security and auth model for Lambda function URLs](#).

Type: String

Valid Values: NONE | AWS_IAM

Required: Yes

Cors

The [cross-origin resource sharing \(CORS\)](#) settings for your function URL.

Type: [Cors \(p. 1410\)](#) object

Required: No

CreationTime

When the function URL was created, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Required: Yes

FunctionArn

The Amazon Resource Name (ARN) of your function.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\\$\\$LATEST|[a-zA-Z0-9-_]+))?

Required: Yes

FunctionUrl

The HTTP URL endpoint for your function.

Type: String

Length Constraints: Minimum length of 40. Maximum length of 100.

Required: Yes

InvokeMode

Use one of the following options:

- BUFFERED – This is the default option. Lambda invokes your function using the Invoke API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.
- RESPONSE_STREAM – Your function streams payload results as they become available. Lambda invokes your function using the InvokeWithResponseStream API operation. The maximum response payload size is 20 MB, however, you can [request a quota increase](#).

Type: String

Valid Values: BUFFERED | RESPONSE_STREAM

Required: No

LastModifiedTime

When the function URL configuration was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Required: Yes

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

ImageConfig

Configuration values that override the container image Dockerfile settings. For more information, see [Container image settings](#).

Contents

Command

Specifies parameters that you want to pass in with ENTRYPPOINT.

Type: Array of strings

Array Members: Maximum number of 1500 items.

Required: No

EntryPoint

Specifies the entry point to their application, which is typically the location of the runtime executable.

Type: Array of strings

Array Members: Maximum number of 1500 items.

Required: No

WorkingDirectory

Specifies the working directory.

Type: String

Length Constraints: Maximum length of 1000.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

ImageConfigError

Error response to GetFunctionConfiguration.

Contents

ErrorCode

Error code.

Type: String

Required: No

Message

Error message.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

ImageConfigResponse

Response to a GetFunctionConfiguration request.

Contents

Error

Error response to GetFunctionConfiguration.

Type: [ImageConfigError \(p. 1441\)](#) object

Required: No

ImageConfig

Configuration values that override the container image Dockerfile.

Type: [ImageConfig \(p. 1440\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

InvokeResponseStreamUpdate

A chunk of the streamed response payload.

Contents

Payload

Data returned by your Lambda function.

Type: Base64-encoded binary data object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

InvokeWithResponseStreamCompleteEvent

A response confirming that the event stream is complete.

Contents

ErrorCode

An error code.

Type: String

Required: No

ErrorDetails

The details of any returned error.

Type: String

Required: No

LogResult

The last 4 KB of the execution log, which is base64-encoded.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

InvokeWithResponseStreamResponseEvent

An object that includes a chunk of the response payload. When the stream has ended, Lambda includes a `InvokeComplete` object.

Contents

InvokeComplete

An object that's returned when the stream has ended and all the payload chunks have been returned.

Type: [InvokeWithResponseStreamCompleteEvent \(p. 1444\)](#) object

Required: No

PayloadChunk

A chunk of the streamed response payload.

Type: [InvokeResponseStreamUpdate \(p. 1443\)](#) object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Layer

An [AWS Lambda layer](#).

Contents

Arn

The Amazon Resource Name (ARN) of the function layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+

Required: No

CodeSize

The size of the layer archive in bytes.

Type: Long

Required: No

SigningJobArn

The Amazon Resource Name (ARN) of a signing job.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*)([a-zA-Z0-9\-\-])+([a-z]{2}(-gov)?-[a-z]+\d{1})?:(\d{12})?:(.**)

Required: No

SigningProfileVersionArn

The Amazon Resource Name (ARN) for a signing profile version.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*)([a-zA-Z0-9\-\-])+([a-z]{2}(-gov)?-[a-z]+\d{1})?:(\d{12})?:(.**)

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

LayersListItem

Details about an [AWS Lambda layer](#).

Contents

LatestMatchingVersion

The newest version of the layer.

Type: [LayerVersionsListItem \(p. 1450\)](#) object

Required: No

LayerArn

The Amazon Resource Name (ARN) of the function layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+`

Required: No

LayerName

The name of the layer.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: `(arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+)|[a-zA-Z0-9-_]+`

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

LayerVersionContentInput

A ZIP archive that contains the contents of an [AWS Lambda layer](#). You can specify either an Amazon S3 location, or upload a layer archive directly.

Contents

S3Bucket

The Amazon S3 bucket of the layer archive.

Type: String

Length Constraints: Minimum length of 3. Maximum length of 63.

Pattern: ^[0-9A-Za-z\._\-_]*(?<!\.).\$

Required: No

S3Key

The Amazon S3 key of the layer archive.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Required: No

S3ObjectVersion

For versioned objects, the version of the layer archive object to use.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Required: No

ZipFile

The base64-encoded contents of the layer archive. AWS SDK and AWS CLI clients handle the encoding for you.

Type: Base64-encoded binary data object

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

LayerVersionContentOutput

Details about a version of an [AWS Lambda layer](#).

Contents

CodeSha256

The SHA-256 hash of the layer archive.

Type: String

Required: No

CodeSize

The size of the layer archive in bytes.

Type: Long

Required: No

Location

A link to the layer archive in Amazon S3 that is valid for 10 minutes.

Type: String

Required: No

SigningJobArn

The Amazon Resource Name (ARN) of a signing job.

Type: String

Required: No

SigningProfileVersionArn

The Amazon Resource Name (ARN) for a signing profile version.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

LayerVersionsListItem

Details about a version of an [AWS Lambda layer](#).

Contents

CompatibleArchitectures

A list of compatible [instruction set architectures](#).

Type: Array of strings

Array Members: Maximum number of 2 items.

Valid Values: x86_64 | arm64

Required: No

CompatibleRuntimes

The layer's compatible runtimes.

The following list includes deprecated runtimes. For more information, see [Runtime deprecation policy](#).

Type: Array of strings

Array Members: Maximum number of 15 items.

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | nodejs16.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | python3.9 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | dotnet6 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2 | nodejs18.x | python3.10 | java17

Required: No

CreatedDate

The date that the version was created, in ISO 8601 format. For example, 2018-11-27T15:10:45.123+0000.

