
AWS Lambda

Developer Guide



AWS Lambda: Developer Guide

Copyright © 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 should I use Lambda?	1
Lambda features	2
Getting started with Lambda	2
Related services	3
Accessing Lambda	3
Pricing for Lambda	4
Setting up	5
AWS Account	5
AWS CLI	5
AWS SAM	5
AWS SAM CLI	6
Tools for container images	6
Code authoring tools	6
Getting started	8
Create a function	9
Create a Lambda function with default function code	9
Create a function defined as a container image	11
Code editor	17
Working with files and folders	17
Working with code	19
Working in fullscreen mode	22
Working with preferences	22
AWS CLI	23
Prerequisites	23
Create the execution role	23
Create the function	24
List the Lambda functions in your account	27
Retrieve a Lambda function	27
Clean up	28
Concepts	29
Function	29
Trigger	29
Event	29
Execution environment	30
Deployment package	30
Runtime	30
Layer	30
Extension	31
Concurrency	31
Qualifier	31
Features	32
Programming model	32
Scaling	34
Concurrency controls	35
Asynchronous invocation	36
Event source mappings	37
Destinations	38
Function blueprints	39
Testing and deployment tools	40
Application templates	40
Deployment packages	41
Container images	41
.zip file archives	41

Layers	42
Using other AWS services	42
Quotas	44
Permissions	46
Execution role	47
Creating an execution role in the IAM console	48
Grant least privilege access to your Lambda execution role	48
Managing roles with the IAM API	49
AWS managed policies for Lambda features	50
Resource-based policies	51
Granting function access to AWS services	52
Granting function access to other accounts	53
Granting layer access to other accounts	54
Cleaning up resource-based policies	55
User policies	56
Function development	56
Layer development and use	59
Cross-account roles	60
Condition keys for VPC settings	60
Resources and conditions	61
Function resource names	62
Function actions	64
Event source mapping actions	66
Layer actions	66
Permissions boundaries	67
Managing functions	69
Configure console	70
Functions defined as container images	72
Function version \$LATEST	72
Container image deployment	72
Amazon ECR permissions	72
Override the container settings	73
Creating a function (console)	73
Updating the function code (console)	74
Overriding the image parameters (console)	74
Using the Lambda API	75
AWS CloudFormation	75
Environment variables	77
Configuring environment variables	77
Configuring environment variables with the API	78
Example scenario for environment variables	78
Retrieve environment variables	79
Defined runtime environment variables	80
Securing environment variables	81
Sample code and templates	83
Concurrency	84
Configuring reserved concurrency	85
Configuring provisioned concurrency	87
Configuring concurrency with the Lambda API	90
Versions	95
Creating function versions	95
Managing versions with the Lambda API	95
Using versions	96
Granting permissions	96
Aliases	97
Creating a function alias (Console)	97
Managing aliases with the Lambda API	97

Using aliases	98
Resource policies	98
Alias routing configuration	98
Creating layers	101
Creating layer content	101
Compiling the .zip file archive for your layer	101
Including library dependencies in a layer	102
Creating a layer	103
Deleting a layer version	104
Configuring layer permissions	104
Using AWS CloudFormation with layers	105
Using layers	106
Configuring a function to use layers	106
Accessing the contents of a layer	107
Finding layer information	107
Updating a layer version that your function uses	108
Adding layer permissions	108
Using AWS SAM to add a layer to a function	105
Sample applications	109
Memory	110
Configuring function memory	110
Accepting function memory recommendations	110
Network	111
Execution role and user permissions	111
Configuring VPC access (console)	112
Configuring VPC access (API)	112
Using IAM condition keys for VPC settings	113
Internet and service access for VPC-connected functions	116
VPC tutorials	116
Sample VPC configurations	116
Interface VPC endpoints	118
Considerations for Lambda interface endpoints	118
Creating an interface endpoint for Lambda	119
Creating an interface endpoint policy for Lambda	119
Database	121
Creating a database proxy (console)	121
Using the function's permissions for authentication	122
Sample application	122
File system	126
Connecting to a file system (console)	126
Configuring a file system and access point	126
Execution role and user permissions	127
Configuring file system access with the Lambda API	128
AWS CloudFormation and AWS SAM	128
Sample applications	129
Code signing	130
Signature validation	130
Configuration prerequisites	131
Creating code signing configurations	131
Updating a code signing configuration	131
Deleting a code signing configuration	132
Enabling code signing for a function	132
Configuring IAM policies	132
Configuring code signing with the Lambda API	133
Tags	134
Adding tags to a function (console)	134
Using tags to filter functions (console)	134

Using Tags with the AWS CLI	135
Tag Key and Value Requirements	136
Invoking functions	137
Synchronous invocation	138
Asynchronous invocation	140
Configuring error handling for asynchronous invocation	142
Configuring destinations for asynchronous invocation	142
Asynchronous invocation configuration API	144
Dead-letter queues	145
Event source mapping	148
Function states	151
Function scaling	153
Error handling	158
Using extensions	160
Execution environment	161
Impact on performance and resources	161
Permissions	161
Configuring extensions (.zip file archive)	162
Using extensions in container images	162
Next steps	162
Invoking functions defined as container images	164
Function lifecycle	164
Invoking the function	164
Image security	164
Mobile SDK for Android	165
Tutorial	165
Sample code	171
Lambda applications	174
Manage applications	175
Monitoring applications	175
Custom monitoring dashboards	175
Tutorial – Create an application	178
Prerequisites	179
Create an application	179
Invoke the function	180
Add an AWS resource	181
Update the permissions boundary	183
Update the function code	183
Next steps	184
Troubleshooting	185
Clean up	186
Rolling deployments	187
Example AWS SAM Lambda template	187
Use cases	189
Example 1: Amazon S3 pushes events and invokes a Lambda function	189
Example 2: AWS Lambda pulls events from a Kinesis stream and invokes a Lambda function	190
Best practices	192
Function code	192
Function configuration	193
Metrics and alarms	194
Working with streams	194
Lambda runtimes	195
Runtime support policy	198
Execution environment	200
Runtime lifecycle	200
Container images	203
AWS base images for Lambda	203

Runtime interface clients	203
Runtime interface emulator	204
Runtime API	205
Next invocation	205
Invocation response	206
Initialization error	206
Invocation error	207
Extensions API	209
Lambda execution environment lifecycle	210
Extensions API reference	218
Logs API	223
Subscribing to receive logs	224
Memory usage	224
Destination protocols	224
Buffering configuration	224
Example subscription	225
Sample code for Logs API	225
Logs API reference	226
Log messages	227
Runtime modifications	231
Language-specific environment variables	231
Wrapper scripts	233
Custom runtimes	236
Using a custom runtime	236
Building a custom runtime	236
Tutorial – Custom runtime	239
Prerequisites	239
Create a function	239
Create a layer	241
Update the function	242
Update the runtime	243
Share the layer	243
Clean up	243
AVX2 vectorization	245
Compiling from source	245
Enabling AVX2 for Intel MKL	245
AVX2 support in other languages	245
Container images	247
Creating images	248
Image types	248
Container tools	248
Lambda requirements for container images	249
Container image settings	249
Create an image from an AWS base image for Lambda	249
Create an image from an alternative base image	251
Create an image using the AWS SAM toolkit	253
Testing images	254
Guidelines for using the RIE	254
Environment variables	254
Test an image with RIE included in the image	255
Build RIE into your base image	255
Test an image without adding RIE to the image	256
Working with other services	257
Alexa	260
API Gateway	261
Permissions	263
Handling errors with an API Gateway API	264

Choosing an API type	265
Sample applications	267
Tutorial	267
Sample code	277
Microservice blueprint	280
Sample template	281
CloudTrail	283
CloudTrail logs	285
Tutorial	288
Sample code	292
CloudWatch Events	295
Tutorial	296
Sample template	299
Schedule expressions	300
CloudWatch Logs	302
CloudFormation	303
CloudFront (Lambda@Edge)	305
CodeCommit	307
CodePipeline	308
Permissions	309
Cognito	310
Config	311
Connect	312
DynamoDB	313
Execution role permissions	314
Configuring a stream as an event source	315
Event source mapping APIs	316
Error handling	318
Amazon CloudWatch metrics	319
Time windows	319
Reporting batch item failures	322
Tutorial	324
Sample code	329
Sample template	332
EC2	334
Permissions	334
Tutorial – Spot Instances	335
ElastiCache	345
Prerequisites	345
Create the execution role	345
Create an ElastiCache cluster	346
Create a deployment package	346
Create the Lambda function	347
Test the Lambda function	347
Clean up your resources	277
Elastic Load Balancing	349
EFS	351
Connections	351
Throughput	352
IOPS	352
IoT	353
IoT Events	354
Apache Kafka	356
Managing access and permissions	357
Kafka authentication	358
Network configuration	359
Adding a Kafka cluster as an event source	359

Using a Kafka cluster as an event source	361
Auto scaling of the Kafka event source	361
Event source API operations	361
Event source errors	362
Event source configuration parameters	362
Kinesis Firehose	364
Kinesis Streams	365
Configuring your data stream and function	366
Execution role permissions	367
Configuring a stream as an event source	367
Event source mapping API	368
Error handling	370
Amazon CloudWatch metrics	371
Time windows	372
Reporting batch item failures	374
Tutorial	376
Sample code	380
Sample template	383
Lex	386
Roles and permissions	387
MQ	389
Lambda consumer group	390
Execution role permissions	391
Configuring a broker as an event source	391
Event source mapping API	391
Event source mapping errors	393
MSK	395
Managing access and permissions	395
Network configuration	359
Adding Amazon MSK as an event source	397
Auto scaling of the Amazon MSK event source	398
Amazon MSK configuration parameters	362
RDS	400
Tutorial	400
Configuring the function	404
S3	406
Tutorial: Use an S3 trigger	407
Tutorial: Use an S3 trigger to create thumbnails	412
Sample SAM template	425
S3 Batch	426
Invoking Lambda functions from Amazon S3 batch operations	427
Secrets Manager	428
SES	429
SNS	431
Tutorial	432
Sample code	436
SQS	438
Scaling and processing	439
Configuring a queue to use with Lambda	440
Execution role permissions	440
Configuring a queue as an event source	440
Event source mapping APIs	316
Tutorial	442
Sample code	446
Sample template	448
X-Ray	450
Execution role permissions	452

The AWS X-Ray daemon	452
Enabling active tracing with the Lambda API	452
Enabling active tracing with AWS CloudFormation	453
Orchestrating functions	454
Application patterns	454
State machine components	454
State machine application patterns	454
Applying patterns to state machines	455
Example branching application pattern	455
Manage state machines	457
Viewing state machine details	458
Editing a state machine	458
Running a state machine	458
Orchestration examples	459
Configuring a Lambda function as a task	459
Configuring a state machine as an event source	459
Handling function and service errors	460
AWS CloudFormation and AWS SAM	461
Samples	463
Blank function	465
Architecture and handler code	466
Deployment automation with AWS CloudFormation and the AWS CLI	467
Instrumentation with the AWS X-Ray	469
Dependency management with layers	470
Error processor	472
Architecture and event structure	473
Instrumentation with AWS X-Ray	474
AWS CloudFormation template and additional resources	474
List manager	477
Architecture and event structure	478
Instrumentation with AWS X-Ray	479
AWS CloudFormation templates and additional resources	482
Working with Node.js	483
Handler	486
Async handlers	486
Non-async handlers	487
Deploy .zip file archives	488
Prerequisites	488
Updating a function with no dependencies	488
Updating a function with additional dependencies	489
Deploy container images	491
AWS base images for Node.js	491
Using a Node.js base image	491
Node.js runtime interface clients	491
Context	493
Logging	495
Creating a function that returns logs	495
Using the Lambda console	496
Using the CloudWatch console	496
Using the AWS Command Line Interface (AWS CLI)	496
Deleting logs	499
Errors	500
Syntax	500
How it works	500
Using the Lambda console	501
Using the AWS Command Line Interface (AWS CLI)	501
Error handling in other AWS services	502

What's next?	503
Tracing	504
Enabling active tracing with the Lambda API	506
Enabling active tracing with AWS CloudFormation	507
Storing runtime dependencies in a layer	507
Working with Python	509
Handler	511
Naming	511
How it works	511
Returning a value	512
Examples	512
Deploy .zip file archives	514
Prerequisites	514
What is a runtime dependency?	515
Tutorial: Create function	515
Update function	524
Deploy container images	528
AWS base images for Python	528
Python runtime interface clients	529
Deploying Python with an AWS base image	529
Create a Python image from an alternative base image	530
Context	531
Logging	533
Creating a function that returns logs	533
Using the Lambda console	534
Using the CloudWatch console	534
Using the AWS Command Line Interface (AWS CLI)	534
Deleting logs	536
Logging library	536
Errors	538
How it works	538
Using the Lambda console	539
Using the AWS Command Line Interface (AWS CLI)	539
Error handling in other AWS services	540
Error examples	540
Sample applications	541
What's next?	503
Tracing	542
Enabling active tracing with the Lambda API	544
Enabling active tracing with AWS CloudFormation	544
Storing runtime dependencies in a layer	545
Working with Ruby	546
Handler	549
Deploy .zip file archives	550
Prerequisites	550
Tools and libraries	550
Updating a function with no dependencies	551
Updating a function with additional dependencies	551
Deploy container images	553
AWS base images for Ruby	553
Using a Ruby base image	553
Ruby runtime interface clients	553
Context	555
Logging	556
Creating a function that returns logs	556
Using the Lambda console	557
Using the CloudWatch console	557

Using the AWS Command Line Interface (AWS CLI)	558
Deleting logs	560
Errors	561
Syntax	561
How it works	561
Using the Lambda console	562
Using the AWS Command Line Interface (AWS CLI)	562
Error handling in other AWS services	563
Sample applications	564
What's next?	564
Tracing	565
Enabling active tracing with the Lambda API	567
Enabling active tracing with AWS CloudFormation	567
Storing runtime dependencies in a layer	568
Working with Java	569
Sample apps	571
Handler	573
Choosing input and output types	574
Handler interfaces	575
Sample handler code	576
Deploy .zip file archives	577
Prerequisites	577
Tools and libraries	577
Deploy container images	584
AWS base images for Java	584
Using a Java base image	584
Java runtime interface clients	584
Context	586
Context in sample applications	587
Logging	589
Creating a function that returns logs	589
Using the Lambda console	590
Using the CloudWatch console	590
Using the AWS Command Line Interface (AWS CLI)	591
Deleting logs	593
Advanced logging with Log4j 2 and SLF4J	593
Sample logging code	595
Errors	596
Syntax	596
How it works	597
Creating a function that returns exceptions	597
Using the Lambda console	598
Using the AWS Command Line Interface (AWS CLI)	599
Error handling in other AWS services	599
Sample applications	600
What's next?	600
Tracing	601
Enabling active tracing with the Lambda API	603
Enabling active tracing with AWS CloudFormation	603
Storing runtime dependencies in a layer	604
Tracing in sample applications	604
Tutorial - Eclipse IDE	606
Prerequisites	606
Create and build a project	606
Working with Go	609
Handler	610
Lambda function handler using structured types	611

Using global state	612
Context	614
Accessing invoke context information	614
Deploy .zip file archives	616
Prerequisites	616
Tools and libraries	616
Sample applications	616
Creating a .zip file on macOS and Linux	617
Creating a .zip file on Windows	617
Deploy container images	619
AWS base images for Go	619
Go runtime interface clients	619
Using the Go:1.x base image	620
Deploying Go with the provided.al2 base image	620
Deploying Go with an alternative base image	621
Logging	623
Creating a function that returns logs	623
Using the Lambda console	624
Using the CloudWatch console	624
Using the AWS Command Line Interface (AWS CLI)	624
Deleting logs	627
Errors	628
Creating a function that returns exceptions	628
How it works	628
Using the Lambda console	629
Using the AWS Command Line Interface (AWS CLI)	629
Error handling in other AWS services	630
What's next?	631
Tracing	632
Enabling active tracing with the Lambda API	634
Enabling active tracing with AWS CloudFormation	634
Environment variables	636
Working with C#	637
Handler	639
Handling streams	639
Handling standard data types	640
Handler signatures	641
Serializing Lambda functions	641
Lambda function handler restrictions	642
Using async in C# functions with AWS Lambda	642
Deployment package	644
.NET Core CLI	644
AWS Toolkit for Visual Studio	647
Deploy container images	650
AWS base images for .NET	650
Using a .NET base image	650
.NET runtime interface clients	650
Context	651
Logging	652
Creating a function that returns logs	652
Using the Lambda console	653
Using the CloudWatch console	653
Using the AWS Command Line Interface (AWS CLI)	654
Deleting logs	656
Errors	657
Syntax	657
How it works	659

Using the Lambda console	660
Using the AWS Command Line Interface (AWS CLI)	660
Error handling in other AWS services	661
What's next?	661
Tracing	662
Enabling active tracing with the Lambda API	664
Enabling active tracing with AWS CloudFormation	664
Working with PowerShell	666
Development Environment	667
Deployment package	668
Tools and libraries	668
Sample applications	668
Creating a Lambda function	668
Handler	671
Returning data	671
Context	672
Logging	673
Creating a function that returns logs	673
Using the Lambda console	674
Using the CloudWatch console	674
Using the AWS Command Line Interface (AWS CLI)	674
Deleting logs	677
Errors	678
Syntax	678
How it works	679
Using the Lambda console	679
Using the AWS Command Line Interface (AWS CLI)	680
Error handling in other AWS services	680
What's next?	681
Monitoring	682
Monitoring console	683
Pricing	683
Using the Lambda console	683
Types of monitoring graphs	683
Viewing graphs on the Lambda console	683
Viewing queries in the CloudWatch Logs console	684
What's next?	684
Function insights	685
How it works	685
Pricing	685
Supported runtimes	685
Enabling Lambda Insights in the console	685
Enabling Lambda Insights programmatically	686
Using the Lambda Insights dashboard	686
Detecting function anomalies	687
Troubleshooting a function	689
What's next?	684
Function metrics	691
Viewing metrics in the CloudWatch console	691
Types of metrics	691
Function logs	694
Prerequisites	694
Pricing	694
Using the Lambda console	694
Using the AWS CLI	695
What's next?	695
Example workflows	695