Type: String

Required: No

Description

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

LayerVersionArn

The ARN of the layer version.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 140.

Pattern: arn:[a-zA-Z0-9-]+:lambda:[a-zA-Z0-9-]+:\d{12}:layer:[a-zA-Z0-9-_]+:[0-9]+

Required: No

LicenseInfo

The layer's open-source license.

Type: String

Length Constraints: Maximum length of 512.

Required: No

Version

The version number.

Type: Long

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

OnFailure

A destination for events that failed processing.

Contents

Destination

The Amazon Resource Name (ARN) of the destination resource.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 350.

Pattern: ^\$|arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-.])+:([a-z]{2}(-gov)?-[a-z]+\-\d{1})?:(\d{12})?:(.*)

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

OnSuccess

A destination for events that were processed successfully.

Contents

Destination

The Amazon Resource Name (ARN) of the destination resource.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 350.

Pattern: ^\$|arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-.])+:([a-z]{2}(-gov)?-[a-z]+\-\d{1})?:(\d{12})?:(.*)

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

ProvisionedConcurrencyConfigListItem

Details about the provisioned concurrency configuration for a function alias or version.

Contents

AllocatedProvisionedConcurrentExecutions

The amount of provisioned concurrency allocated. When a weighted alias is used during linear and canary deployments, this value fluctuates depending on the amount of concurrency that is provisioned for the function versions.

Type: Integer

Valid Range: Minimum value of 0.

Required: No

AvailableProvisionedConcurrentExecutions

The amount of provisioned concurrency available.

Type: Integer

Valid Range: Minimum value of 0.

Required: No

FunctionArn

The Amazon Resource Name (ARN) of the alias or version.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:lambda:[a-z]{2}(-gov)?-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(:(\$LATEST|[a-zA-Z0-9-_]+))?`

Required: No

LastModified

The date and time that a user last updated the configuration, in [ISO 8601 format](#).

Type: String

Required: No

RequestedProvisionedConcurrentExecutions

The amount of provisioned concurrency requested.

Type: Integer

Valid Range: Minimum value of 1.

Required: No

Status

The status of the allocation process.

Type: String

Valid Values: IN_PROGRESS | READY | FAILED

Required: No

StatusReason

For failed allocations, the reason that provisioned concurrency could not be allocated.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

RuntimeVersionConfig

The ARN of the runtime and any errors that occurred.

Contents

Error

Error response when Lambda is unable to retrieve the runtime version for a function.

Type: [RuntimeVersionError \(p. 1457\)](#) object

Required: No

RuntimeVersionArn

The ARN of the runtime version you want the function to use.

Type: String

Length Constraints: Minimum length of 26. Maximum length of 2048.

Pattern: ^arn:(aws[a-zA-Z-]*):lambda:[a-z]{2}((-gov)|(-iso(b?)))?-([a-z]+-\d{1}):runtime:.+\$

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

RuntimeVersionError

Any error returned when the runtime version information for the function could not be retrieved.

Contents

ErrorCode

The error code.

Type: String

Required: No

Message

The error message.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

ScalingConfig

(Amazon SQS only) The scaling configuration for the event source. To remove the configuration, pass an empty value.

Contents

MaximumConcurrency

Limits the number of concurrent instances that the Amazon SQS event source can invoke.

Type: Integer

Valid Range: Minimum value of 2. Maximum value of 1000.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

SelfManagedEventSource

The self-managed Apache Kafka cluster for your event source.

Contents

Endpoints

The list of bootstrap servers for your Kafka brokers in the following format:

"KAFKA_BOOTSTRAP_SERVERS": ["abc.xyz.com:xxxx", "abc2.xyz.com:xxxx"].

Type: String to array of strings map

Map Entries: Maximum number of 2 items.

Valid Keys: KAFKA_BOOTSTRAP_SERVERS

Array Members: Minimum number of 1 item. Maximum number of 10 items.

Length Constraints: Minimum length of 1. Maximum length of 300.

Pattern: ^(([a-zA-Z0-9] | [a-zA-Z0-9][a-zA-Z0-9\-\-]*[a-zA-Z0-9])\.)*([A-Za-z0-9] | [A-Za-z0-9][A-Za-z0-9\-\-]*[A-Za-z0-9]):[0-9]{1,5}

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

SelfManagedKafkaEventSourceConfig

Specific configuration settings for a self-managed Apache Kafka event source.

Contents

ConsumerGroupId

The identifier for the Kafka consumer group to join. The consumer group ID must be unique among all your Kafka event sources. After creating a Kafka event source mapping with the consumer group ID specified, you cannot update this value. For more information, see [Customizable consumer group ID](#).

Type: String

Length Constraints: Minimum length of 1. Maximum length of 200.

Pattern: [a-zA-Z0-9-\\/*:_+=.\\@-]*

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

SnapStart

The function's Lambda SnapStart setting. Set ApplyOn to PublishedVersions to create a snapshot of the initialized execution environment when you publish a function version.

SnapStart is supported with the java11 runtime. For more information, see [Improving startup performance with Lambda SnapStart](#).

Contents

ApplyOn

Set to PublishedVersions to create a snapshot of the initialized execution environment when you publish a function version.

Type: String

Valid Values: PublishedVersions | None

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

SnapStartResponse

The function's [SnapStart](#) setting.

Contents

ApplyOn

When set to PublishedVersions, Lambda creates a snapshot of the execution environment when you publish a function version.

Type: String

Valid Values: PublishedVersions | None

Required: No

OptimizationStatus

When you provide a [qualified Amazon Resource Name \(ARN\)](#), this response element indicates whether SnapStart is activated for the specified function version.

Type: String

Valid Values: On | Off

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

SourceAccessConfiguration

To secure and define access to your event source, you can specify the authentication protocol, VPC components, or virtual host.

Contents

Type

The type of authentication protocol, VPC components, or virtual host for your event source. For example: "Type": "SASL_SCRAM_512_AUTH".