Prerequisites	696
Pricing	696
Viewing a service map	697
Viewing trace details	697
Using Trusted Advisor to view recommendations	698
What's next?	698
Security	699
Data protection	699
Encryption in transit	700
Encryption at rest	700
Identity and access management	701
Audience	701
Authenticating with identities	701
Managing access using policies	703
How AWS Lambda works with IAM	705
Identity-based policy examples	705
Troubleshooting	706
Compliance validation	709
Resilience	709
Infrastructure security	710
Configuration and vulnerability analysis	710
Troubleshooting	712
Deployment	712
General: Permission is denied / Cannot load such file	712
General: Error occurs when calling the UpdateFunctionCode	712
Amazon S3: Error Code PermanentRedirect.	713
General: Cannot find, cannot load, unable to import, class not found, no such file or directory ...	713
General: Undefined method handler	713
Lambda: InvalidParameterValueException or RequestEntityTooLargeException	714
Lambda: InvalidParameterValueException	714
Invocation	714
IAM: lambda:InvokeFunction not authorized	715
Lambda: Operation cannot be performed ResourceConflictException	715
Lambda: Function is stuck in Pending	715
Lambda: One function is using all concurrency	715
General: Cannot invoke function with other accounts or services	716
General: Function invocation is looping	716
Lambda: Cold starts with provisioned concurrency	716
Lambda: Latency variability with provisioned concurrency	717
Lambda: Cold starts with new versions	717
EFS: Function could not mount the EFS file system	717
EFS: Function could not connect to the EFS file system	717
EFS: Function could not mount the EFS file system due to timeout	717
Lambda: Lambda detected an IO process that was taking too long	718
Execution	718
Lambda: Execution takes too long	718
Lambda: Logs or traces don't appear	718
Lambda: The function returns before execution finishes	719
AWS SDK: Versions and updates	719
Python: Libraries load incorrectly	719
Networking	719
VPC: Function loses internet access or times out	720
VPC: Function needs access to AWS services without using the internet	720
VPC: Limit was reached for the function's VPC	720
Container images	720
Container: Error occurs on runtime InvalidEntrypoint	720
Lambda: System provisioning additonal capacity	720

CloudFormation: ENTRYPPOINT is being overridden with a null or empty value	721
Releases	722
Earlier updates	731
API reference	736
Actions	736
AddLayerVersionPermission	738
AddPermission	742
CreateAlias	746
CreateCodeSigningConfig	750
CreateEventSourceMapping	753
CreateFunction	762
DeleteAlias	774
DeleteCodeSigningConfig	776
DeleteEventSourceMapping	778
DeleteFunction	783
DeleteFunctionCodeSigningConfig	785
DeleteFunctionConcurrency	787
DeleteFunctionEventInvokeConfig	789
DeleteLayerVersion	791
DeleteProvisionedConcurrencyConfig	793
GetAccountSettings	795
GetAlias	797
GetCodeSigningConfig	800
GetEventSourceMapping	802
GetFunction	807
GetFunctionCodeSigningConfig	811
GetFunctionConcurrency	814
GetFunctionConfiguration	816
GetFunctionEventInvokeConfig	823
GetLayerVersion	826
GetLayerVersionByArn	829
GetLayerVersionPolicy	832
GetPolicy	834
GetProvisionedConcurrencyConfig	836
Invoke	839
InvokeAsync	845
ListAliases	847
ListCodeSigningConfigs	850
ListEventSourceMappings	852
ListFunctionEventInvokeConfigs	855
ListFunctions	858
ListFunctionsByCodeSigningConfig	862
ListLayers	864
ListLayerVersions	866
ListProvisionedConcurrencyConfigs	869
ListTags	872
ListVersionsByFunction	874
PublishLayerVersion	878
PublishVersion	882
PutFunctionCodeSigningConfig	890
PutFunctionConcurrency	893
PutFunctionEventInvokeConfig	896
PutProvisionedConcurrencyConfig	900
RemoveLayerVersionPermission	903
RemovePermission	905
TagResource	907
UntagResource	909

UpdateAlias	911
UpdateCodeSigningConfig	915
UpdateEventSourceMapping	918
UpdateFunctionCode	926
UpdateFunctionConfiguration	935
UpdateFunctionEventInvokeConfig	946
Data Types	949
AccountLimit	951
AccountUsage	952
AliasConfiguration	953
AliasRoutingConfiguration	955
AllowedPublishers	956
CodeSigningConfig	957
CodeSigningPolicies	959
Concurrency	960
DeadLetterConfig	961
DestinationConfig	962
Environment	963
EnvironmentError	964
EnvironmentResponse	965
EventSourceMappingConfiguration	966
FileSystemConfig	970
FunctionCode	971
FunctionCodeLocation	973
FunctionConfiguration	974
FunctionEventInvokeConfig	979
ImageConfig	981
ImageConfigError	982
ImageConfigResponse	983
Layer	984
LayersListItem	985
LayerVersionContentInput	986
LayerVersionContentOutput	987
LayerVersionsListItem	988
OnFailure	990
OnSuccess	991
ProvisionedConcurrencyConfigListItem	992
SelfManagedEventSource	994
SourceAccessConfiguration	995
TracingConfig	996
TracingConfigResponse	997
VpcConfig	998
VpcConfigResponse	999
Certificate errors when using an SDK	999
AWS glossary	1001

What is AWS Lambda?

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, code monitoring and logging. With Lambda, you can run code for virtually any type of application or backend service. All you need to do is supply your code in one of the [languages that Lambda supports \(p. 195\)](#).

Note

In the AWS Lambda Developer Guide, we assume that you have experience with coding, compiling, and deploying programs using one of the supported languages.

You organize your code into [Lambda functions \(p. 29\)](#). Lambda runs your function only when needed and scales automatically, from a few requests per day to thousands per second. You pay only for the compute time that you consume—there is no charge when your code is not running.

You can invoke your Lambda functions using the Lambda API, or Lambda can run your functions in response to events from other AWS services. For example, you can use Lambda to:

- Build data-processing triggers for AWS services such as Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB.
- Process streaming data stored in Amazon Kinesis.
- Create your own backend that operates at AWS scale, performance, and security.

Lambda is a highly available service. For more information, see the [AWS Lambda Service Level Agreement](#).

Sections

- [When should I use Lambda? \(p. 1\)](#)
- [Lambda features \(p. 2\)](#)
- [Getting started with Lambda \(p. 2\)](#)
- [Related services \(p. 3\)](#)
- [Accessing Lambda \(p. 3\)](#)
- [Pricing for Lambda \(p. 4\)](#)

When should I use Lambda?

Lambda is an ideal compute service for many application scenarios, as long as you can run your application code using the Lambda [standard runtime environment \(p. 200\)](#) and within the resources that Lambda provides.

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 \(p. 195\)](#). Lambda performs operational and administrative activities on your behalf, including managing capacity, monitoring, and logging your Lambda functions.

If you need to manage your own compute resources, AWS has other compute services to meet your needs. For example:

- Amazon Elastic Compute Cloud (Amazon EC2) offers a wide range of EC2 instance types to choose from. It lets you customize operating systems, 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.
- AWS Elastic Beanstalk enables you to deploy and scale applications onto Amazon EC2. You retain ownership and full control over the underlying EC2 instances.

Lambda features

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

Concurrency and scaling controls

[Concurrency and scaling controls \(p. 153\)](#) such as concurrency limits and provisioned concurrency give you fine-grained control over the scaling and responsiveness of your production applications.

Functions defined as container images

Use your preferred [container image \(p. 247\)](#) tooling, workflows, and dependencies to build, test, and deploy your Lambda functions.

Code signing

[Code signing \(p. 130\)](#) 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.

Lambda extensions

You can use [Lambda extensions \(p. 209\)](#) to augment your Lambda functions. For example, use extensions to more easily integrate Lambda with your favorite tools for monitoring, observability, security, and governance.

Function blueprints

A function blueprint provides sample code that shows how to use Lambda with other AWS services or third-party applications. Blueprints include sample code and function configuration presets for Node.js and Python runtimes.

Database access

A [database proxy \(p. 121\)](#) manages a pool of database connections and relays queries from a function. This enables a function to reach high concurrency levels without exhausting database connections.

File systems access

You can configure a function to mount an [Amazon Elastic File System \(Amazon EFS\) file system \(p. 126\)](#) to a local directory. With Amazon EFS, your function code can access and modify shared resources safely and at high concurrency.

Getting started with Lambda

To work effectively with Lambda, you need coding experience and expertise in the following domains:

- Linux OS and commands, as well as concepts such as processes, threads, and file permissions.
- Cloud concepts and IP networking concepts (for public and private networks).
- Distributed computing concepts such as HTML as an IPC, queues, messaging, notifications, and concurrency.

- Familiarity with security services and concepts: AWS Identity and Access Management (IAM) and access control principles, and AWS Key Management Service (AWS KMS) and public key infrastructure.
- Familiarity with key services that interact with Lambda: Amazon API Gateway, Amazon S3, Amazon Simple Queue Service (Amazon SQS), and DynamoDB.
- Configuring EC2 instances with Linux.

If you are a first-time user of Lambda, we recommend that you start with the following topics to help you learn the basics:

1. Read the [Lambda product overview](#) and explore the [Lambda getting started page](#).
2. To create and test a Lambda function using the Lambda console, try the [console-based getting started exercise \(p. 8\)](#). This exercise teaches you about the Lambda programming model and other concepts.
3. If you are familiar with container image workflows, try the [getting started exercise to create a Lambda function defined as a container image \(p. 11\)](#).

AWS also provides the following resources for learning about serverless applications and Lambda:

- The [AWS Compute Blog](#) includes useful articles about Lambda.
- [AWS Serverless](#) provides blogs, videos, and training related to AWS serverless development.
- 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.

Related services

Lambda integrates with other AWS services (p. 257) to invoke functions based on events that you specify. For example:

- Use [API Gateway \(p. 261\)](#) to provide a secure and scalable gateway for web APIs that route HTTP requests to Lambda functions.
- For services that generate a queue or data stream (such as [DynamoDB \(p. 313\)](#) and [Kinesis \(p. 365\)](#)), Lambda polls the queue or data stream from the service and invokes your function to process the received data.
- Define [Amazon S3 \(p. 406\)](#) events that invoke a Lambda function to process Amazon S3 objects, for example, when an object is created or deleted.
- Use a Lambda function to process [Amazon SQS \(p. 438\)](#) messages or [Amazon Simple Notification Service \(Amazon SNS\) \(p. 431\)](#) notifications.
- Use [AWS Step Functions \(p. 454\)](#) to connect Lambda functions together into serverless workflows called state machines.

Accessing Lambda

You can create, invoke, and manage your 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 [Configuring functions in the console \(p. 70\)](#).
- **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 AWS Lambda with the AWS Command Line Interface \(p. 23\)](#).

- **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 \(p. 174\)](#). AWS CloudFormation also supports the [AWS Cloud Development Kit \(CDK\)](#).
- **AWS Serverless Application Model (AWS SAM)** – Provides templates and a CLI to configure and manage AWS serverless applications. For more information, see [AWS SAM \(p. 5\)](#).

Pricing for Lambda

There is no additional charge for creating Lambda functions. There are charges for running a function and for data transfer between Lambda and other AWS services. Some optional Lambda features (such as [provisioned concurrency \(p. 84\)](#)) also incur charges. For more information, see [AWS Lambda Pricing](#).

Setting up with Lambda

To use AWS Lambda, you need an AWS account. If you plan to configure and use Lambda functions from the command line, set up the AWS CLI. You can set up other development and build tools as required for the environment and language that you are planning to use.

Sections

- [AWS Account \(p. 5\)](#)
- [AWS CLI \(p. 5\)](#)
- [AWS SAM \(p. 5\)](#)
- [AWS SAM CLI \(p. 6\)](#)
- [Tools for container images \(p. 6\)](#)
- [Code authoring tools \(p. 6\)](#)

AWS Account

To use Lambda and other AWS services, you need an AWS account. If you do not have an account, visit aws.amazon.com and choose **Create an AWS Account**. For instructions, see [How do I create and activate a new AWS account?](#)

As a best practice, create an AWS Identity and Access Management (IAM) user with administrator permissions, and then use that IAM user for all work that does not require root credentials. Create a password for console access, and create access keys to use command line tools. For instructions, see [Creating your first IAM admin user and group](#) in the *IAM User Guide*.

AWS CLI

If you plan to configure and use Lambda functions from the command line, install the AWS Command Line Interface (AWS CLI). Tutorials in this guide use the AWS CLI, which has commands for all Lambda API operations. Some functionality is not available in the Lambda console and can be accessed only with the AWS CLI or the AWS SDKs.

To set up the AWS CLI, see the following topics in the *AWS Command Line Interface User Guide*.

- [Installing, updating, and uninstalling the AWS CLI](#)
- [Configuring the AWS CLI](#)

To verify that the AWS CLI is configured correctly, run the `list-functions` command to see a list of your Lambda functions in the current AWS Region.

```
aws lambda list-functions
```

AWS SAM

The AWS Serverless Application Model (AWS SAM) is an extension for the AWS CloudFormation template language that lets you define serverless applications at a higher level. AWS SAM abstracts away common

tasks such as function role creation, making it easier to write templates. AWS SAM is supported directly by AWS CloudFormation, and includes additional functionality through the AWS CLI and AWS SAM CLI.

For more information about AWS SAM templates, see the [AWS SAM specification](#) in the *AWS Serverless Application Model Developer Guide*.

AWS SAM CLI

The AWS SAM CLI is a separate command line tool that you can use to manage and test AWS SAM applications. In addition to commands for uploading artifacts and launching AWS CloudFormation stacks that are also available in the AWS CLI, the AWS SAM CLI provides commands for validating templates and running applications locally in a Docker container. You can use the AWS SAM CLI to build functions deployed as .zip file archives or container images.

To set up the AWS SAM CLI, see [Installing the AWS SAM CLI](#) in the *AWS Serverless Application Model Developer Guide*.

Tools for container images

To create and test functions deployed as container images, you can use native container tools such as the Docker CLI.

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*.

Code authoring tools

You can author your Lambda function code in the languages that Lambda supports. For a list of supported languages, see [Lambda runtimes \(p. 195\)](#). There are tools for authoring code, such as the Lambda console, Eclipse integrated development environment (IDE), and Visual Studio IDE. But 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.

The following table lists the languages that Lambda supports, and the tools and options that you can use with them.

Language	Tools and options for authoring code
Node.js	<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	<ul style="list-style-type: none">• Eclipse, with the AWS Toolkit for Eclipse• IntelliJ, with the AWS Toolkit for JetBrains• Your own authoring environment

Language	Tools and options for authoring code
C#	<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	<ul style="list-style-type: none">Lambda consolePyCharm, with the AWS Toolkit for JetBrainsYour own authoring environment
Ruby	<ul style="list-style-type: none">Lambda consoleYour own authoring environment
Go	<ul style="list-style-type: none">Your own authoring environment
PowerShell	<ul style="list-style-type: none">Your own authoring environmentPowerShell 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)

Getting started with Lambda

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

Note

To use Lambda and other AWS services, you need an AWS account. If you don't have an account, visit aws.amazon.com and choose **Create an AWS Account**. For instructions, see [How do I create and activate a new AWS account?](#)

As a best practice, create an AWS Identity and Access Management (IAM) user with administrator permissions, and then use that IAM user for all work that does not require root credentials. Create a password for console access, and create access keys to use command line tools. For instructions, see [Creating your first IAM admin user and group](#) in the *IAM User Guide*.

You can author functions in the Lambda console, or with an IDE toolkit, command line tools, or the AWS SDKs. The Lambda console provides a [code editor \(p. 17\)](#) for non-compiled languages that lets you modify and test code quickly. The [AWS Command Line Interface \(AWS CLI\) \(p. 23\)](#) gives you direct access to the Lambda API for advanced configuration and automation use cases.

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

- A .zip file archive that contains your function code and its dependencies. For an example tutorial, see [Using AWS Lambda with the AWS Command Line Interface \(p. 23\)](#).
- A container image that is compatible with the [Open Container Initiative \(OCI\)](#) specification. For an example tutorial, see [Create a function defined as a container image \(p. 11\)](#).

Topics

- [Create a Lambda function with the console \(p. 9\)](#)
- [Creating functions using the AWS Lambda console editor \(p. 17\)](#)
- [Using AWS Lambda with the AWS Command Line Interface \(p. 23\)](#)
- [Lambda concepts \(p. 29\)](#)
- [Lambda features \(p. 32\)](#)
- [Lambda deployment packages \(p. 41\)](#)
- [Lambda quotas \(p. 44\)](#)

Create a Lambda function with the console

In the following getting started exercises, you create Lambda functions using the console.

In the first exercise, you create a function and use the default code that Lambda creates. The Lambda console provides a [code editor \(p. 17\)](#) for non-compiled languages that lets you modify and test code quickly.

In the next exercise, you create a function defined as a container image. First, create a container image for your function code, and then use the Lambda console to create a function from the container image.

Topics

- [Create a Lambda function with default function code \(p. 9\)](#)
- [Create a function defined as a container image \(p. 11\)](#)

Create a Lambda function with default function code

In this getting started exercise, you create a Node.js Lambda function using the Lambda console. Lambda automatically creates default code for the function. Next, you manually invoke the Lambda function using sample event data. Lambda runs the function and returns results. You then verify the results, including the logs that your Lambda function created and various Amazon CloudWatch metrics.

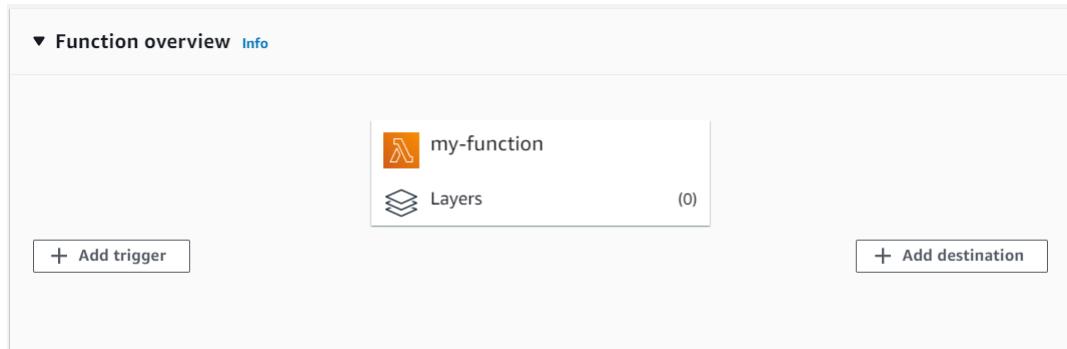
To create a Lambda function with the console

1. Open the [Functions page](#) on the Lambda console.
2. Choose **Create function**.
3. Under **Basic information**, do the following:
 - a. For **Function name**, enter `my-function`.
 - b. For **Runtime**, confirm that **Node.js 14.x** is selected. Note that Lambda provides runtimes for .NET (PowerShell, C#), Go, Java, Node.js, Python, and Ruby.
4. Choose **Create function**.

Lambda creates a Node.js function and an [execution role \(p. 47\)](#) that grants the function permission to upload logs. 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.

Use the function overview

The **Function overview** shows a visualization of your function, including any triggers, destinations, and layer that you have configured for the function.



Invoke the Lambda function

Invoke your Lambda function using the sample event data provided in the console.

To invoke a function

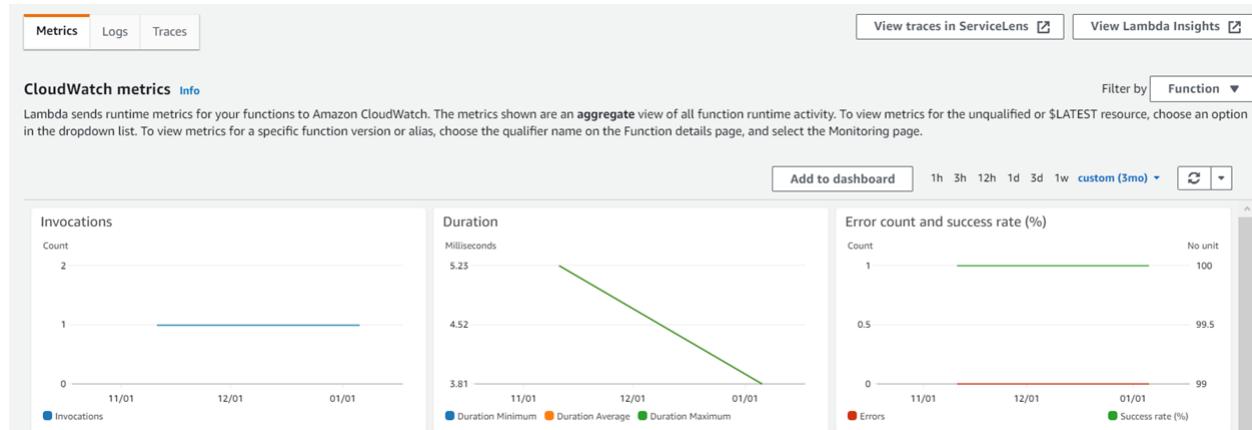
1. After selecting your function, choose the **Test** tab.
2. In the **Test event** section, choose **New event**. In **Template**, leave the default **hello-world** option. Enter a **Name** for this test and note the following sample event template:

```
{  
    "key1": "value1",  
    "key2": "value2",  
    "key3": "value3"  
}
```

3. Choose **Save changes**, and then choose **Test**. Each user can create up to 10 test events per function. Those test events are not available to other users.

Lambda runs your function on your behalf. The function handler receives and then processes the sample event.

4. Upon successful completion, view the results in the console.
 - The **Execution result** shows the execution status as **succeeded**. To view the function execution results, expand **Details**. Note that the **logs** link opens the **Log groups** page in the CloudWatch console.
 - The **Summary** section shows the key information reported in the **Log output** section (the *REPORT* line in the execution log).
 - The **Log output** section shows the log that Lambda generates for each invocation. The function writes these logs to CloudWatch. The Lambda console shows these logs for your convenience. Choose **Click here** to add logs to the CloudWatch log group and open the **Log groups** page in the CloudWatch console.
5. Run the function (choose **Test**) a few more times to gather some metrics that you can view in the next step.
6. Choose the **Monitor** tab. This page shows graphs for the metrics that Lambda sends to CloudWatch.



For more information on these graphs, see [Monitoring functions in the AWS Lambda console \(p. 683\)](#).

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

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Actions, Delete**.
4. In the **Delete function** dialog box, choose **Delete**.

To delete the log group

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

To delete the execution role

1. Open the [Roles page](#) of the AWS Identity and Access Management (IAM) console.
2. Select the function's role (my-function-role-*31exxmp1*).
3. Choose **Delete role**.
4. In the **Delete role** dialog box, choose **Yes, 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 \(p. 463\)](#).

Create a function defined as a container image

In this getting started exercise, you use the Docker CLI to create a container image and then use the Lambda console to create a function from the container image.

Topics

- [Prerequisites \(p. 11\)](#)
- [Create the container image \(p. 12\)](#)
- [Upload the image to the Amazon ECR repository \(p. 13\)](#)
- [Update the user permissions \(p. 13\)](#)
- [Create a Lambda function defined as a container image \(p. 14\)](#)
- [Invoke the Lambda function \(p. 14\)](#)
- [Clean up \(p. 15\)](#)

Prerequisites

To complete the following steps, you need a command line terminal or shell to run commands. 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. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

This exercise uses Docker CLI commands to create the container image. To install the Docker CLI, see [Get Docker](#) on the Docker Docs website.

Create the container image

AWS provides a set of base images in the Amazon Elastic Container Registry (Amazon ECR). In this getting started exercise, we use the Node.js base image to create a container image. For more information about base images, see [AWS base images for Lambda \(p. 203\)](#).

In the following commands, replace 123456789012 with your AWS account ID.

To create an image using the AWS Node.js 12 base image

1. On your local machine, create a project directory for your new function.
2. Create a file named `app.js` in your project directory. Add the following code to `app.js`:

```
exports.handler = async (event) => {
    // TODO implement
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

3. Use a text editor to create a new file named `Dockerfile` in your project directory. Add the following content to `Dockerfile`:

```
FROM public.ecr.aws/lambda/nodejs:12

# Copy function code and package.json
COPY app.js package.json /var/task/

# Install NPM dependencies for function
RUN npm install

# Set the CMD to your handler
CMD [ "app.handler" ]
```

4. Create the `package.json` file. From your project directory, run the `npm init` command. Accept all of the default values:

```
npm init
```

5. Build your Docker image. From your project directory, run the following command:

```
docker build -t hello-world .
```

6. (Optional) AWS base images include the Lambda runtime interface emulator, so you can test your function locally.
 - a. Run your Docker image. From your project directory, run the `docker run` command:

```
docker run -p 9000:8080 hello-world:latest
```

6. (Optional) AWS base images include the Lambda runtime interface emulator, so you can test your function locally.
 - b. Test your Lambda function. In a new terminal window, run a `curl` command to invoke your function:

```
curl -XPOST "http://localhost:9000/2015-03-31/functions/function/invocations" -d
'{}'
```

Upload the image to the Amazon ECR repository

1. Authenticate the Docker CLI to your Amazon ECR registry.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-
stdin 123456789012.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
```

3. Tag your image to match your repository name using the `docker tag` command.

```
docker tag hello-world:latest 123456789012.dkr.ecr.us-east-1.amazonaws.com/hello-
world:latest
```

4. Deploy the image to Amazon ECR using the `docker push` command.

```
docker push 123456789012.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

Update the user permissions

Make sure that the permissions for the IAM user or role that creates the function contain the AWS managed policies `GetRepositoryPolicy`, `SetRepositoryPolicy`, and `InitiateLayerUpload`. For information about access policies, see [Access Management](#) in the *IAM User Guide*

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",
      "ecr:InitiateLayerUpload"],
      "Resource": "arn:aws:ecr:<region>:<account>:repository/<repo name>/"
    }
  ]
}
```

```
    ]  
}
```

Create a Lambda function defined as a container image

Use the Lambda console to create a function defined as a container image.

To create the function with the console

1. Open the [Functions page](#) on 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 `my-function`.
 - b. For **Container image URI**, enter the URI of the Amazon ECR image that you created previously.
5. Choose **Create function**.

Lambda creates your function and an [execution role \(p. 47\)](#) 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.

Invoke the Lambda 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. In the **Test event** section, choose **New event**. In **Template**, leave the default **hello-world** option. Enter a **Name** for this test and note the following sample event template:

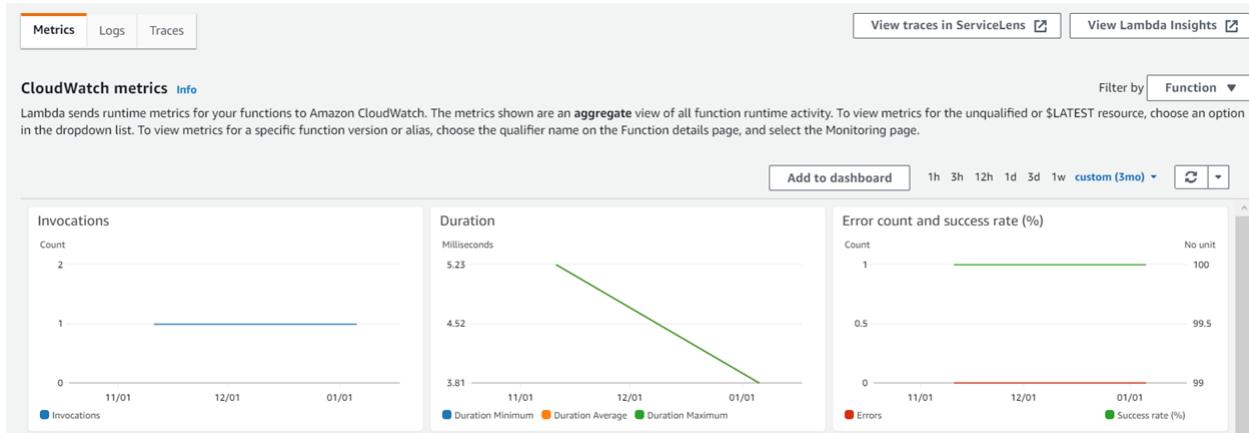
```
{  
  "key1": "value1",  
  "key2": "value2",  
  "key3": "value3"  
}
```

3. Choose **Save changes**, and then choose **Test**. Each user can create up to 10 test events per function. Those test events are not available to other users.

Lambda runs your function on your behalf. The function handler receives and then processes the sample event.

4. Upon successful completion, view the results in the console.
 - The **Execution result** shows the execution status as **succeeded**. To view the function execution results, expand **Details**. Note that the **logs** link opens the **Log groups** page in the CloudWatch console.
 - The **Summary** section shows the key information reported in the **Log output** section (the *REPORT* line in the execution log).
 - The **Log output** section shows the log that Lambda generates for each invocation. The function writes these logs to CloudWatch. The Lambda console shows these logs for your convenience. Choose **Click here** to add logs to the CloudWatch log group and open the **Log groups** page in the CloudWatch console.

5. Run the function (choose **Test**) a few more times to gather some metrics that you can view in the next step.
6. Choose the **Monitor** tab. This page shows graphs for the metrics that Lambda sends to CloudWatch.



For more information on these graphs, see [Monitoring functions in the AWS Lambda console \(p. 683\)](#).

Clean up

If you are finished with the container image, see [Deleting an image](#) in the *Amazon Elastic Container Registry User Guide*

If you are done working with your 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

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Actions, Delete**.
4. In the **Delete function** dialog box, choose **Delete**.

To delete the log group

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

To delete the execution role

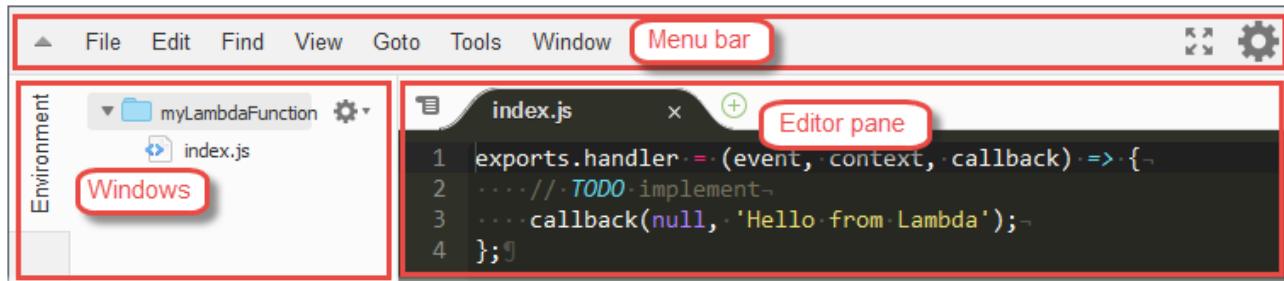
1. Open the [Roles page](#) of the IAM console.
2. Select the function's role (my-function-role-31exxmpl).
3. Choose **Delete role**.
4. In the **Delete role** dialog box, choose **Yes, delete**.

You can automate the creation and cleanup of functions, log groups, and roles with AWS CloudFormation and the AWS CLI. For fully functional sample applications, see [Lambda sample applications \(p. 463\)](#).

Creating functions using the AWS Lambda console editor

The Lambda console provides a code editor for languages that do not require compiling, such as Node.js or Python. The code editor in the AWS Lambda console enables you to write, test, and view the execution results of your Lambda function code.

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



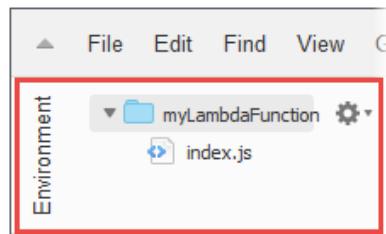
For a list of what the commands do, see the [Menu commands reference](#) in the *AWS Cloud9 User Guide*. Note that some of the commands listed in that reference are not available in the code editor.

Topics

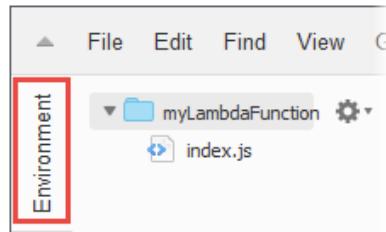
- [Working with files and folders \(p. 17\)](#)
- [Working with code \(p. 19\)](#)
- [Working in fullscreen mode \(p. 22\)](#)
- [Working with preferences \(p. 22\)](#)

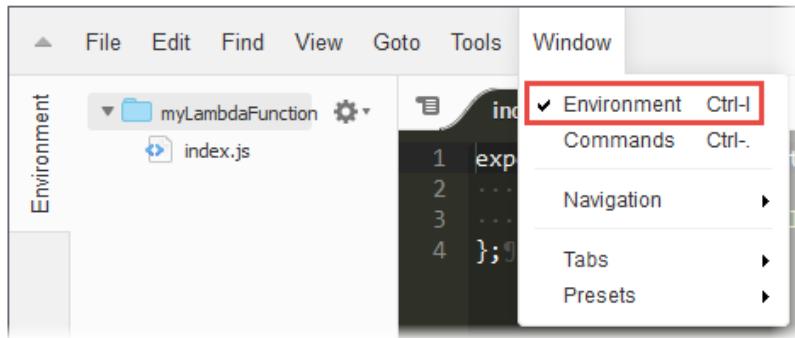
Working with files and folders

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.



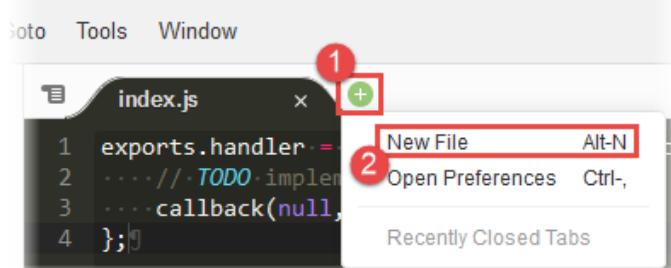


To open a single file and show its contents in the editor pane, double-click the file in the **Environment** window.

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**.

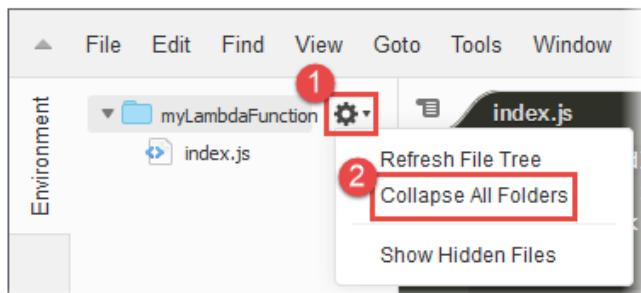
To save a file, with the file open and its contents visible in the editor pane, choose **File, Save** on the menu bar.