- BASIC_AUTH – (Amazon MQ) The AWS Secrets Manager secret that stores your broker credentials.
- BASIC_AUTH – (Self-managed Apache Kafka) The Secrets Manager ARN of your secret key used for SASL/PLAIN authentication of your Apache Kafka brokers.
- VPC_SUBNET – (Self-managed Apache Kafka) The subnets associated with your VPC. Lambda connects to these subnets to fetch data from your self-managed Apache Kafka cluster.
- VPC_SECURITY_GROUP – (Self-managed Apache Kafka) The VPC security group used to manage access to your self-managed Apache Kafka brokers.
- SASL_SCRAM_256_AUTH – (Self-managed Apache Kafka) The Secrets Manager ARN of your secret key used for SASL SCRAM-256 authentication of your self-managed Apache Kafka brokers.
- SASL_SCRAM_512_AUTH – (Amazon MSK, Self-managed Apache Kafka) The Secrets Manager ARN of your secret key used for SASL SCRAM-512 authentication of your self-managed Apache Kafka brokers.
- VIRTUAL_HOST -- (RabbitMQ) The name of the virtual host in your RabbitMQ broker. Lambda uses this RabbitMQ host as the event source. This property cannot be specified in an UpdateEventSourceMapping API call.
- CLIENT_CERTIFICATE_TLS_AUTH – (Amazon MSK, self-managed Apache Kafka) The Secrets Manager ARN of your secret key containing the certificate chain (X.509 PEM), private key (PKCS#8 PEM), and private key password (optional) used for mutual TLS authentication of your MSK/ Apache Kafka brokers.
- SERVER_ROOT_CA_CERTIFICATE – (Self-managed Apache Kafka) The Secrets Manager ARN of your secret key containing the root CA certificate (X.509 PEM) used for TLS encryption of your Apache Kafka brokers.

Type: String

Valid Values: BASIC_AUTH | VPC_SUBNET | VPC_SECURITY_GROUP | SASL_SCRAM_512_AUTH | SASL_SCRAM_256_AUTH | VIRTUAL_HOST | CLIENT_CERTIFICATE_TLS_AUTH | SERVER_ROOT_CA_CERTIFICATE

Required: No

URI

The value for your chosen configuration in Type. For example: "URI": "arn:aws:secretsmanager:us-east-1:01234567890:secret:MyBrokerSecretName".

Type: String

Length Constraints: Minimum length of 1. Maximum length of 200.

Pattern: [a-zA-Z0-9-\/*:_+=.@-]*

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

TracingConfig

The function's [AWS X-Ray](#) tracing configuration. To sample and record incoming requests, set Mode to Active.

Contents

Mode

The tracing mode.

Type: String

Valid Values: Active | PassThrough

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

TracingConfigResponse

The function's AWS X-Ray tracing configuration.

Contents

Mode

The tracing mode.

Type: String

Valid Values: Active | PassThrough

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

VpcConfig

The VPC security groups and subnets that are attached to a Lambda function. For more information, see [Configuring a Lambda function to access resources in a VPC](#).

Contents

SecurityGroupIds

A list of VPC security group IDs.

Type: Array of strings

Array Members: Maximum number of 5 items.

Required: No

SubnetIds

A list of VPC subnet IDs.

Type: Array of strings

Array Members: Maximum number of 16 items.

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

VpcConfigResponse

The VPC security groups and subnets that are attached to a Lambda function.

Contents

SecurityGroupIds

A list of VPC security group IDs.

Type: Array of strings

Array Members: Maximum number of 5 items.

Required: No

SubnetIds

A list of VPC subnet IDs.

Type: Array of strings

Array Members: Maximum number of 16 items.

Required: No

VpcId

The ID of the VPC.

Type: String

Required: No

See Also

For more information about using this API in one of the language-specific AWS SDKs, see the following:

- [AWS SDK for C++](#)
- [AWS SDK for Go](#)
- [AWS SDK for Java V2](#)
- [AWS SDK for Ruby V3](#)

Certificate errors when using an SDK

Because AWS SDKs use the CA certificates from your computer, changes to the certificates on the AWS servers can cause connection failures when you attempt to use an SDK. You can prevent these failures by keeping your computer's CA certificates and operating system up-to-date. If you encounter this issue in a corporate environment and do not manage your own computer, you might need to ask an administrator to assist with the update process. The following list shows minimum operating system and Java versions:

- Microsoft Windows versions that have updates from January 2005 or later installed contain at least one of the required CAs in their trust list.
- Mac OS X 10.4 with Java for Mac OS X 10.4 Release 5 (February 2007), Mac OS X 10.5 (October 2007), and later versions contain at least one of the required CAs in their trust list.

- Red Hat Enterprise Linux 5 (March 2007), 6, and 7 and CentOS 5, 6, and 7 all contain at least one of the required CAs in their default trusted CA list.
- Java 1.4.2_12 (May 2006), 5 Update 2 (March 2005), and all later versions, including Java 6 (December 2006), 7, and 8, contain at least one of the required CAs in their default trusted CA list.

When accessing the AWS Lambda management console or AWS Lambda API endpoints, whether through browsers or programmatically, you will need to ensure your client machines support any of the following CAs:

- Amazon Root CA 1
- Starfield Services Root Certificate Authority - G2
- Starfield Class 2 Certification Authority

Root certificates from the first two authorities are available from [Amazon trust services](#), but keeping your computer up-to-date is the more straightforward solution. To learn more about ACM-provided certificates, see [AWS Certificate Manager FAQs](#).

Common Errors

This section lists the errors common to the API actions of all AWS services. For errors specific to an API action for this service, see the topic for that API action.

AccessDeniedException

You do not have sufficient access to perform this action.

HTTP Status Code: 400

IncompleteSignature

The request signature does not conform to AWS standards.

HTTP Status Code: 400

InternalFailure

The request processing has failed because of an unknown error, exception or failure.

HTTP Status Code: 500

InvalidAction

The action or operation requested is invalid. Verify that the action is typed correctly.

HTTP Status Code: 400

InvalidClientTokenId

The X.509 certificate or AWS access key ID provided does not exist in our records.

HTTP Status Code: 403

InvalidParameterCombination

Parameters that must not be used together were used together.

HTTP Status Code: 400

InvalidParameterValue

An invalid or out-of-range value was supplied for the input parameter.

HTTP Status Code: 400

InvalidQueryParameter

The AWS query string is malformed or does not adhere to AWS standards.

HTTP Status Code: 400

MalformedQueryString

The query string contains a syntax error.

HTTP Status Code: 404

MissingAction

The request is missing an action or a required parameter.

HTTP Status Code: 400

MissingAuthenticationToken

The request must contain either a valid (registered) AWS access key ID or X.509 certificate.

HTTP Status Code: 403

MissingParameter

A required parameter for the specified action is not supplied.

HTTP Status Code: 400

NotAuthorized

You do not have permission to perform this action.

HTTP Status Code: 400

OptInRequired

The AWS access key ID needs a subscription for the service.

HTTP Status Code: 403

RequestExpired

The request reached the service more than 15 minutes after the date stamp on the request or more than 15 minutes after the request expiration date (such as for pre-signed URLs), or the date stamp on the request is more than 15 minutes in the future.

HTTP Status Code: 400

ServiceUnavailable

The request has failed due to a temporary failure of the server.

HTTP Status Code: 503

ThrottlingException

The request was denied due to request throttling.

HTTP Status Code: 400

ValidationException

The input fails to satisfy the constraints specified by an AWS service.

HTTP Status Code: 400

Common Parameters

The following list contains the parameters that all actions use for signing Signature Version 4 requests with a query string. Any action-specific parameters are listed in the topic for that action. For more information about Signature Version 4, see [Signing AWS API requests](#) in the *IAM User Guide*.

Action

The action to be performed.

Type: string

Required: Yes

Version

The API version that the request is written for, expressed in the format YYYY-MM-DD.

Type: string

Required: Yes

X-Amz-Algorithm

The hash algorithm that you used to create the request signature.

Condition: Specify this parameter when you include authentication information in a query string instead of in the HTTP authorization header.

Type: string

Valid Values: AWS4-HMAC-SHA256

Required: Conditional

X-Amz-Credential

The credential scope value, which is a string that includes your access key, the date, the region you are targeting, the service you are requesting, and a termination string ("aws4_request"). The value is expressed in the following format: *access_key/YYYYMMDD/region/service/aws4_request*.

For more information, see [Create a signed AWS API request](#) in the *IAM User Guide*.

Condition: Specify this parameter when you include authentication information in a query string instead of in the HTTP authorization header.

Type: string

Required: Conditional

X-Amz-Date

The date that is used to create the signature. The format must be ISO 8601 basic format (YYYYMMDD'T'HHMMSS'Z'). For example, the following date time is a valid X-Amz-Date value: 20120325T120000Z.

Condition: X-Amz-Date is optional for all requests; it can be used to override the date used for signing requests. If the Date header is specified in the ISO 8601 basic format, X-Amz-Date is not required. When X-Amz-Date is used, it always overrides the value of the Date header. For more information, see [Elements of an AWS API request signature](#) in the *IAM User Guide*.

Type: string

Required: Conditional

X-Amz-Security-Token

The temporary security token that was obtained through a call to AWS Security Token Service (AWS STS). For a list of services that support temporary security credentials from AWS STS, see [AWS services that work with IAM](#) in the *IAM User Guide*.

Condition: If you're using temporary security credentials from AWS STS, you must include the security token.

Type: string

Required: Conditional

X-Amz-Signature

Specifies the hex-encoded signature that was calculated from the string to sign and the derived signing key.

Condition: Specify this parameter when you include authentication information in a query string instead of in the HTTP authorization header.

Type: string

Required: Conditional

X-Amz-SignedHeaders

Specifies all the HTTP headers that were included as part of the canonical request. For more information about specifying signed headers, see [Create a signed AWS API request](#) in the *IAM User Guide*.

Condition: Specify this parameter when you include authentication information in a query string instead of in the HTTP authorization header.

Type: string

Required: Conditional

AWS Lambda releases

The following table describes the important changes to the *AWS Lambda Developer Guide* since May 2018. For notification about updates to this documentation, subscribe to the [RSS feed](#).

Change	Description	Date
Response streaming	Lambda now supports streaming responses from functions. For more information, see Configuring a Lambda function to stream responses .	April 6, 2023
Asynchronous invocation metrics	Lambda releases asynchronous invocation metrics. For more information, see Asynchronous invocation metrics .	February 9, 2023
Runtime version controls	Lambda releases new runtime versions that include security updates, bug fixes, and new features. You can now control when your functions get updated to the new runtime versions. For more information, see Lambda runtime updates .	January 23, 2023
Lambda SnapStart	Use Lambda SnapStart to reduce startup time for Java functions without provisioning additional resources or implementing complex performance optimizations. For more information, see Improving startup performance with Lambda SnapStart .	November 28, 2022
Node.js 18 runtime	Lambda now supports a new runtime for Node.js 18. Node.js 18 uses Amazon Linux 2. For details, see Building Lambda functions with Node.js .	November 18, 2022
lambda:SourceFunctionArn condition key	For an AWS resource, the <code>lambda:SourceFunctionArn</code> condition key filters access to the resource by the ARN of a Lambda function. For details, see Working with Lambda execution environment credentials .	July 1, 2022
Node.js 16 runtime	Lambda now supports a new runtime for Node.js 16. Node.js 16 uses Amazon Linux 2. For details, see Building Lambda functions with Node.js .	May 11, 2022