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.

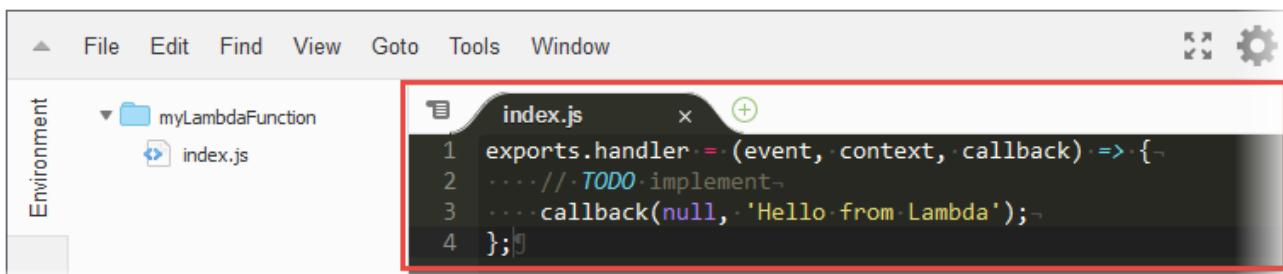
To collapse folders, choose the gear icon in the **Environment** window, and then choose **Collapse All Folders**.



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

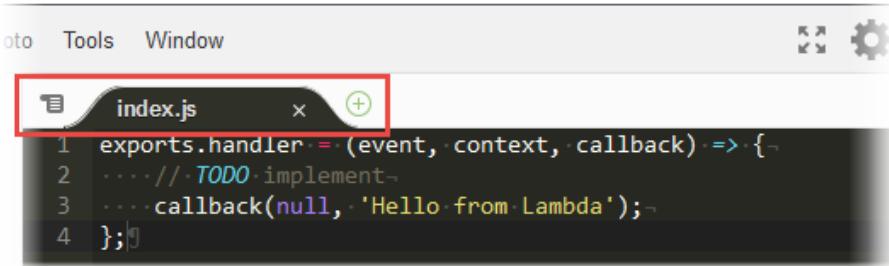
Working with code

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

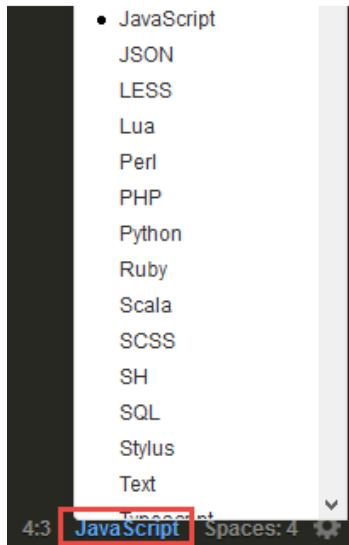
Use the status bar to move quickly to a line in the active file and to change how code is displayed.



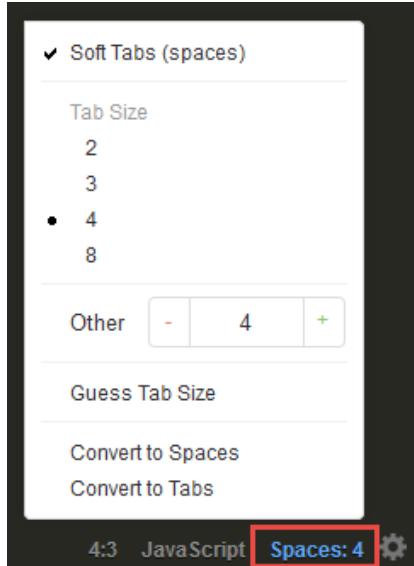
To move quickly to a line in the active file, choose the line selector, type the line number to go to, and then press **Enter**.



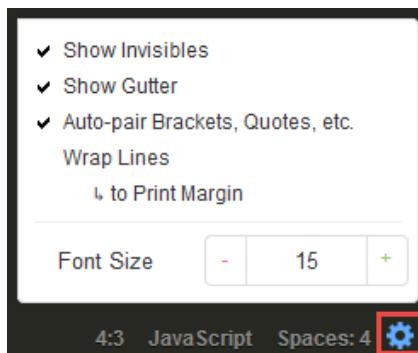
To change the code color scheme in the active file, choose the code color scheme selector, and then choose the new code color scheme.



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.



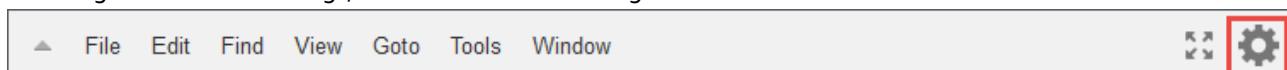
To shrink the code editor to its original size, choose the **Toggle fullscreen** button again.

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.

Using AWS Lambda with the AWS Command Line Interface

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" \(p. 9\)](#).

To complete the following steps, you need a command line terminal or shell to run commands. 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. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

This tutorial uses the AWS Command Line Interface (AWS CLI) to call service API operations. To install the AWS CLI, see [Installing the AWS CLI](#) in the AWS Command Line Interface User Guide.

Create the execution role

Create the [execution role \(p. 47\)](#) that gives your function permission to access AWS resources. To create an execution role with the AWS CLI, use the `create-role` command.

```
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": "*",
          "Action": "sts:AssumeRole"
        }
      ]
    }
  }
}
```

```
        "Effect": "Allow",
        "Principal": {
            "Service": "lambda.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
    }
}
}
```

The `trust-policy.json` file is a JSON file in the current directory that defines the [trust policy](#) for the role. 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 `AssumeRole` action.

Example `trust-policy.json`

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

You can also 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"}]}'
```

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 nodejs12.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": "nodejs12.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":  
    "U1RBULQgUmVxdWVzdElkOiaA4N2QwNDRiOC1mMTU0LTEzTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc21vb...",  
    "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:
$LATEST\n",
      "ingestionTime": 1559763003309
    },
    {
      "timestamp": 1559763003173,
      "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\", \r ...",
      "ingestionTime": 1559763018353
    },
    {
      "timestamp": 1559763003173,
      "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
      "ingestionTime": 1559763018353
    },
    {
      "timestamp": 1559763003218,
      "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
      "ingestionTime": 1559763018353
    },
    {
      "timestamp": 1559763003218,
```

```
        "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75 MB\t\n",
        "ingestionTime": 1559763018353
    },
],
"nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
"nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

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": "nodejs12.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": "nodejs12.x",
            "Role": "arn:aws:iam::123456789012:role/lambda-role",
            "Handler": "index.handler",
            ...
        },
        ...
    ],
    "NextToken": "eyJNYXJrZXIiOiBudWxsLCAiYm90b190cnVuY2F0ZV9hbW91bnQiOiAxMH0="
}
```