<u>Lambda function URLs</u>	Lambda now supports function URLs, which are dedicated HTTP(S) endpoints for Lambda functions. For details, see <u>Lambda function URLs</u> .	April 6, 2022
<u>Shared test events in the AWS Lambda console</u>	Lambda now supports sharing test events with other users in the same AWS account. For details, see <u>Testing Lambda functions in the console</u> .	March 16, 2022
<u>PrincipalOrgId in resource-based policies</u>	Lambda now supports granting permissions to an organization in AWS Organizations. For details, see <u>Using resource-based policies for AWS Lambda</u> .	March 11, 2022
<u>.NET 6 runtime</u>	Lambda now supports a new runtime for .NET 6. For details, see <u>Lambda runtimes</u> .	February 23, 2022
<u>Event filtering for Kinesis, DynamoDB, and Amazon SQS event sources</u>	Lambda now supports event filtering for Kinesis, DynamoDB, and Amazon SQS event sources. For details, see <u>Lambda event filtering</u> .	November 24, 2021
<u>mTLS authentication for Amazon MSK and self-managed Apache Kafka event sources</u>	Lambda now supports mTLS authentication for Amazon MSK and self-managed Apache Kafka event sources. For details, see <u>Using Lambda with Amazon MSK</u> .	November 19, 2021
<u>Lambda on Graviton2</u>	Lambda now supports Graviton2 for functions using arm64 architecture. For details, see <u>Lambda instruction set architectures</u> .	September 29, 2021
<u>Python 3.9 runtime</u>	Lambda now supports a new runtime for Python 3.9. For details, see <u>Lambda runtimes</u> .	August 16, 2021
<u>New runtime versions for Node.js, Python, and Java</u>	New runtime versions are available for Node.js, Python, and Java. For details, see <u>Lambda runtimes</u> .	July 21, 2021

Support for RabbitMQ as an event source on Lambda	Lambda now supports Amazon MQ for RabbitMQ as an event source. Amazon MQ is a managed message broker service for Apache ActiveMQ and RabbitMQ that makes it easy to set up and operate message brokers in the cloud. For details, see Using Lambda with Amazon MQ .	July 7, 2021
SASL/PLAIN authentication for self-managed Kafka on Lambda	SASL/PLAIN is now a supported authentication mechanism for self-managed Kafka event sources on Lambda. Customers already using SASL/PLAIN on their self-managed Kafka cluster can now easily use Lambda to build consumer applications without having to modify the way they authenticate. For details, see Using Lambda with self-managed Apache Kafka .	June 29, 2021
Lambda Extensions API	General availability for Lambda extensions. Use extensions to augment your Lambda functions. You can use extensions provided by Lambda Partners, or you can create your own Lambda extensions. For details, see Lambda Extensions API .	May 24, 2021
New Lambda console experience (p. 1473)	The Lambda console has been redesigned to improve performance and consistency.	March 2, 2021
Node.js 14 runtime	Lambda now supports a new runtime for Node.js 14. Node.js 14 uses Amazon Linux 2. For details, see Building Lambda functions with Node.js .	January 27, 2021
Lambda container images	Lambda now supports functions defined as container images. You can combine the flexibility of container tooling with the agility and operational simplicity of Lambda to build applications. For details, see Using container images with Lambda .	December 1, 2020

<u>Code signing for Lambda functions</u>	Lambda now supports code signing. Administrators can configure Lambda functions to accept only signed code on deployment. Lambda checks the signatures to ensure that the code is not altered or tampered. Additionally, Lambda ensures that the code is signed by trusted developers before accepting the deployment. For details, see Configuring code signing for Lambda .	November 23, 2020
<u>Preview: Lambda Runtime Logs API</u>	Lambda now supports the Runtime Logs API. Lambda extensions can use the Logs API to subscribe to log streams in the execution environment. For details, see Lambda Runtime Logs API .	November 12, 2020
<u>New event source for Amazon MQ</u>	Lambda now supports Amazon MQ as an event source. Use a Lambda function to process records from your Amazon MQ message broker. For details, see Using Lambda with Amazon MQ .	November 5, 2020
<u>Preview: Lambda Extensions API</u>	Use Lambda extensions to augment your Lambda functions. You can use extensions provided by Lambda Partners, or you can create your own Lambda extensions. For details, see Lambda Extensions API .	October 8, 2020
<u>Support for Java 8 and custom runtimes on AL2</u>	Lambda now supports Java 8 and custom runtimes on Amazon Linux 2. For details, see Lambda runtimes .	August 12, 2020
<u>New event source for Amazon Managed Streaming for Apache Kafka</u>	Lambda now supports Amazon MSK as an event source. Use a Lambda function with Amazon MSK to process records in a Kafka topic. For details, see Using Lambda with Amazon MSK .	August 11, 2020

<u>IAM condition keys for Amazon VPC settings</u>	You can now use Lambda-specific condition keys for VPC settings. For example, you can require that all functions in your organization are connected to a VPC. You can also specify the subnets and security groups that the function's users can and can't use. For details, see Configuring VPC for IAM functions .	August 10, 2020
<u>Concurrency settings for Kinesis HTTP/2 stream consumers</u>	You can now use the following concurrency settings for Kinesis consumers with enhanced fan-out (HTTP/2 streams): ParallelizationFactor, MaximumRetryAttempts, MaximumRecordAgeInSeconds, DestinationConfig, and BisectBatchOnFunctionError. For details, see Using AWS Lambda with Amazon Kinesis .	July 7, 2020
<u>Batch window for Kinesis HTTP/2 stream consumers</u>	You can now configure a batch window (MaximumBatchingWindowInSeconds) for HTTP/2 streams. Lambda reads records from the stream until it has gathered a full batch, or until the batch window expires. For details, see Using AWS Lambda with Amazon Kinesis .	June 18, 2020
<u>Support for Amazon EFS file systems</u>	You can now connect an Amazon EFS file system to your Lambda functions for shared network file access. For details, see Configuring file system access for Lambda functions .	June 16, 2020
<u>AWS CDK sample applications in the Lambda console</u>	The Lambda console now includes sample applications that use the AWS Cloud Development Kit (AWS CDK) for TypeScript. The AWS CDK is a framework that enables you to define your application resources in TypeScript, Python, Java, or .NET. For a tutorial on creating applications, see Creating an application with continuous delivery in the Lambda console .	June 1, 2020