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 eyJNYXJrZXIiOiBudWxsLCAiYm90b190cnVuY2F0ZV9hbW91bnQiOiAxMH0=
```

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": "nodejs12.x",  
        "Role": "arn:aws:iam::123456789012:role/lambda-ex",  
        "CodeSha256": "FpFMvUhayLkOoVBpNuNiIVML/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 \(p. 807\)](#).

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 concepts

With Lambda, you run functions to process events. To send events to your function, you can invoke it using the Lambda API, or you can configure an AWS service or resource to invoke it.

Concepts

- [Function \(p. 29\)](#)
- [Trigger \(p. 29\)](#)
- [Event \(p. 29\)](#)
- [Execution environment \(p. 30\)](#)
- [Deployment package \(p. 30\)](#)
- [Runtime \(p. 30\)](#)
- [Layer \(p. 30\)](#)
- [Extension \(p. 31\)](#)
- [Concurrency \(p. 31\)](#)
- [Qualifier \(p. 31\)](#)

Function

A *function* is a resource that you can invoke to run your code in Lambda. A function has code to process the [events \(p. 29\)](#) that you pass into the function or that other AWS services send to the function.

For more information, see [Managing AWS Lambda functions \(p. 69\)](#).

Trigger

A *trigger* is a resource or configuration that invokes a Lambda function. This includes AWS services that you can configure to invoke a function, applications that you develop, and [event source mappings \(p. 148\)](#). 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 AWS Lambda functions \(p. 137\)](#) and [Using AWS Lambda with other services \(p. 257\)](#).

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/ebXEkAi6RibDsvpi+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 \(p. 257\)](#).

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 \(p. 31\)](#) associated with your function.

For more information, see [AWS Lambda execution environment \(p. 200\)](#).

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 \(p. 41\)](#).

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 \(p. 195\)](#).

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 \(p. 236\)](#), 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 \(p. 44\)](#). 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 \(p. 101\)](#).

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 \(p. 209\)](#).

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 [Using Lambda extensions \(p. 160\)](#).

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 \(p. 44\)](#) 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 [Managing concurrency for a Lambda function \(p. 84\)](#).

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 \(p. 95\)](#).

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.

Features

- [Programming model \(p. 32\)](#)
- [Scaling \(p. 34\)](#)
- [Concurrency controls \(p. 35\)](#)
- [Asynchronous invocation \(p. 36\)](#)
- [Event source mappings \(p. 37\)](#)
- [Destinations \(p. 38\)](#)
- [Function blueprints \(p. 39\)](#)
- [Testing and deployment tools \(p. 40\)](#)
- [Application templates \(p. 40\)](#)

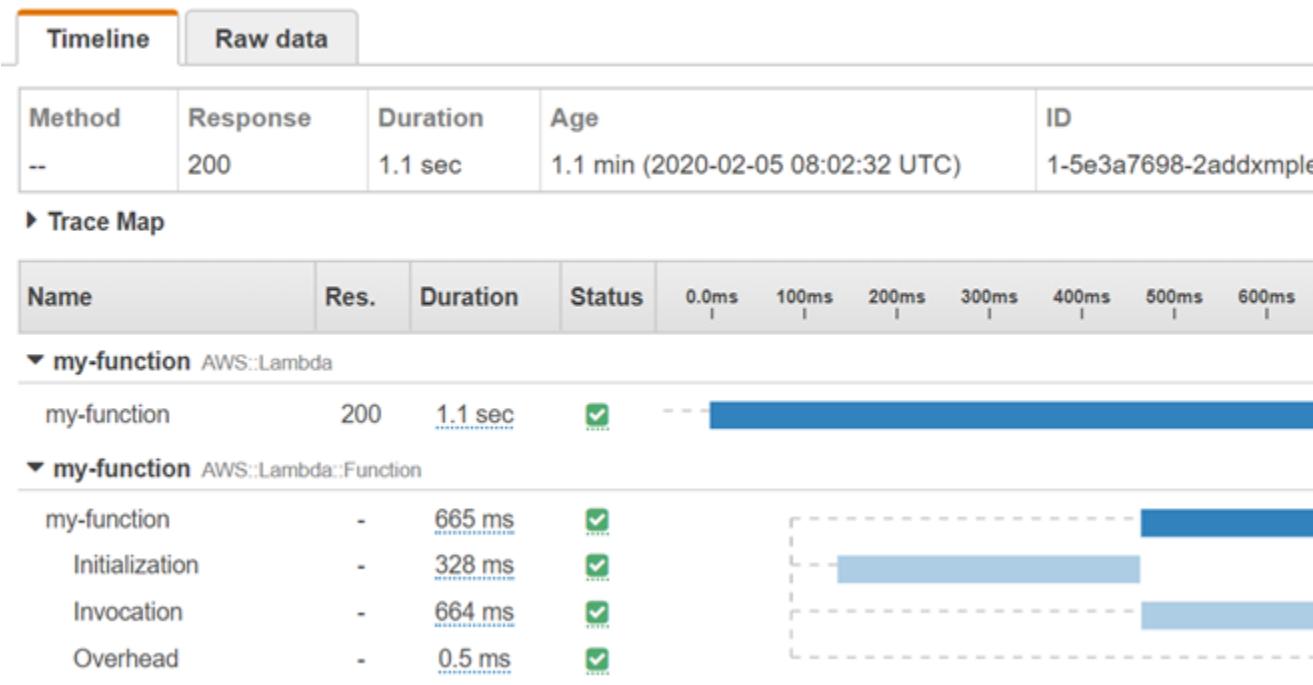
Programming model

Authoring specifics vary between runtimes, but all runtimes share a common programming model that defines the interface between your code and the runtime code. You tell the runtime which method to run by defining a *handler* in the function configuration, and the runtime runs that method. 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.

When [AWS X-Ray tracing \(p. 450\)](#) is enabled, the runtime records separate subsegments for initialization and execution.

Traces > Details



Your function also has access to local storage in the `/tmp` directory. Instances of your function that are serving requests remain active for a few hours before being recycled.

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.

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

- [Building Lambda functions with Node.js \(p. 483\)](#)
- [Building Lambda functions with Python \(p. 509\)](#)
- [Building Lambda functions with Ruby \(p. 546\)](#)
- [Building Lambda functions with Java \(p. 569\)](#)
- [Building Lambda functions with Go \(p. 609\)](#)
- [Building Lambda functions with C# \(p. 637\)](#)
- [Building Lambda functions with PowerShell \(p. 666\)](#)

Lambda scales your function by running additional instances of it as demand increases, and by stopping instances as demand decreases. Unless noted otherwise, incoming requests might be processed out of order or concurrently. Store your application's state in other services, and don't rely on instances of your function being long lived. 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.

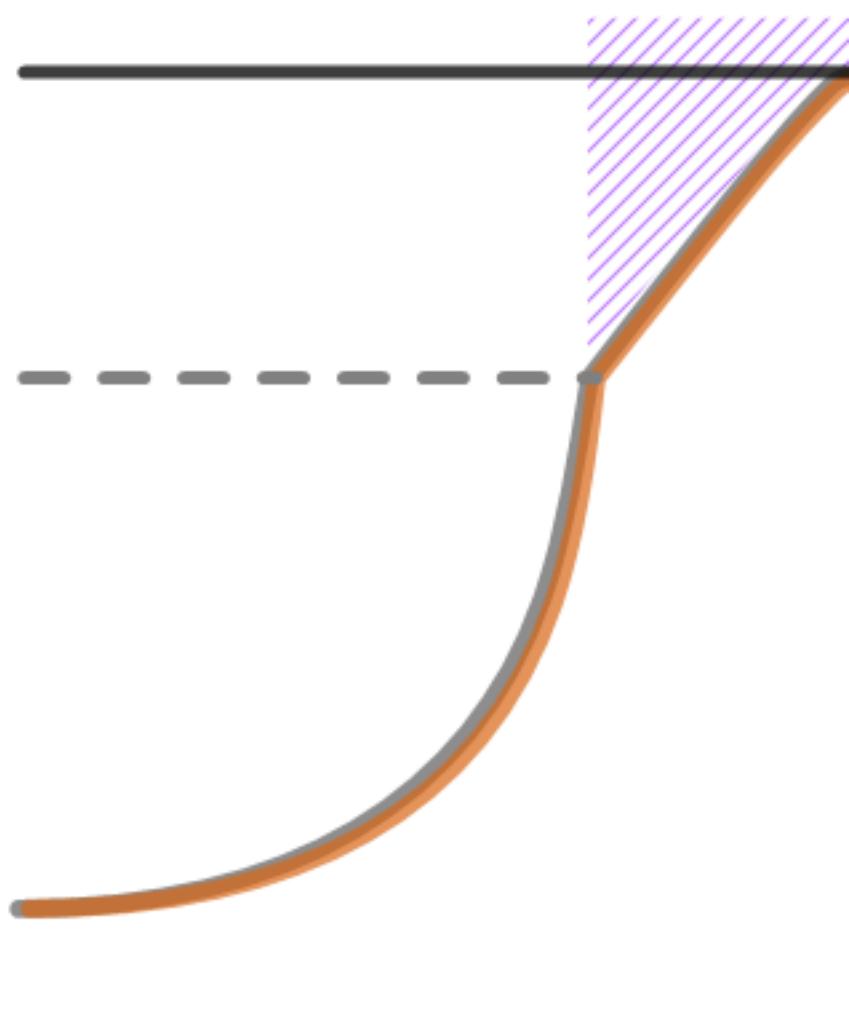
Scaling

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 only pay for the time that your function is initializing or processing events.

Function Scaling with Concurrency Limit

Concurrency
limit

Burst limit

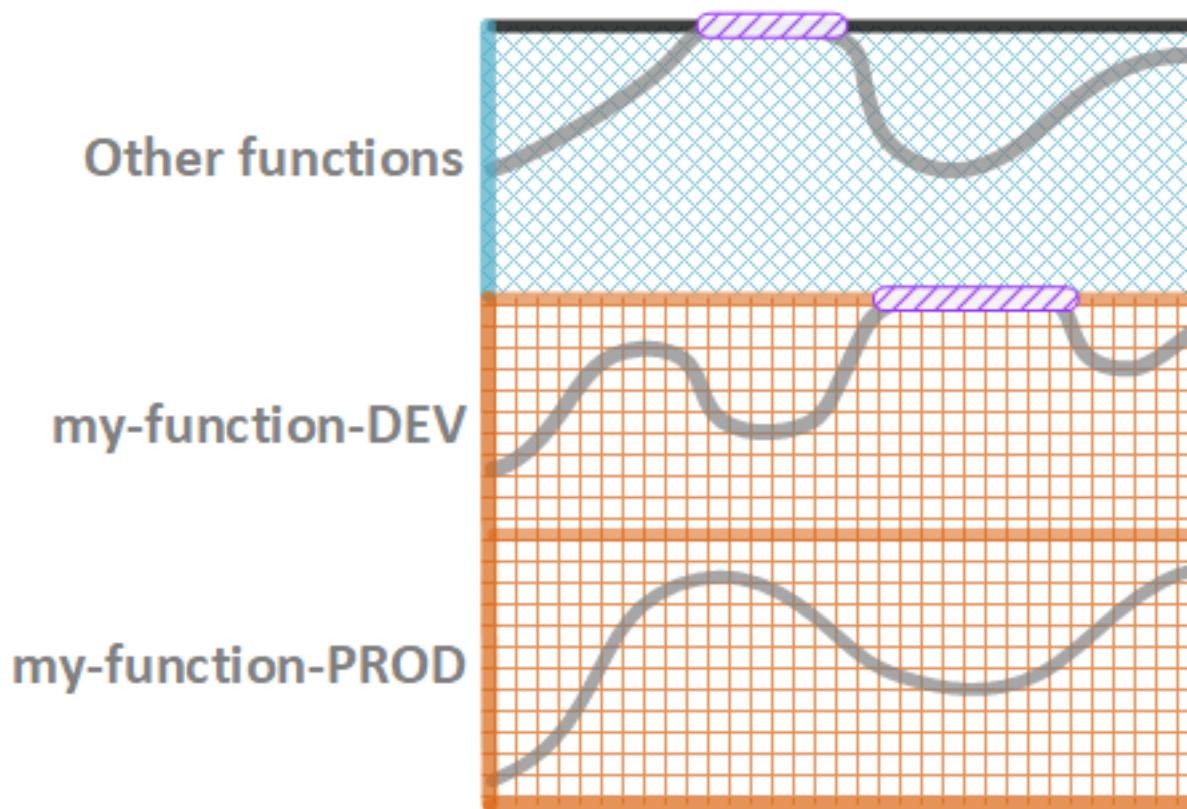


For more information, see [AWS Lambda function scaling \(p. 153\)](#).

Concurrency controls

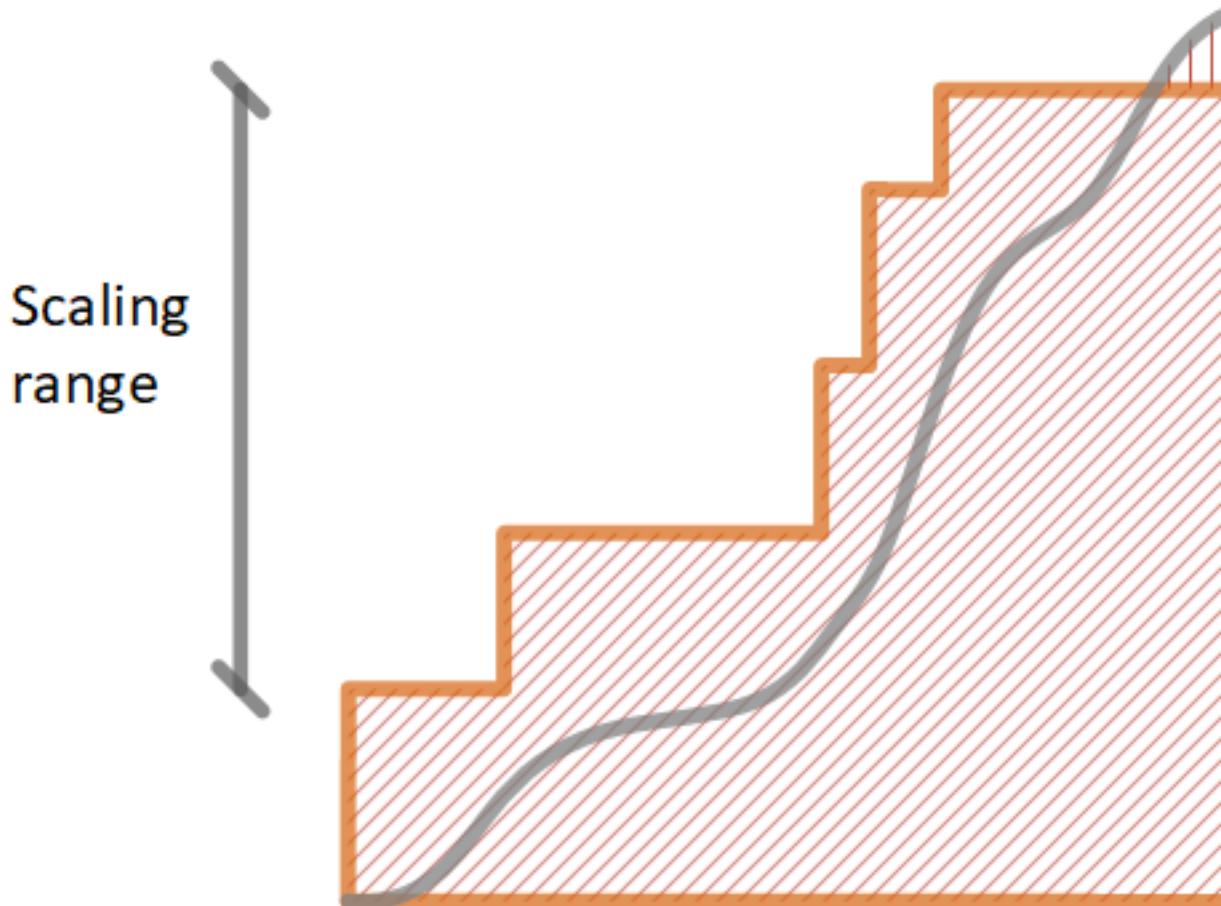
Use concurrency settings to ensure that your production applications are highly available and highly responsive. To prevent a function from using too much concurrency, and to reserve a portion of your account's available concurrency for a function, use *reserved concurrency*. Reserved concurrency splits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

Reserved Concurrency



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.

Autoscaling with Provisioned Concurrency

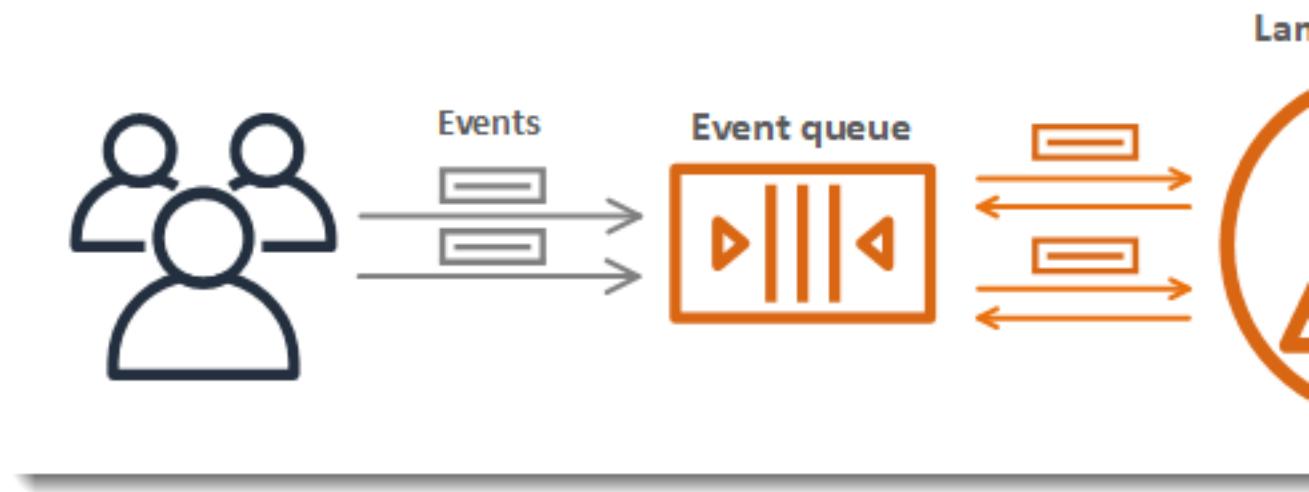


For more information, see [Managing concurrency for a Lambda function \(p. 84\)](#).

Asynchronous invocation

When you invoke a function, you can choose to invoke it synchronously or asynchronously. With [synchronous invocation \(p. 138\)](#), 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



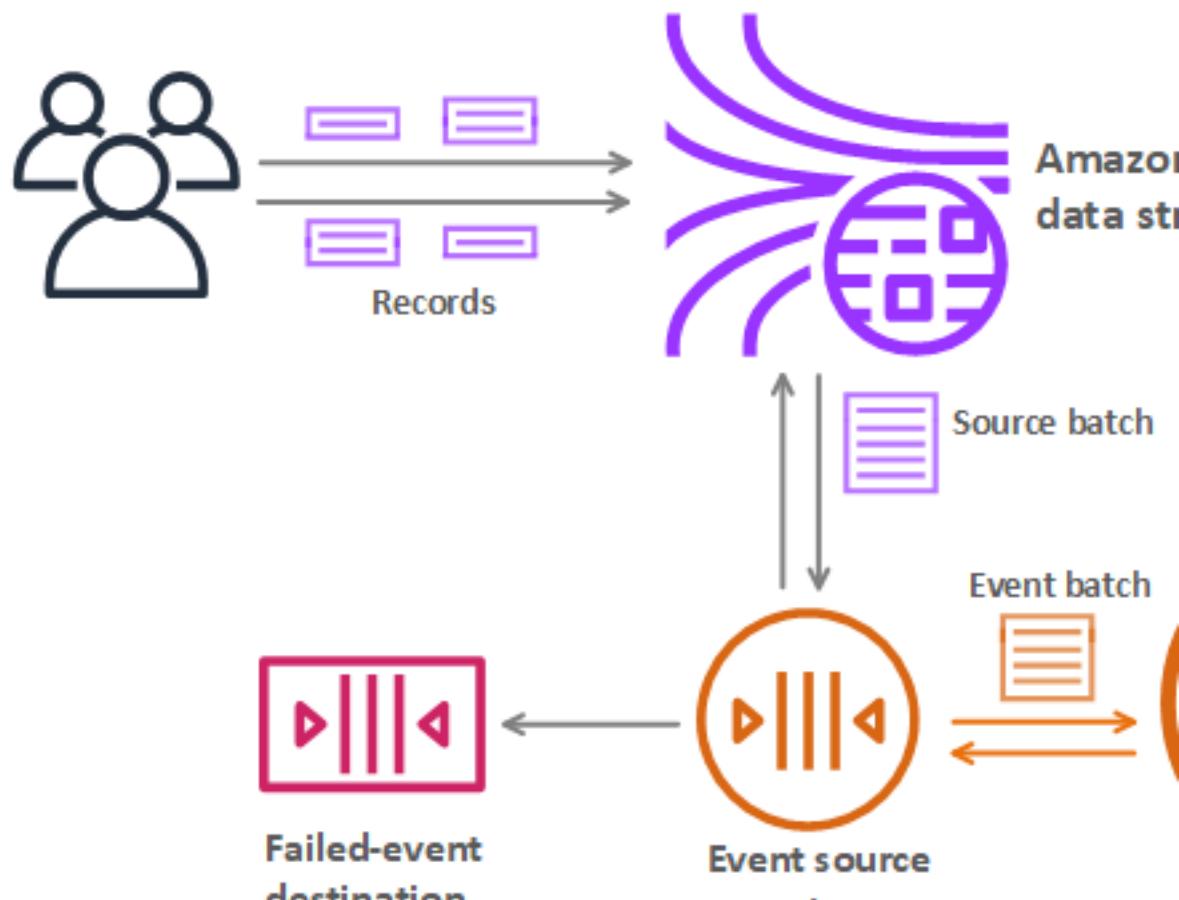
For asynchronous invocations, Lambda handles retries if the function returns an error or is throttled. To customize this behavior, you can configure error handling settings on a function, version, or alias. You can also configure Lambda to send events that failed processing to a dead-letter queue, or to send a record of any invocation to a [destination \(p. 38\)](#).

For more information, see [Asynchronous invocation \(p. 140\)](#).

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



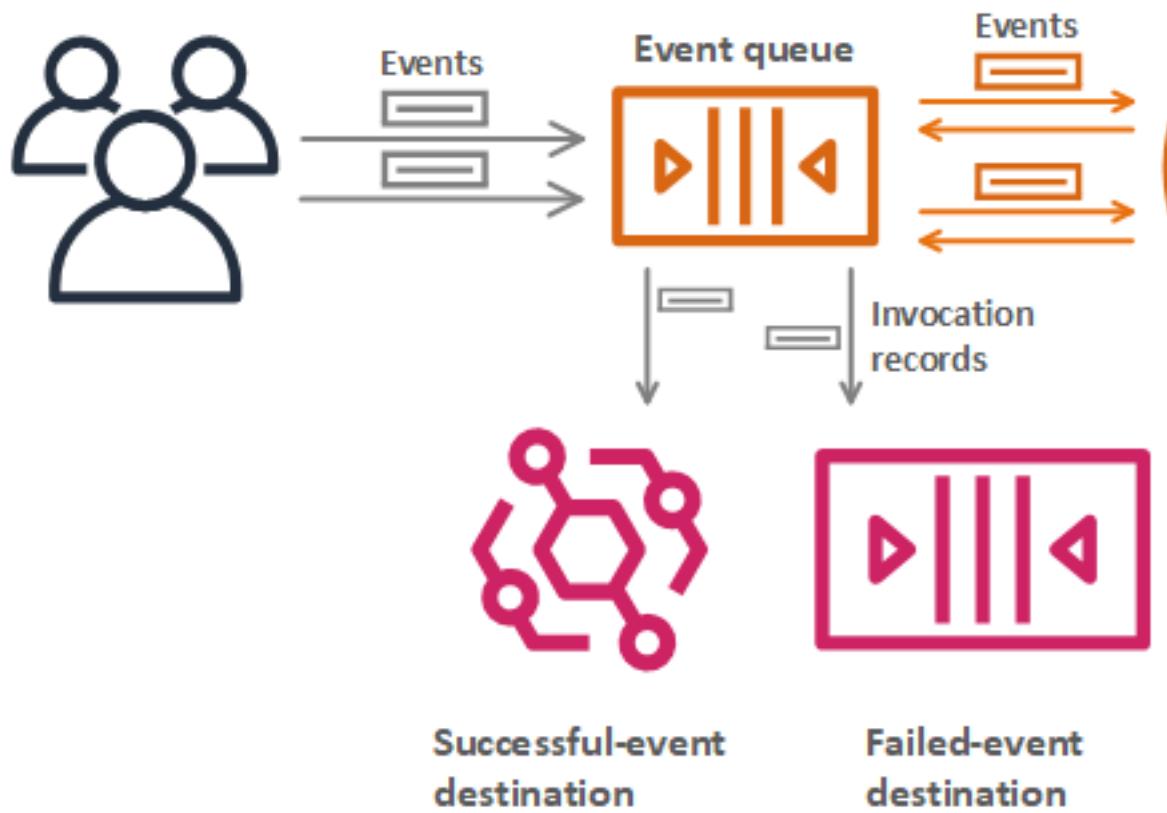
Event source mappings maintain a local queue of unprocessed items and handle retries if the function returns an error or is throttled. You can configure an event source mapping to customize batching behavior and error handling, or to send a record of items that fail processing to a destination.

For more information, see [AWS Lambda event source mappings \(p. 148\)](#).

Destinations

A destination is an AWS resource that receives invocation records for a function. For [asynchronous invocation \(p. 36\)](#), 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 \(p. 37\)](#) 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 \(p. 142\)](#) and the error handling sections of [Using AWS Lambda with Amazon DynamoDB \(p. 313\)](#) and [Using AWS Lambda with Amazon Kinesis \(p. 365\)](#).

Function blueprints

When you create a function in the Lambda console, you can choose to start from scratch, use a blueprint, use a [container image \(p. 41\)](#), or deploy an application from the [AWS Serverless Application Repository](#). 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 \(p. 41\)](#). You can use a rich tools ecosystem for authoring, building, and deploying your Lambda functions using AWS and popular community tools like the Docker command line interface (CLI).

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*.

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 \(p. 178\)](#).

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 \(p. 41\)](#)
- [.zip file archives \(p. 41\)](#)
- [Layers \(p. 42\)](#)
- [Using other AWS services to build a deployment package \(p. 42\)](#)

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 [Using container images with Lambda \(p. 247\)](#).

.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.

- Deploy Node.js Lambda functions with .zip file archives (p. 488)
- Deploy Python Lambda functions with .zip file archives (p. 514)
- Deploy Ruby Lambda functions with .zip file archives (p. 550)
- Deploy Java Lambda functions with .zip or JAR file archives (p. 577)
- Deploy Go Lambda functions with .zip file archives (p. 616)
- Deploy C# Lambda functions with .zip file archives (p. 644)
- Deploy PowerShell Lambda functions with .zip file archives (p. 668)

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 \(p. 101\)](#).

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, we recommend uploading 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 quotas

Lambda sets quotas for the amount of compute and storage resources that you can use to run and store functions. The following quotas apply per AWS Region and can be increased. 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.	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. 111)	250	Hundreds
Note This quota is shared with other services, such as Amazon Elastic File System (Amazon EFS). See Amazon VPC quotas .		

For details on concurrency and how Lambda scales your function concurrency in response to traffic, see [AWS Lambda function scaling \(p. 153\)](#).

The following quotas apply to function configuration, deployments, and execution. They cannot be changed.

Resource	Quota
Function memory allocation (p. 70)	128 MB to 10,240 MB, in 1-MB increments.
Function timeout (p. 70)	900 seconds (15 minutes)
Function environment variables (p. 77)	4 KB
Function resource-based policy (p. 51)	20 KB
Function layers (p. 101)	five layers
Function burst concurrency (p. 153)	500 - 3000 (varies per Region)
Invocation payload (p. 137) (request and response)	6 MB (synchronous) 256 KB (asynchronous)
Deployment package (.zip file archive) (p. 41) size	50 MB (zipped, for direct upload) 250 MB (unzipped, including layers) 3 MB (console editor)

Resource	Quota
Container image (p. 247) code package size	10 GB
Test events (console editor)	10
/tmp directory storage	512 MB
File descriptors	1,024
Execution processes/threads	1,024

The following quotas are associated with Lambda API requests.

Resource	Quota
Invocation requests per Region (requests per second)	10 x concurrent executions quota (synchronous (p. 138) , all sources) 10 x concurrent executions quota (asynchronous (p. 140) , non-AWS sources)
Invocation requests per Region (requests per second) for asynchronous AWS service sources (p. 257)	Unlimited requests accepted. Execution rate is based on concurrency available to the function. See Asynchronous invocation (p. 140) .
Invocation requests per function version or alias (requests per second)	10 x allocated provisioned concurrency (p. 84) Note This quota applies only to functions that use provisioned concurrency.
GetFunction (p. 807) API requests	100 requests per second
GetPolicy (p. 834) API requests	15 requests per second
Remainder of the control plane API requests (excludes invocation, GetFunction , and GetPolicy requests)	15 requests per second

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 \(p. 257\)](#).

AWS Lambda permissions

You can use AWS Identity and Access Management (IAM) to manage access to the Lambda API and resources like functions and layers. For users and applications in your account that use Lambda, you manage permissions in a permissions policy that you can apply to IAM users, groups, or roles. To grant permissions to other accounts or AWS services that use your Lambda resources, you use a policy that applies to the resource itself.

A Lambda function also has a policy, called an [execution role \(p. 47\)](#), that grants it permission to access AWS services and resources. At a minimum, your function needs access to Amazon CloudWatch Logs for log streaming. If you [use AWS X-Ray to trace your function \(p. 450\)](#), or your function accesses services with the AWS SDK, you grant it permission to call them in the execution role. Lambda also uses the execution role to get permission to read from event sources when you use an [event source mapping \(p. 148\)](#) to trigger your function.

Note

If your function needs network access to a resource like a relational database that isn't accessible through AWS APIs or the internet, [configure it to connect to your VPC \(p. 111\)](#).

Use [resource-based policies \(p. 51\)](#) to give other accounts and AWS services permission to use your Lambda resources. Lambda resources include functions, versions, aliases, and layer versions. Each of these resources has a permissions policy that applies when the resource is accessed, in addition to any policies that apply to the user. When an AWS service like Amazon S3 calls your Lambda function, the resource-based policy gives it access.

To manage permissions for users and applications in your accounts, [use the managed policies that Lambda provides \(p. 56\)](#), or write your own. The Lambda console uses multiple services to get information about your function's configuration and triggers. You can use the managed policies as-is, or as a starting point for more restrictive policies.

You can restrict user permissions by the resource an action affects and, in some cases, by additional conditions. For example, you can specify a pattern for the Amazon Resource Name (ARN) of a function that requires a user to include their user name in the name of functions that they create. Additionally, you can add a condition that requires that the user configure functions to use a specific layer to, for example, pull in logging software. For the resources and conditions that are supported by each action, see [Resources and Conditions \(p. 61\)](#).

For more information about IAM, see [What is IAM?](#) in the *IAM User Guide*.

For more information about applying security principles to Lambda applications, see [Security](#) in the *Lambda operator guide*.

Topics

- [AWS Lambda execution role \(p. 47\)](#)
- [Using resource-based policies for AWS Lambda \(p. 51\)](#)
- [Identity-based IAM policies for Lambda \(p. 56\)](#)
- [Resources and conditions for Lambda actions \(p. 61\)](#)
- [Using permissions boundaries for AWS Lambda applications \(p. 67\)](#)

AWS 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. You provide this role when you create a function, and Lambda assumes the role when your function is invoked. You can create an execution role for development that has permission to send logs to Amazon CloudWatch and to upload trace data to AWS X-Ray.

To view a function's execution role

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Permissions**.
4. Under **Resource summary**, view the services and resources that the function can access. The following example shows the CloudWatch Logs permissions that Lambda adds to an execution role when you create it in the Lambda console.

The screenshot shows the 'Resource summary' section of the Lambda console. It lists 'Amazon CloudWatch Logs' with '3 actions, 2 resources'. Below this, there are two rows of permissions:

Resource	Actions
arn:aws:logs:us-east-2:123456789012:*	Allow: log
arn:aws:logs:us-east-2:123456789012:log-group:/aws/lambda/my-function:*	Allow: log Allow: log

Information: Lambda obtained this information from the following policy statements:

- Managed policy AWSLambdaBasicExecutionRole-493eae43-bffa-xmpl-9e86-cf39eeae586c,
- Managed policy AWSLambdaBasicExecutionRole-493eae43-bffa-xmpl-9e86-cf39eeae586c,

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, make an update to its code or configuration as well. This forces running instances of your function, which have out-of-date credentials, to stop and be replaced.

Topics

- [Creating an execution role in the IAM console \(p. 48\)](#)
- [Grant least privilege access to your Lambda execution role \(p. 48\)](#)
- [Managing roles with the IAM API \(p. 49\)](#)
- [AWS managed policies for Lambda features \(p. 50\)](#)

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 \(p. 9\)](#). 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 **Common use cases**, choose **Lambda**.
4. Choose **Next: Permissions**.
5. Under **Attach permissions policies**, choose the AWS managed policies **AWSLambdaBasicExecutionRole** and **AWSXRayDaemonWriteAccess**.
6. Choose **Next: Tags**.
7. Choose **Next: Review**.
8. For **Role name**, enter **lambda-role**.
9. 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, best practice is to adjust the policy to include only the required permissions. For more information, see [granting least privilege](#).

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.

To learn more, 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.