<u>Support for .NET Core 3.1.0 runtime in AWS Lambda</u>	AWS Lambda now supports the .NET Core 3.1.0 runtime. For details, see .NET Core CLI .	March 31, 2020
<u>Support for API Gateway HTTP APIs</u>	Updated and expanded documentation for using Lambda with API Gateway, including support for HTTP APIs. Added a sample application that creates an API and function with AWS CloudFormation. For details, see Using Lambda with Amazon API Gateway .	March 23, 2020
<u>Ruby 2.7</u>	A new runtime is available for Ruby 2.7, ruby2.7, which is the first Ruby runtime to use Amazon Linux 2. For details, see Building Lambda functions with Ruby .	February 19, 2020
<u>Concurrency metrics</u>	Lambda now reports the ConcurrentExecutions metric for all functions, aliases, and versions. You can view a graph for this metric on the monitoring page for your function. Previously, ConcurrentExecutions was only reported at the account level and for functions that use reserved concurrency. For details, see AWS Lambda function metrics .	February 18, 2020

Update to function states	Function states are now enforced for all functions by default. When you connect a function to a VPC, Lambda creates shared elastic network interfaces. This enables your function to scale up without creating additional network interfaces. During this time, you can't perform additional operations on the function, including updating its configuration and publishing versions. In some cases, invocation is also impacted. Details about a function's current state are available from the Lambda API.	January 24, 2020
Updates to function configuration API output	This update is being released in phases. For details, see Updated Lambda states lifecycle for VPC networking on the AWS Compute Blog. For more information about states, see AWS Lambda function states .	January 20, 2020
Provisioned concurrency	Added reason codes to StateReasonCode (<code>InvalidSubnet</code> , <code>InvalidSecurityGroup</code>) and <code>LastUpdateStatusReasonCode</code> (<code>SubnetOutOfIPAddresses</code> , <code>InvalidSubnet</code> , <code>InvalidSecurityGroup</code>) for functions that connect to a VPC. For more information about states, see AWS Lambda function states .	December 3, 2019

Create a database proxy	You can now use the Lambda console to create a database proxy for a Lambda function. A database proxy enables a function to reach high concurrency levels without exhausting database connections. For details, see Configuring database access for a Lambda function .	December 3, 2019
Percentiles support for the duration metric	You can now filter the duration metric based on percentiles. For details, see AWS Lambda metrics .	November 26, 2019
Increased concurrency for stream event sources	A new option for DynamoDB stream and Kinesis stream event source mappings enables you to process more than one batch at a time from each shard. When you increase the number of concurrent batches per shard, your function's concurrency can be up to 10 times the number of shards in your stream. For details, see Lambda event source mapping .	November 25, 2019
Function states	When you create or update a function, it enters a pending state while Lambda provisions resources to support it. If you connect your function to a VPC, Lambda can create a shared elastic network interface right away, instead of creating network interfaces when your function is invoked. This results in better performance for VPC-connected functions, but might require an update to your automation. For details, see AWS Lambda function states .	November 25, 2019
Error handling options for asynchronous invocation	New configuration options are available for asynchronous invocation. You can configure Lambda to limit retries and set a maximum event age. For details, see Configuring error handling for asynchronous invocation .	November 25, 2019

Error handling for stream event sources	New configuration options are available for event source mappings that read from streams. You can configure DynamoDB stream and Kinesis stream event source mappings to limit retries and set a maximum record age. When errors occur, you can configure the event source mapping to split batches before retrying, and to send invocation records for failed batches to a queue or topic. For details, see Lambda event source mapping .	November 25, 2019
Destinations for asynchronous invocation	You can now configure Lambda to send records of asynchronous invocations to another service. Invocation records contain details about the event, context, and function response. You can send invocation records to an SQS queue, SNS topic, Lambda function, or EventBridge event bus. For details, see Configuring destinations for asynchronous invocation .	November 25, 2019
New runtimes for Node.js, Python, and Java	New runtimes are available for Node.js 12, Python 3.8, and Java 11. For details, see Lambda runtimes .	November 18, 2019
FIFO queue support for Amazon SQS event sources	You can now create an event source mapping that reads from a first-in, first-out (FIFO) queue. Previously, only standard queues were supported. For details, see Using Lambda with Amazon SQS .	November 18, 2019
Create applications in the Lambda console	Application creation in the Lambda console is now generally available. For instructions, see Creating an application with continuous delivery in the Lambda console .	October 31, 2019

[Create applications in the Lambda console \(beta\)](#)

You can now create a Lambda application with an integrated continuous delivery pipeline in the Lambda console. The console provides sample applications that you can use as a starting point for your own project. Choose between AWS CodeCommit and GitHub for source control. Each time you push changes to your repository, the included pipeline builds and deploys them automatically. For instructions, see [Creating an application with continuous delivery in the Lambda console](#).

October 3, 2019

[Performance improvements for VPC-connected functions](#)

Lambda now uses a new type of elastic network interface that is shared by all functions in a virtual private cloud (VPC) subnet. When you connect a function to a VPC, Lambda creates a network interface for each combination of security group and subnet that you choose. When the shared network interfaces are available, the function no longer needs to create additional network interfaces as it scales up. This dramatically improves startup times. For details, see [Configuring a Lambda function to access resources in a VPC](#).

September 3, 2019

[Stream batch settings](#)

You can now configure a batch window for [Amazon DynamoDB](#) and [Amazon Kinesis](#) event source mappings. Configure a batch window of up to five minutes to buffer incoming records until a full batch is available. This reduces the number of times that your function is invoked when the stream is less active.

August 29, 2019

[CloudWatch Logs Insights integration](#)

The monitoring page in the Lambda console now includes reports from Amazon CloudWatch Logs Insights. For details, see [Monitoring functions in the AWS Lambda console](#).