```
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"  
                }  
            ]  
        }  
    }  
}
```

The `trust-policy.json` file is a JSON file in the current directory that defines the [trust policy](#) for the role. 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 `AssumeRole` action.

Example `trust-policy.json`

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

You can also 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"}]}'
```

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
```

AWS managed policies for Lambda features

The following AWS managed policies provide permissions that are required to use Lambda features:

- **AWSLambdaBasicExecutionRole** – Permission to upload logs to CloudWatch.
- **AWSLambdaDynamoDBExecutionRole** – Permission to read records from an Amazon DynamoDB stream.
- **AWSLambdaKinesisExecutionRole** – Permission to read events from an Amazon Kinesis data stream or consumer.
- **AWSLambdaMQExecutionRole** – Permission to read records from an Amazon MQ broker.
- **AWSLambdaMSKExecutionRole** – Permission to read records from an Amazon Managed Streaming for Apache Kafka (Amazon MSK) cluster.
- **AWSLambdaSQSQueueExecutionRole** – Permission to read a message from an Amazon Simple Queue Service (Amazon SQS) queue.
- **AWSLambdaVPCAccessExecutionRole** – Permission to manage elastic network interfaces to connect your function to a virtual private cloud (VPC).
- **AWSXRayDaemonWriteAccess** – Permission to upload trace data to X-Ray.
- **CloudWatchLambdaInsightsExecutionRolePolicy** – Permission to write runtime metrics to CloudWatch Lambda Insights.

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 \(p. 148\)](#) 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. You can use event source mappings with the following services:

Services that Lambda reads events from

- [Amazon DynamoDB \(p. 313\)](#)
- [Amazon Kinesis \(p. 365\)](#)
- [Amazon MQ \(p. 389\)](#)
- [Amazon Managed Streaming for Apache Kafka \(p. 395\)](#)
- [self-managed Apache Kafka \(p. 356\)](#)
- [Amazon Simple Queue Service \(p. 438\)](#)

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](#).

Using resource-based policies for AWS Lambda

AWS Lambda supports resource-based permissions policies for Lambda functions and layers. Resource-based policies let you grant usage permission to other AWS accounts 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 \(p. 53\)](#) to invoke or manage a function. You can add multiple statements to grant access to several accounts, or let any account invoke your function. You can also use the policy to [grant invoke permission to an AWS service \(p. 52\)](#) that invokes a function in response to activity in your account.

To view a function's resource-based policy

1. Open the [Functions page](#) on 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 \(p. 742\)](#) and [AddLayerVersionPermission \(p. 738\)](#) 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 \(p. 56\)](#).

Topics

- [Granting function access to AWS services \(p. 52\)](#)
- [Granting function access to other accounts \(p. 53\)](#)
- [Granting layer access to other accounts \(p. 54\)](#)
- [Cleaning up resource-based policies \(p. 55\)](#)

Granting function access to AWS services

When you [use an AWS service to invoke your function \(p. 257\)](#), 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": "sns.amazonaws.com", "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 other accounts

To grant permissions to another AWS account, specify the account ID as the principal. The following example grants account 210987654321 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 210987654321 --output text
```

You should see the following output:

```
{"Sid": "xaccount", "Effect": "Allow", "Principal": "arn:aws:iam::210987654321: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 \(p. 56\)](#) 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 \(p. 97\)](#) 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 \(p. 64\)](#). 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 currently support cross-account actions for all of its APIs via resource-based policies. The following APIs are supported:

- [Invoke \(p. 839\)](#)
- [GetFunction \(p. 807\)](#)
- [GetFunctionConfiguration \(p. 816\)](#)
- [UpdateFunctionCode \(p. 926\)](#)
- [DeleteFunction \(p. 783\)](#)

- [PublishVersion \(p. 882\)](#)
- [ListVersionsByFunction \(p. 874\)](#)
- [CreateAlias \(p. 746\)](#)
- [GetAlias \(p. 797\)](#)
- [ListAliases \(p. 847\)](#)
- [UpdateAlias \(p. 911\)](#)
- [DeleteAlias \(p. 774\)](#)
- [GetPolicy \(p. 834\)](#)
- [PutFunctionConcurrency \(p. 893\)](#)
- [DeleteFunctionConcurrency \(p. 787\)](#)
- [ListTags \(p. 872\)](#)
- [TagResource \(p. 907\)](#)
- [UntagResource \(p. 909\)](#)

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 \(p. 56\)](#).

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 210987654321 --version-number 1 --output text
```

You should see output similar to the following:

```
e210ffdc-e901-43b0-824b-5fcd0dd26d16 {"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::210987654321: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-eb847fdcaa0f0
```

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
```

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.

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.

Note

The AWS managed policies **AWSLambdaFullAccess** and **AWSLambdaReadOnlyAccess** will be [deprecated](#) on March 1, 2021. After this date, you cannot attach these policies to new IAM users. For more information, see the related [troubleshooting topic \(p. 708\)](#).

- **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 \(p. 56\)](#)
- [Layer development and use \(p. 59\)](#)
- [Cross-account roles \(p. 60\)](#)
- [Condition keys for VPC settings \(p. 60\)](#)

Function development

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>ListFunctions",  
                "lambda>ListTags",  
                "lambda:GetEventSourceMapping",  
                "lambda>ListEventSourceMappings",  
                "iam>ListRoles"  
            ],  
            "Resource": "intern-/*"  
        }  
    ]  
}
```

```

        "Resource": "*"
    },
    {
        "Sid": "DevelopFunctions",
        "Effect": "Allow",
        "NotAction": [
            "lambda:AddPermission",
            "lambda:PutFunctionConcurrency"
        ],
        "Resource": "arn:aws:lambd*a::function:intern-*"
    },
    {
        "Sid": "DevelopEventSourceMappings",
        "Effect": "Allow",
        "Action": [
            "lambda>DeleteEventSourceMapping",
            "lambda:UpdateEventSourceMapping",
            "lambda>CreateEventSourceMapping"
        ],
        "Resource": "*",
        "Condition": {
            "StringLike": {
                "lambda:FunctionArn": "arn:aws:lambd*a::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 \(p. 61\)](#) 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>ListFunctions",
            "lambda>ListTags",
            "lambda:GetEventSourceMapping",
            "lambda>ListEventSourceMappings",
            "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 \(p. 51\)](#) 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:)"
],
"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 an IAM 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 \(p. 51\)](#) 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" \(p. 113\)](#).

Resources and conditions for Lambda actions

You can restrict the scope of a user's permissions by specifying resources and conditions in an IAM policy. Each API action 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 API 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 account 123456789012 to invoke a function named my-function in the US West (Oregon) 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 \(p. 839\)](#)). For other actions, the action identifier is the operation name prefixed by `lambda:`.

Conditions are an optional policy element that applies additional logic to determine if an action is allowed. In addition to [common conditions](#) supported by all actions, 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. The following policy lets a user grant permission to 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](#) in the *IAM User Guide*.

Sections

- [Function resource names \(p. 62\)](#)
- [Function actions \(p. 64\)](#)
- [Event source mapping actions \(p. 66\)](#)
- [Layer actions \(p. 66\)](#)

Function resource names

You reference a Lambda function in a policy statement using an Amazon Resource Names (ARN). The format of a function ARN depends on whether you are referencing the whole function, a function [version \(p. 95\)](#), or an [alias \(p. 97\)](#).

When making Lambda API calls, users can specify a version or alias by passing a version ARN or alias ARN in the `FunctionName` parameter, or by setting a value in the `Qualifier` parameter. Lambda makes authorization decisions by comparing the resource element in the IAM policy with the `FunctionName` passed in the API calls.

You must use the correct function ARN types in your policies to achieve the results that you expect, especially in policies that deny access. We recommend that you follow the best practices for using Deny statements with functions.

Best practices for using Deny statements with functions

The following table summarizes the resources to use in Deny effects. In the **Resource** column, `MyFunction` is the name of the function, `:1` refers to version 1 of the function, and `MyAlias` is the name of a function alias.

Resource best practices

Policy objective	Resource		
Deny access to all versions of a function	MyFunction*		
Deny access to a specific alias	MyFunction:MyAlias and MyFunction		
Deny access to a specific version of a function	MyFunction:1 and MyFunction		

The following sections provide example policy statements for each of the policy objectives.

Note

You can use only identity-based policies to deny specific function resources. Currently, Lambda does not support the Deny effect in resource-based policies.

For the action list in a policy statement, you can add any of the [actions defined by Lambda](#) that act on a function resource.

Deny access to all function versions

The following identity-based policy statement denies access to the `lambda:GetFunctionConfiguration` action for all versions of the `my-function` function. The wildcard character at the end of the function ARN ensures that this policy applies to all version and alias ARNs.

Example Example identity-based policy

```
{
    "Version": "2020-07-20",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "lambda:GetFunctionConfiguration"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:my-function*"
        }
    ]
}
```

Deny access to a specific function alias

To deny access to a specific alias, you must specify both the alias ARN and the unqualified function ARN in the policy. This prevents users from accessing the specific alias by passing the unqualified ARN as the `FunctionName` and the alias as the `Qualifier`.

Note

If you create this type of policy, API calls need to refer to the unpublished version of the function by specifying a qualified ARN with the `$LATEST` suffix in the `FunctionName` parameter.

The following identity-based policy statement denies access to the `lambda:InvokeFunction` action in the `my-alias` alias of the `my-function` function.

Example Example identity-based policy

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "DenySpecificAlias",  
            "Effect": "Deny",  
            "Action": "lambda:InvokeFunction",  
            "Resource": [  
                "arn:aws:lambda:us-west-2:123456789012:function:my-function:my-alias",  
                "arn:aws:lambda:us-west-2:123456789012:function:my-function"  
            ]  
        }  
    ]  
}
```

Deny access to a specific function version

To deny access to a specific version, you must specify both the qualified ARN and the unqualified ARN in the policy. This prevents users from accessing the specific version by passing the unqualified ARN as the `FunctionName` and the version as the `Qualifier`.

Note

If you create this type of policy, API calls need to refer to the unpublished version of the function by specifying a qualified ARN with the `$LATEST` suffix in the `FunctionName` parameter.

The following identity-based policy statement denies access to the invoke action in version 1 of the `my-function` function.

Example Example identity-based policy

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "DenySpecificFunctionVersion",  
            "Effect": "Deny",  
            "Action": [  
                "lambda:InvokeFunction"  
            ],  
            "Resource": [  
                "arn:aws:lambda:us-west-2:123456789012:function:my-function:1",  
                "arn:aws:lambda:us-west-2:123456789012:function:my-function"  
            ]  
        }  
    ]  
}
```

Function actions

Actions that operate on a function can be restricted to a specific function by function, version, or alias ARN, as described in the following table. Actions that don't support resource restrictions can only be granted for all resources (*).

Functions

Action	Resource	Condition
AddPermission (p. 742) RemovePermission (p. 905)	Function Function version Function alias	<code>lambda:Principal</code>
Invoke (p. 839) Permission: <code>lambda:InvokeFunction</code>	Function Function version Function alias	<code>None</code>
CreateFunction (p. 762) UpdateFunctionConfiguration (p. 935)	Function	<code>lambda:CodeSigningConfigArn</code> <code>lambda:Layer</code> <code>lambda:VpcIds</code> <code>lambda:SubnetIds</code> <code>lambda:SecurityGroupIds</code>
CreateAlias (p. 746) DeleteAlias (p. 774) DeleteFunction (p. 783) DeleteFunctionConcurrency (p. 787) GetAlias (p. 797) GetFunction (p. 807) GetFunctionConfiguration (p. 816) GetPolicy (p. 834) ListAliases (p. 847) ListVersionsByFunction (p. 874) PublishVersion (p. 882) PutFunctionConcurrency (p. 893) UpdateAlias (p. 911) UpdateFunctionCode (p. 926)	Function	<code>None</code>
GetAccountSettings (p. 795) ListFunctions (p. 858) ListTags (p. 872) TagResource (p. 907)	*	<code>None</code>

Action	Resource	Condition
UntagResource (p. 909)		

Event source mapping actions

For event source mappings, delete and update permissions can be restricted 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 mappings

Action	Resource	Condition
DeleteEventSourceMapping (p. 778)	Event source mapping	<code>lambda:FunctionArn</code>
UpdateEventSourceMapping (p. 918)		
CreateEventSourceMapping (p. 753)	*	<code>lambda:FunctionArn</code>
GetEventSourceMapping (p. 802)	*	None
ListEventSourceMappings (p. 852)		

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

Note: the [GetLayerVersion \(p. 826\)](#) action also covers [GetLayerVersionByArn \(p. 829\)](#). Lambda does not support `GetLayerVersionByArn` as an IAM action.

Layers

Action	Resource	Condition
AddLayerVersionPermission (p. 738)	Layer version	None
RemoveLayerVersionPermission (p. 903)		
GetLayerVersion (p. 826)		
GetLayerVersionPolicy (p. 832)		
DeleteLayerVersion (p. 791)		
ListLayerVersions (p. 866)	Layer	None
PublishLayerVersion (p. 878)		
ListLayers (p. 864)	*	None

Using permissions boundaries for AWS Lambda applications

When you [create an application](#) (p. 178) 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. 47) 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 [??? \(p. 178\)](#).

For more information, see [Permissions boundaries for IAM entities](#) in the IAM User Guide.

Managing AWS Lambda functions

You can use the AWS Lambda API or console to create functions and configure function settings. [Basic function settings \(p. 70\)](#) include the description and the execution role that you specify when you create a function in the Lambda console. You can configure more settings after you create a function, or use the API to modify configuration settings such as the handler name, memory allocation, and security groups during creation.

To keep secrets out of your function code, store them in the function's configuration and read them from the execution environment during initialization. [Environment variables \(p. 77\)](#) are always encrypted at rest, and can be encrypted client-side as well. Use environment variables to make your function code portable by removing connection strings, passwords, and endpoints for external resources.

[Versions and aliases \(p. 95\)](#) are secondary resources that you can create to manage function deployment and invocation. Publish [versions \(p. 95\)](#) of your function to store its code and configuration as a separate resource that cannot be changed, and create an [alias \(p. 97\)](#) that points to a specific version. Then you can configure your clients to invoke a function alias, and update the alias when you want to point the client to a new version, instead of updating the client.

As you add libraries and other dependencies to your function, creating and uploading a deployment package can slow down development. Use [layers \(p. 101\)](#) to manage your function's dependencies independently and keep your deployment package small. You can also use layers to share your own libraries with other customers and use publicly available layers with your functions.

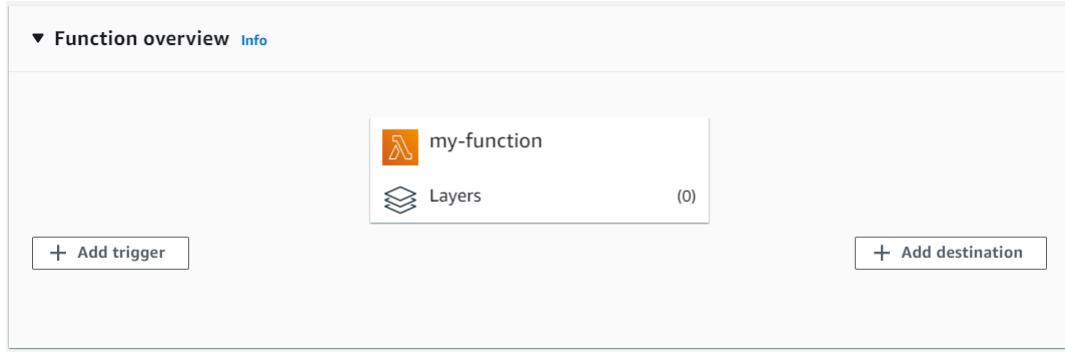
To use your Lambda function with AWS resources in an Amazon VPC, configure it with security groups and subnets to [create a VPC connection \(p. 111\)](#). Connecting your function to a VPC lets you access resources in a private subnet such as relational databases and caches. You can also [create a database proxy \(p. 121\)](#) for MySQL and Aurora DB instances. A database proxy enables a function to reach high concurrency levels without exhausting database connections.

To use [code signing \(p. 130\)](#) with your Lambda function, configure it with a code-signing configuration. When a user attempts to deploy a code package, Lambda checks that the code package has a valid signature from a trusted publisher. The code-signing configuration includes a set of signing profiles, which define the trusted publishers for this function.

Configuring functions in the console

You can use the Lambda console to configure function settings, add triggers and destinations, and update and test your code.

To manage a function, open the Lambda console [Functions page](#) and choose a function.



The function overview shows a visualization of your function and its upstream and downstream resources.

You can use it to configure triggers, layers, and destinations.

- **Triggers** – Triggers are services and resources that you have configured to invoke your function. Choose [Add trigger](#) to create a Lambda [event source mapping](#) (p. 148) or to configure a trigger in another service that the Lambda console integrates with. For details about these services and others, see [Using AWS Lambda with other services](#) (p. 257).
- **Layers** – Choose the **Layers** node to add [layers](#) (p. 101) to your application. A layer is a .zip file archive that contains libraries, a custom runtime, or other dependencies.
- **Destinations** – Add a destination to your function to send details about invocation results to another service. You can send invocation records when your function is invoked [asynchronously](#) (p. 140), or by an [event source mapping](#) (p. 148) that reads from a stream.

By default you are working with the `Latest` function version. You can modify the following settings in this version:

Function settings

- **Code** – The code and dependencies of your function. For scripting languages, you can edit your function code in the embedded [editor](#) (p. 17). To add libraries, or for languages that the editor doesn't support, or to create a function deployed as a container image, upload a [deployment package](#) (p. 41). If your deployment package is larger than 50 MB, choose [Upload a file from Amazon S3](#).
- **Runtime** – The [Lambda runtime](#) (p. 195) that runs your function.
- **Handler** – The method that the runtime runs when your function is invoked, such as `index.handler`. The first value is the name of the file or module. The second value is the name of the method.
- **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. 77).
- **Tags** – Key-value pairs that Lambda attaches to your function resource. [Use tags](#) (p. 134) 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.

- **Execution role** – The [AWS Identity and Access Management \(IAM\) role](#) (p. 47) that Lambda assumes when it runs your function.

- **Description** – A description of the function.
- **Memory** – The amount of memory available to the function at runtime. To [set the memory for your function](#), enter a value between 128 MB and 10,240 MB 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.
- **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 \(VPC\) \(p. 111\)](#).
- **Database proxies** – [Create a database proxy \(p. 121\)](#) for functions that use an Amazon RDS DB instance or cluster.
- **Active tracing** – Sample incoming requests and [trace sampled requests with AWS X-Ray \(p. 450\)](#).
- **Concurrency** – [Reserve concurrency for a function \(p. 84\)](#) 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.
- **Asynchronous invocation** – [Configure error handling behavior \(p. 140\)](#) 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 \(p. 145\)](#) to retain discarded events.

You can configure error handling settings on a function, version, or alias.

Except as noted in the preceding list, you can only change function settings on the unpublished version of a function. When you publish a version, code and most settings are locked to ensure a consistent experience for users of that version. Use [aliases \(p. 97\)](#) to propagate configuration changes in a controlled manner.

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

- [UpdateFunctionCode \(p. 926\)](#) – Update the function's code.
- [UpdateFunctionConfiguration \(p. 935\)](#) – Update version-specific settings.
- [TagResource \(p. 907\)](#) – Tag a function.
- [AddPermission \(p. 742\)](#) – Modify the [resource-based policy \(p. 51\)](#) of a function, version, or alias.
- [PutFunctionConcurrency \(p. 893\)](#) – Configure a function's reserved concurrency.
- [PublishVersion \(p. 882\)](#) – Create an immutable version with the current code and configuration.
- [CreateAlias \(p. 746\)](#) – Create aliases for function versions.
- [PutFunctionEventInvokeConfig](#) – Configure error handling for asynchronous invocation.

For example, to update a function's memory setting with the AWS CLI, use the `update-function-configuration` command.

```
aws lambda update-function-configuration --function-name my-function --memory-size 256
```

For function configuration best practices, see [Function configuration \(p. 193\)](#).

Configuring functions defined as container images

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 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 \(p. 816\)](#) API operation. To update the function to a newer image version, you must use the Lambda console to [update the function code \(p. 74\)](#), or use the [UpdateFunctionCode \(p. 926\)](#) API operation. Configuration operations such as [UpdateFunctionConfiguration \(p. 935\)](#) do not update the function's container image.

Note

In Amazon ECR, if you reassign the image tag to another image, Lambda does not update the image version.

Topics

- [Function version \\$LATEST \(p. 72\)](#)
- [Container image deployment \(p. 72\)](#)
- [Amazon ECR permissions \(p. 72\)](#)
- [Override the container settings \(p. 73\)](#)
- [Creating a function \(console\) \(p. 73\)](#)
- [Updating the function code \(console\) \(p. 74\)](#)
- [Overriding the image parameters \(console\) \(p. 74\)](#)
- [Using the Lambda API \(p. 75\)](#)
- [AWS CloudFormation \(p. 75\)](#)

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 \(p. 95\)](#).

Container image deployment

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.

Amazon ECR permissions

For your function to access 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*.

Override the 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 \(p. 249\)](#).

Creating a function (console)

To create a function defined as a container image, you must first [create the image \(p. 248\)](#) and then store the image in the Amazon ECR repository.

To create the function

1. Open the [Functions page](#) on 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**, 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.
 - c. (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 ("").

5. (Optional) Under **Permissions**, expand **Change default execution role**. Then, choose to create a new **Execution role**, or to use an existing role.
6. Choose **Create function**.

Updating the function code (console)

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 \(p. 248\)](#), 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](#) on 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**.

Overriding the image parameters (console)

You can use the Lambda console to override the configuration values in the container image.

To override the configuration values in the container image

1. Open the [Functions page](#) on 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” \(p. 77\)](#).

Using the Lambda API

To manage functions defined as container images, use the following API operations:

- [CreateFunction \(p. 762\)](#)
- [UpdateFunctionCode \(p. 926\)](#)
- [UpdateFunctionConfiguration \(p. 935\)](#)

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. Note that you must create the function from the same account as the container registry in Amazon EFS.

```
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*.

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 are 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 \(p. 77\)](#)
- [Configuring environment variables with the API \(p. 78\)](#)
- [Example scenario for environment variables \(p. 78\)](#)
- [Retrieve environment variables \(p. 79\)](#)
- [Defined runtime environment variables \(p. 80\)](#)
- [Securing environment variables \(p. 81\)](#)
- [Sample code and templates \(p. 83\)](#)

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 \(p. 70\)](#).

You create an environment variable on 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](#) on 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 \(p. 80\)](#).
- 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 \(p. 935\)](#)
- [GetFunctionConfiguration \(p. 816\)](#)
- [CreateFunction \(p. 762\)](#)

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": "nodejs12.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	rd1owwlydynam5.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 \(p. 81\)](#), 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 \(p. 195\)](#) 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 \(p. 450\)](#).
- `AWS_REGION` – The AWS Region where the Lambda function is executed.
- `AWS_EXECUTION_ENV` – The [runtime identifier \(p. 195\)](#), prefixed by `AWS_Lambda_`—for example, `AWS_Lambda_java8`.
- `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 either on-demand or provisioned-concurrency. For information, see [Configuring provisioned concurrency \(p. 87\)](#).

- `AWS_LAMBDA_LOG_GROUP_NAME`, `AWS_LAMBDA_LOG_STREAM_NAME` – The name of the Amazon CloudWatch Logs group and stream for the function.
- `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_SESSION_TOKEN` – The access keys obtained from the function's [execution role \(p. 47\)](#).
- `AWS_LAMBDA_RUNTIME_API` – ([Custom runtime \(p. 236\)](#)) The host and port of the [runtime API \(p. 205\)](#).
- `LAMBDA_TASK_ROOT` – The path to your Lambda function code.
- `LAMBDA_RUNTIME_DIR` – The path to runtime libraries.
- `TZ` – The environment's time zone (UTC). The execution environment uses NTP to synchronize the system clock.

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 \(p. 483\)](#)) 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 \(p. 509\)](#)) The Python library path (`$LAMBDA_RUNTIME_DIR`).

- **GEM_PATH** – ([Ruby \(p. 546\)](#)) 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 \(p. 87\)](#).

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

Securing environment variables

Lambda encrypts environment variables with a key that it creates in your account (an AWS managed customer master key (CMK)). Use of this key is free. You can also choose to provide your own key for Lambda to use instead of the default key.

When you provide the key, only users in your account with access to the key can view or manage environment variables on the function. Your organization might also have internal or external requirements to manage keys that are used for encryption and to control when they're rotated.

To use a customer managed CMK

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Configuration**, then choose **Environment variables**.
4. Under **Environment variables**, choose **Edit**.
5. Expand **Encryption configuration**.
6. Choose **Use a customer master key**.
7. Choose your customer managed CMK.
8. Choose **Save**.

Customer managed CMKs incur standard [AWS KMS charges](#).

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 CMK, 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.

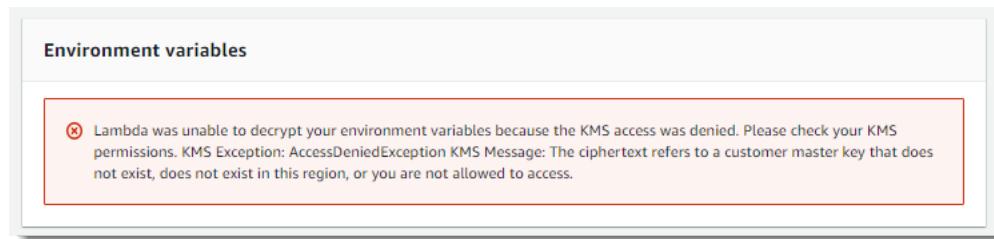
- `kms>ListAliases` – To view keys in the Lambda console.
- `kms>CreateGrant, kms:Encrypt` – To configure a customer managed CMK on a function.
- `kms:Decrypt` – To view and manage environment variables that are encrypted with a customer managed CMK.

You can get these permissions from your user account or from a key's resource-based permissions policy. `ListAliases` is provided by the [managed policies for Lambda \(p. 56\)](#). 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](#).

You can also encrypt environment variable values on the client side before sending them to Lambda, and decrypt them in your function code. This obscures secret values in the Lambda console and API output, even for users who have permission to use the key. In your code, you retrieve the encrypted value from the environment and decrypt it by using the AWS KMS API.

To encrypt environment variables on the client side

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Configuration**, then choose **Environment variables**.
4. Under **Environment variables**, choose **Edit**.
5. Expand **Encryption configuration**.
6. Choose **Enable helpers for encryption in transit**.
7. Choose **Encrypt** next to a variable to encrypt its value.
8. Choose **Save**.

Note

When you use the console encryption helpers, your function needs permission to call the `kms:Decrypt` API operation in its [execution role \(p. 47\)](#).

To view sample code for your function's language, choose **Code** next to an environment variable. The sample code shows how to retrieve an environment variable in a function and decrypt its value.

Another option is to store passwords in AWS Secrets Manager secrets. You can reference the secret in your AWS CloudFormation templates to set passwords on databases. You can also set the value of an environment variable on the Lambda function. For an example, see the next section.

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 \(p. 465\)](#) – Create a function and an Amazon SNS topic in the same template. Pass the name of the topic to the function in an environment variable. Read environment variables in code (multiple languages).
- [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.

Managing concurrency for a Lambda function

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. Concurrency is subject to a Regional [quota \(p. 44\)](#) that is shared by all functions in a Region.

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 and provisioned concurrency. To learn about how concurrency interacts with scaling, [see Lambda function scaling](#).

To ensure that a function can always reach a certain level of concurrency, configure the function with [reserved concurrency \(p. 85\)](#). When a function has reserved concurrency, no other function can use that concurrency. Reserved concurrency also limits the maximum concurrency for the function, and applies to the function as a whole, including versions and aliases.

When Lambda allocates an instance of your function, the [runtime \(p. 195\)](#) 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. As your function [scales up \(p. 153\)](#), this causes the portion of requests that are served by new instances to have higher latency than the rest.

To enable your function to scale without fluctuations in latency, use [provisioned concurrency \(p. 87\)](#). By allocating provisioned concurrency before an increase in invocations, you can ensure that all requests are served by initialized instances with very low latency. You can configure provisioned concurrency on a version of a function, or on an alias.

Lambda also integrates with [Application Auto Scaling](#). You can configure Application Auto Scaling to manage provisioned concurrency on a schedule or based on utilization. Use scheduled scaling to increase provisioned concurrency in anticipation of peak traffic. To increase provisioned concurrency automatically as needed, use [the Application Auto Scaling API \(p. 90\)](#) to register a target and create a scaling policy.

Provisioned concurrency counts towards a function's reserved concurrency and Regional quotas. 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.

Note

You can't allocate more provisioned concurrency than reserved concurrency for a function.

Sections

- [Configuring reserved concurrency \(p. 85\)](#)
- [Configuring provisioned concurrency \(p. 87\)](#)
- [Configuring concurrency with the Lambda API \(p. 90\)](#)

Configuring reserved concurrency

To manage reserved concurrency settings for a function, use the Lambda console.