June 18, 2019

Amazon Linux 2018.03	The Lambda execution environment is being updated to use Amazon Linux 2018.03. For details, see Execution environment .	May 21, 2019
Node.js 10	A new runtime is available for Node.js 10, nodejs10.x. This runtime uses Node.js 10.15 and will be updated with the latest point release of Node.js 10 periodically. Node.js 10 is also the first runtime to use Amazon Linux 2. For details, see Building Lambda functions with Node.js .	May 13, 2019
GetLayerVersionByArn API	Use the GetLayerVersionByArn API to download layer version information with the version ARN as input. Compared to GetLayerVersion, GetLayerVersionByArn lets you use the ARN directly instead of parsing it to get the layer name and version number.	April 25, 2019
Ruby	AWS Lambda now supports Ruby 2.5 with a new runtime. For details, see Building Lambda functions with Ruby .	November 29, 2018
Layers	With Lambda layers, you can package and deploy libraries, custom runtimes, and other dependencies separately from your function code. Share your layers with your other accounts or the whole world. For details, see Lambda layers .	November 29, 2018
Custom runtimes	Build a custom runtime to run Lambda functions in your favorite programming language. For details, see Custom Lambda runtimes .	November 29, 2018
Application Load Balancer triggers	Elastic Load Balancing now supports Lambda functions as a target for Application Load Balancers. For details, see Using Lambda with application load balancers .	November 29, 2018

<u>Use Kinesis HTTP/2 stream consumers as a trigger</u>	You can use Kinesis HTTP/2 data stream consumers to send events to AWS Lambda. Stream consumers have dedicated read throughput from each shard in your data stream and use HTTP/2 to minimize latency. For details, see Using Lambda with Kinesis .	November 19, 2018
<u>Python 3.7</u>	AWS Lambda now supports Python 3.7 with a new runtime. For more information, see Building Lambda functions with Python .	November 19, 2018
<u>Payload limit increase for asynchronous function invocation</u>	The maximum payload size for asynchronous invocations increased from 128 KB to 256 KB, which matches the maximum message size from an Amazon SNS trigger. For details, see Lambda quotas .	November 16, 2018
<u>AWS GovCloud (US-East) Region</u>	AWS Lambda is now available in the AWS GovCloud (US-East) Region.	November 12, 2018
<u>Moved AWS SAM topics to a separate Developer Guide</u>	A number of topics were focused on building serverless applications using the AWS Serverless Application Model (AWS SAM). These topics have been moved to AWS Serverless Application Model developer guide .	October 25, 2018
<u>View Lambda applications in the console</u>	You can view the status of your Lambda applications on the Applications page in the Lambda console. This page shows the status of the AWS CloudFormation stack. It includes links to pages where you can view more information about the resources in the stack. You can also view aggregate metrics for the application and create custom monitoring dashboards.	October 11, 2018
<u>Function execution timeout limit</u>	To allow for long-running functions, the maximum configurable execution timeout increased from 5 minutes to 15 minutes. For details, see Lambda limits .	October 10, 2018

Support for PowerShell Core language in AWS Lambda	AWS Lambda now supports the PowerShell Core language. For more information, see Programming model for authoring Lambda functions in PowerShell .	September 11, 2018
Support for .NET Core 2.1.0 runtime in AWS Lambda	AWS Lambda now supports the .NET Core 2.1.0 runtime. For more information, see .NET Core CLI .	July 9, 2018
Updates now available over RSS	You can now subscribe to an RSS feed to follow releases for this guide.	July 5, 2018
Support for Amazon SQS as event source	AWS Lambda now supports Amazon Simple Queue Service (Amazon SQS) as an event source. For more information, see Invoking Lambda functions .	June 28, 2018
China (Ningxia) Region	AWS Lambda is now available in the China (Ningxia) Region. For more information about Lambda Regions and endpoints, see Regions and endpoints in the AWS General Reference .	June 28, 2018

Earlier updates

The following table describes the important changes in each release of the *AWS Lambda Developer Guide* before June 2018.

Change	Description	Date
Runtime support for Node.js runtime 8.10	AWS Lambda now supports Node.js runtime version 8.10. For more information, see Building Lambda functions with Node.js (p. 254) .	April 2, 2018
Function and alias revision IDs	AWS Lambda now supports revision IDs on your function versions and aliases. You can use these IDs to track and apply conditional updates when you are updating your function version or alias resources.	January 25, 2018
Runtime support for Go and .NET 2.0	AWS Lambda has added runtime support for Go and .NET 2.0. For more information, see Building Lambda functions with Go (p. 453) and Building Lambda functions with C# (p. 487) .	January 15, 2018
Console Redesign	AWS Lambda has introduced a new Lambda console to simplify your experience and added a Cloud9 Code Editor to enhance your ability debug and revise your function code. For more information, see Edit code using the console editor (p. 21) .	November 30, 2017

Change	Description	Date
Setting Concurrency Limits on Individual Functions	AWS Lambda now supports setting concurrency limits on individual functions. For more information, see Configuring reserved concurrency (p. 210) .	November 30, 2017
Shifting Traffic with Aliases	AWS Lambda now supports shifting traffic with aliases. For more information, see Rolling deployments for Lambda functions (p. 973) .	November 28, 2017
Gradual Code Deployment	AWS Lambda now supports safely deploying new versions of your Lambda function by leveraging Code Deploy. For more information, see Gradual code deployment .	November 28, 2017
China (Beijing) Region	AWS Lambda is now available in the China (Beijing) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the <i>AWS General Reference</i> .	November 9, 2017
Introducing SAM Local	AWS Lambda introduces SAM Local (now known as SAM CLI), a AWS CLI tool that provides an environment for you to develop, test, and analyze your serverless applications locally before uploading them to the Lambda runtime. For more information, see Testing and debugging serverless applications .	August 11, 2017
Canada (Central) Region	AWS Lambda is now available in the Canada (Central) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the <i>AWS General Reference</i> .	June 22, 2017
South America (São Paulo) Region	AWS Lambda is now available in the South America (São Paulo) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the <i>AWS General Reference</i> .	June 6, 2017
AWS Lambda support for AWS X-Ray.	Lambda introduces support for X-Ray, which allows you to detect, analyze, and optimize performance issues with your Lambda applications. For more information, see Using AWS Lambda with AWS X-Ray (p. 807) .	April 19, 2017
Asia Pacific (Mumbai) Region	AWS Lambda is now available in the Asia Pacific (Mumbai) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the <i>AWS General Reference</i> .	March 28, 2017
AWS Lambda now supports Node.js runtime v6.10	AWS Lambda added support for Node.js runtime v6.10. For more information, see Building Lambda functions with Node.js (p. 254) .	March 22, 2017
Europe (London) Region	AWS Lambda is now available in the Europe (London) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the <i>AWS General Reference</i> .	February 1, 2017
AWS Lambda support for the .NET runtime, Lambda@Edge (Preview), Dead Letter Queues and automated deployment of serverless applications.	AWS Lambda added support for C#. For more information, see Building Lambda functions with C# (p. 487) . Lambda@Edge allows you to run Lambda functions at the AWS Edge locations in response to CloudFront events. For more information, see Using AWS Lambda with CloudFront Lambda@Edge (p. 601) .	December 3, 2016

Change	Description	Date
AWS Lambda adds Amazon Lex as a supported event source.	Using Lambda and Amazon Lex, you can quickly build chat bots for various services like Slack and Facebook. For more information, see Using AWS Lambda with Amazon Lex (p. 706) .	November 30, 2016
US West (N. California) Region	AWS Lambda is now available in the US West (N. California) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the <i>AWS General Reference</i> .	November 21, 2016
Introduced the AWS SAM for creating and deploying Lambda-based applications and using environment variables for Lambda function configuration settings.	<p>AWS SAM: You can now use the AWS SAM to define the syntax for expressing resources within a serverless application. In order to deploy your application, simply specify the resources you need as part of your application, along with their associated permissions policies in a AWS CloudFormation template file (written in either JSON or YAML), package your deployment artifacts, and deploy the template. For more information, see AWS Lambda applications (p. 960).</p> <p>Environment variables: You can use environment variables to specify configuration settings for your Lambda function outside of your function code. For more information, see Using AWS Lambda environment variables (p. 76).</p>	November 18, 2016
Asia Pacific (Seoul) Region	AWS Lambda is now available in the Asia Pacific (Seoul) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the <i>AWS General Reference</i> .	August 29, 2016
Asia Pacific (Sydney) Region	Lambda is now available in the Asia Pacific (Sydney) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the <i>AWS General Reference</i> .	June 23, 2016
Updates to the Lambda console	The Lambda console has been updated to simplify the role-creation process.	June 23, 2016
AWS Lambda now supports Node.js runtime v4.3	AWS Lambda added support for Node.js runtime v4.3. For more information, see Building Lambda functions with Node.js (p. 254) .	April 07, 2016
Europe (Frankfurt) region	Lambda is now available in the Europe (Frankfurt) region. For more information about Lambda regions and endpoints, see Regions and endpoints in the <i>AWS General Reference</i> .	March 14, 2016
VPC support	You can now configure a Lambda function to access resources in your VPC. For more information, see Configuring a Lambda function to access resources in a VPC (p. 222) .	February 11, 2016
Lambda runtime has been updated.	The execution environment (p. 37) has been updated.	November 4, 2015

Change	Description	Date
Versioning support, Python for developing code for Lambda functions, scheduled events, and increase in execution time	<p>You can now develop your Lambda function code using Python. For more information, see Building Lambda functions with Python (p. 318).</p> <p>Versioning: You can maintain one or more versions of your Lambda function. Versioning allows you to control which Lambda function version is executed in different environments (for example, development, testing, or production). For more information, see Lambda function versions (p. 83).</p> <p>Scheduled events: You can also set up Lambda to invoke your code on a regular, scheduled basis using the Lambda console. You can specify a fixed rate (number of hours, days, or weeks) or you can specify a cron expression. For an example, see Using AWS Lambda with Amazon EventBridge (CloudWatch Events) (p. 591).</p> <p>Increase in execution time: You can now set up your Lambda functions to run for up to five minutes allowing longer running functions such as large volume data ingestion and processing jobs.</p>	October 08, 2015
Support for DynamoDB Streams	DynamoDB Streams is now generally available and you can use it in all the regions where DynamoDB is available. You can enable DynamoDB Streams for your table and use a Lambda function as a trigger for the table. Triggers are custom actions you take in response to updates made to the DynamoDB table. For an example walkthrough, see Tutorial: Using AWS Lambda with Amazon DynamoDB streams (p. 648) .	July 14, 2015
Lambda now supports invoking Lambda functions with REST-compatible clients.	<p>Until now, to invoke your Lambda function from your web, mobile, or IoT application you needed the AWS SDKs (for example, AWS SDK for Java, AWS SDK for Android, or AWS SDK for iOS). Now, Lambda supports invoking a Lambda function with REST-compatible clients through a customized API that you can create using Amazon API Gateway. You can send requests to your Lambda function endpoint URL. You can configure security on the endpoint to allow open access, leverage AWS Identity and Access Management (IAM) to authorize access, or use API keys to meter access to your Lambda functions by others.</p> <p>For an example Getting Started exercise, see Using AWS Lambda with Amazon API Gateway (p. 562).</p> <p>For more information about the Amazon API Gateway, see https://aws.amazon.com/api-gateway/.</p>	July 09, 2015
The Lambda console now provides blueprints to easily create Lambda functions and test them.	Lambda console provides a set of <i>blueprints</i> . Each blueprint provides a sample event source configuration and sample code for your Lambda function that you can use to easily create Lambda-based applications. All of the Lambda Getting Started exercises now use the blueprints. For more information, see Getting started with Lambda (p. 3) .	July 09, 2015

Change	Description	Date
Lambda now supports Java to author your Lambda functions.	You can now author Lambda code in Java. For more information, see Building Lambda functions with Java (p. 386) .	June 15, 2015
Lambda now supports specifying an Amazon S3 object as the function .zip when creating or updating a Lambda function.	You can upload a Lambda function deployment package (.zip file) to an Amazon S3 bucket in the same region where you want to create a Lambda function. Then, you can specify the bucket name and object key name when you create or update a Lambda function.	May 28, 2015
Lambda now generally available with added support for mobile backends	<p>Lambda is now generally available for production use. The release also introduces new features that make it easier to build mobile, tablet, and Internet of Things (IoT) backends using Lambda that scale automatically without provisioning or managing infrastructure. Lambda now supports both real-time (synchronous) and asynchronous events. Additional features include easier event source configuration and management. The permission model and the programming model have been simplified by the introduction of resource policies for your Lambda functions.</p> <p>The documentation has been updated accordingly. For information, see the following topics:</p> <p>Getting started with Lambda (p. 3)</p> <p>AWS Lambda</p>	April 9, 2015
Preview release	Preview release of the <i>AWS Lambda Developer Guide</i> .	November 13, 2014