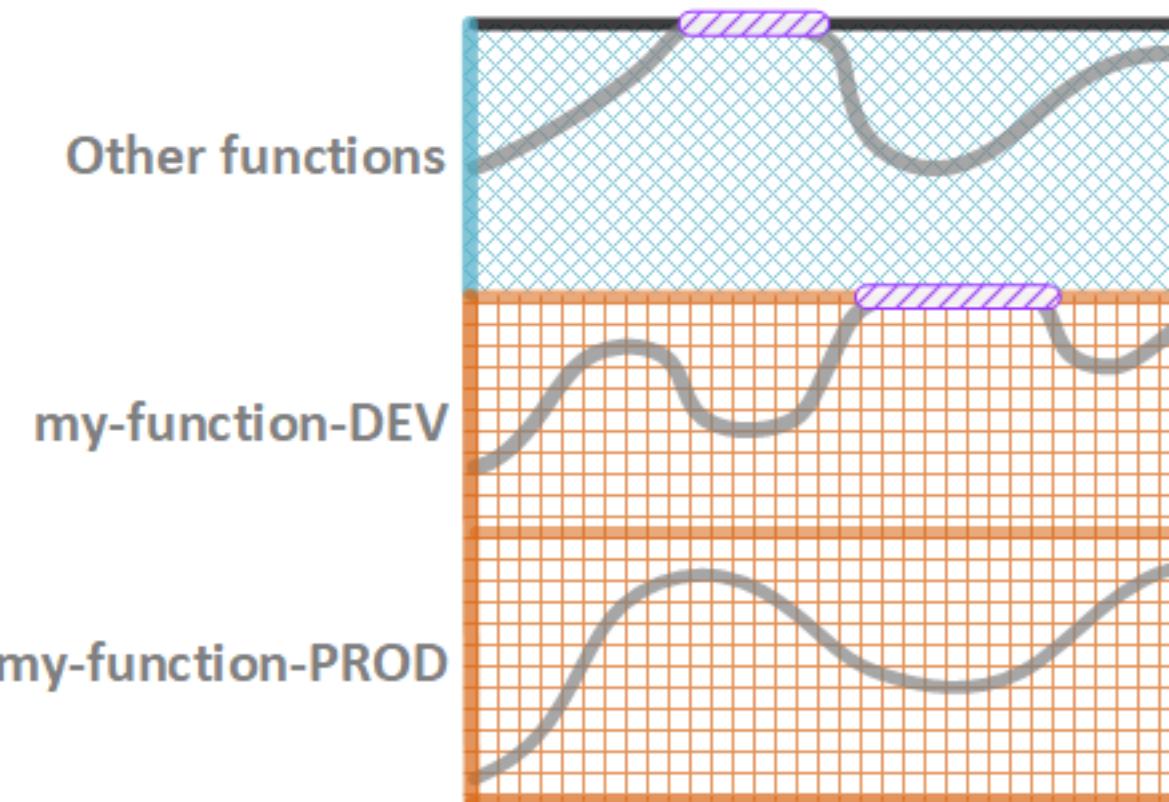
To reserve concurrency for a function

1. Open the [Functions page](#) on 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 provisioned concurrency

To manage provisioned concurrency settings for a version or alias, use the Lambda console.

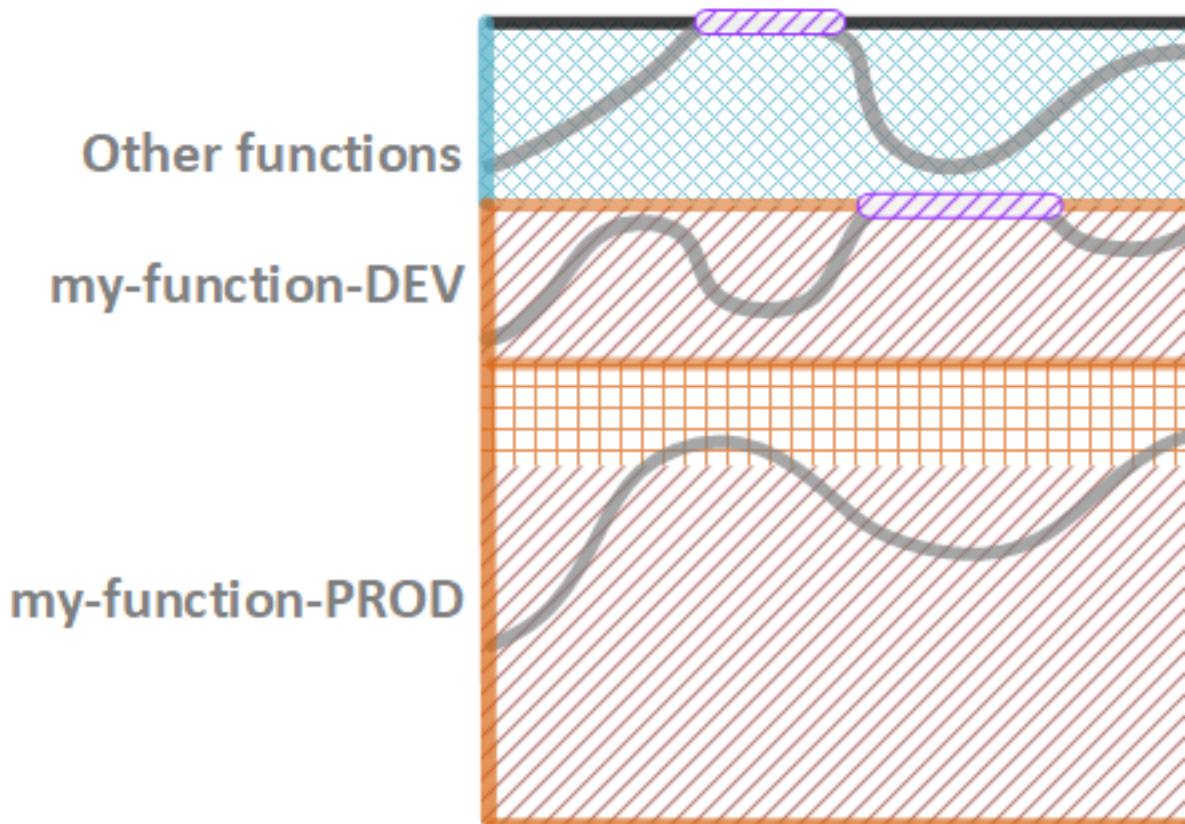
To provision concurrency for an alias

1. Open the [Functions page](#) on 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 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.

In the following example, the `my-function-DEV` and `my-function-PROD` functions are configured with both reserved and provisioned concurrency. For `my-function-DEV`, the full pool of reserved concurrency is also provisioned concurrency. In this case, all invocations either run on provisioned concurrency or are throttled. For `my-function-PROD`, a portion of the reserved concurrency pool is standard concurrency. When all provisioned concurrency is in use, the function scales on standard concurrency to serve any additional requests.

Provisioned Concurrency with Reserved Concurrency

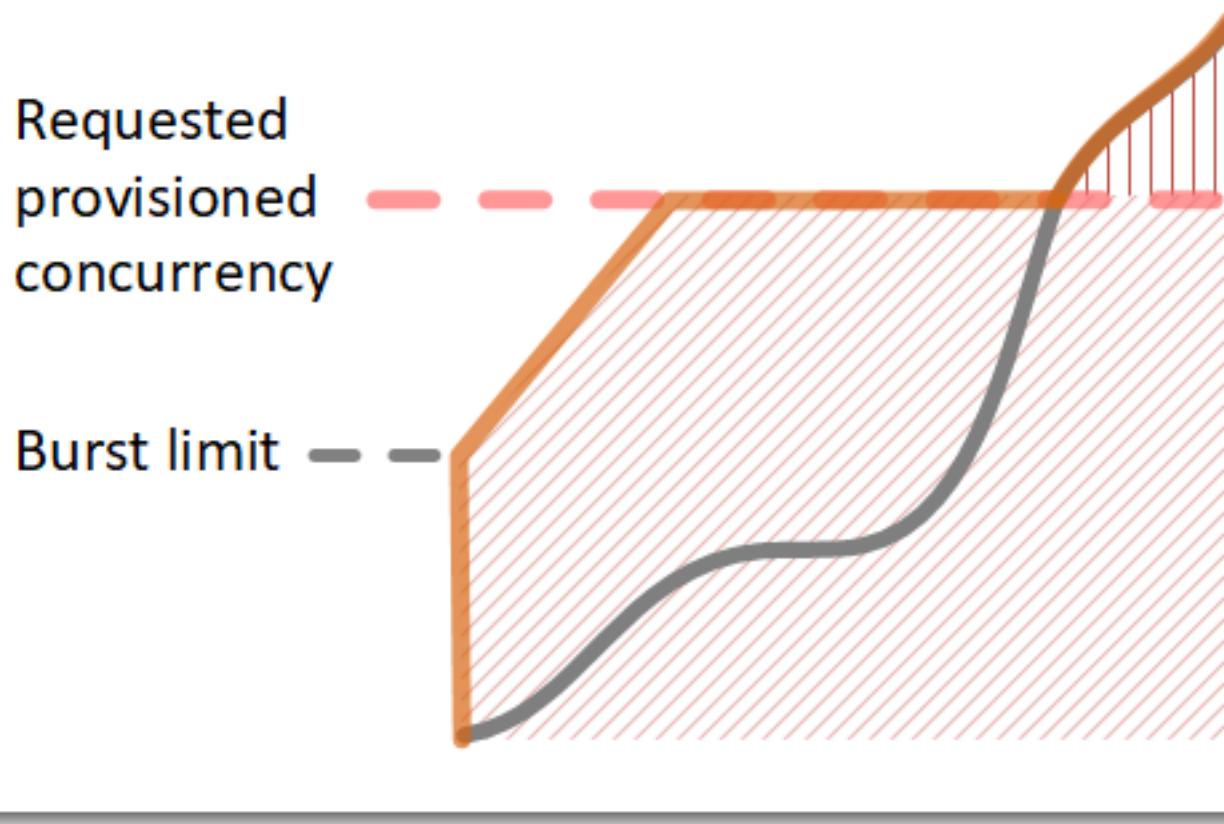


Legend

- Function concurrency
- Reserved concurrency
- Provisioned concurrency
- Unreserved concurrency
- Throttling

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 \(p. 153\)](#), 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



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 \(p. 32\)](#) 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 \(p. 450\)](#) 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](#).

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. Also, you can't allocate provisioned concurrency on an alias that points to the unpublished version (\$LATEST).

When you change the version that an alias points to, provisioned concurrency is deallocated from the old version and then allocated to the new version. You can add a routing configuration to an alias that has provisioned concurrency. However, you can't manage provisioned concurrency settings on the alias while the routing configuration is in place.

Lambda emits the following metrics for provisioned concurrency:

Provisioned concurrency metrics

- ProvisionedConcurrentExecutions
- ProvisionedConcurrencyInvocations
- ProvisionedConcurrencySpilloverInvocations
- ProvisionedConcurrencyUtilization

For details, see [Working with AWS Lambda function metrics \(p. 691\)](#).

Configuring concurrency with the Lambda API

To manage concurrency settings and autoscaling with the AWS CLI or AWS SDK, use the following API operations.

- [PutFunctionConcurrency \(p. 893\)](#)
- [GetFunctionConcurrency](#)
- [DeleteFunctionConcurrency \(p. 787\)](#)
- [PutProvisionedConcurrencyConfig](#)
- [GetProvisionedConcurrencyConfig](#)
- [ListProvisionedConcurrencyConfigs](#)
- [DeleteProvisionedConcurrencyConfig](#)

- [GetAccountSettings \(p. 795\)](#)
- (Application Auto Scaling) [RegisterScalableTarget](#)
- (Application Auto Scaling) [PutScalingPolicy](#)

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  
}
```

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 configure Application Auto Scaling to manage provisioned concurrency, use the Application Auto Scaling to configure [target tracking scaling](#). First, 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-scalable-target --service-namespace lambda \  
--resource-id function:my-function:BLUE --min-capacity 1 --max-capacity 100 \  
--scalable-dimension lambda:function:ProvisionedConcurrency
```

Next, 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:

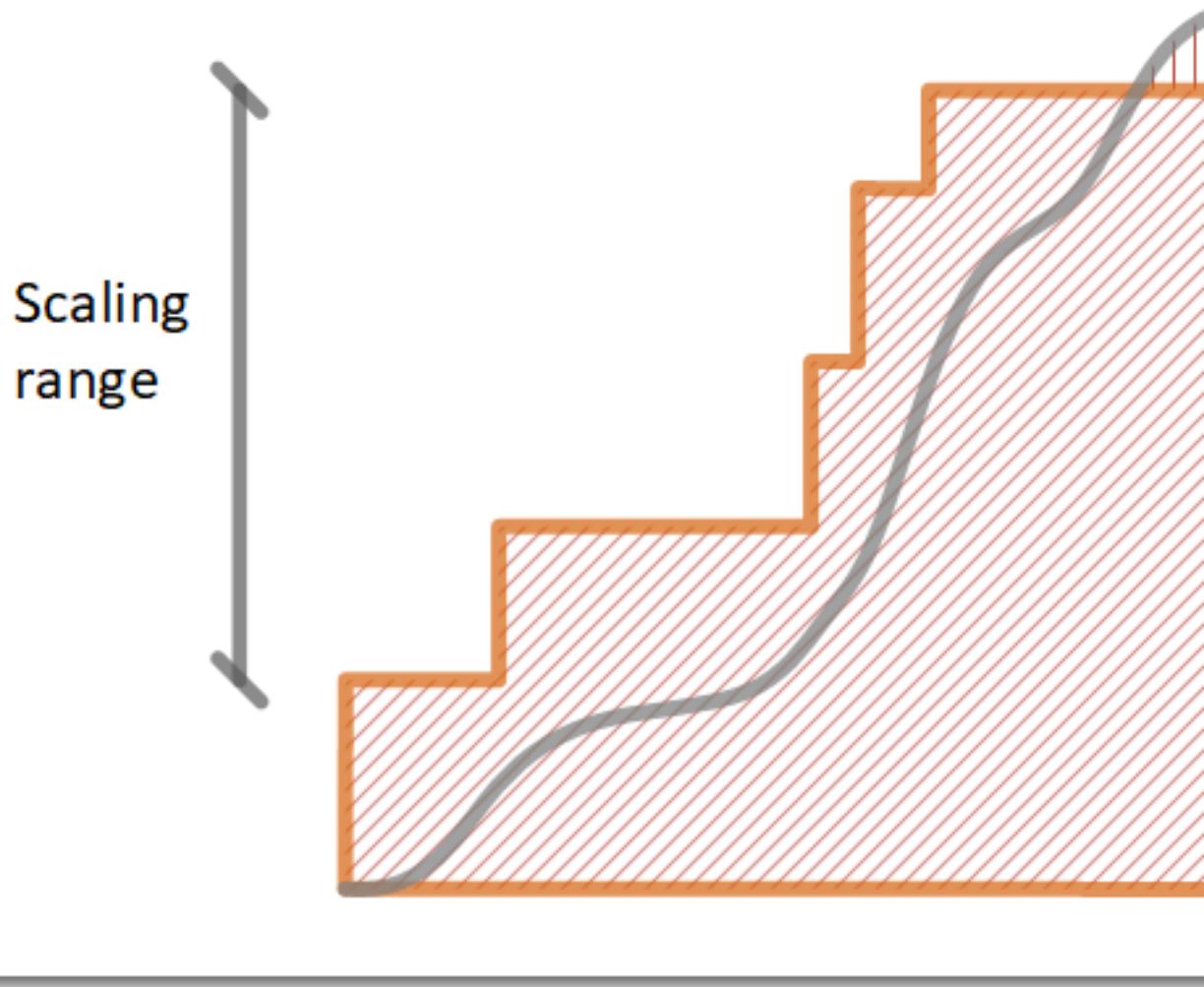
```
{
```

```
"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

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  
    }  
}
```

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.

A function version includes the following information:

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

Sections

- [Creating function versions \(p. 95\)](#)
- [Managing versions with the Lambda API \(p. 95\)](#)
- [Using versions \(p. 96\)](#)
- [Granting permissions \(p. 96\)](#)

Creating function versions

You can change the function code and settings only on the unpublished version of a function. When you publish a version, the code and most of the settings are locked to maintain a consistent experience for users of that version. For more information about configuring function settings, see [Configuring functions in the console \(p. 70\)](#).

You can create a function version using the Lambda console.

To create a new function version

1. Open the [Functions page](#) on 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**.

Managing versions with the Lambda API

To publish a version of a function, use the [PublishVersion \(p. 882\)](#) API operation.

The following example 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": "nodejs12.x",
...
}
```

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 \(p. 148\)](#). 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 \(p. 51\)](#) or an [identity-based policy \(p. 56\)](#) 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 \(p. 61\)](#).

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 \(p. 97\)](#).

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 \(Console\) \(p. 97\)](#)
- [Managing aliases with the Lambda API \(p. 97\)](#)
- [Using aliases \(p. 98\)](#)
- [Resource policies \(p. 98\)](#)
- [Alias routing configuration \(p. 98\)](#)

Creating a function alias (Console)

You can create a function alias using the Lambda console.

To create an alias

1. Open the [Functions page](#) on 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 \(p. 98\)](#).
 - 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 \(p. 746\)](#)
- [UpdateAlias \(p. 911\)](#)
- [DeleteAlias \(p. 774\)](#)

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 [AWS Lambda event source mappings \(p. 148\)](#).

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 \(p. 51\)](#) 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 \(p. 61\)](#).

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.

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 \(p. 47\)](#).
- Both versions must have the same [dead-letter queue](#) 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 \(p. 95\)](#).

1. Open the [Functions page](#) on 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

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 \(p. 746\)](#)
- [UpdateAlias \(p. 911\)](#)

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 AWS Lambda function metrics \(p. 691\)](#).

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

Creating and sharing Lambda layers

A Lambda layer is a .zip file archive that can contain additional code or data. A layer can contain libraries, a [custom runtime \(p. 236\)](#), 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 \(p. 41\)](#). For functions [defined as a container image \(p. 247\)](#), 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*.

Sections

- [Creating layer content \(p. 101\)](#)
- [Compiling the .zip file archive for your layer \(p. 101\)](#)
- [Including library dependencies in a layer \(p. 102\)](#)
- [Creating a layer \(p. 103\)](#)
- [Deleting a layer version \(p. 104\)](#)
- [Configuring layer permissions \(p. 104\)](#)
- [Using AWS CloudFormation with layers \(p. 105\)](#)

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.

Using the AWS CLI

You can create and upload a .zip file for your layer using the Lambda console or the AWS Command Line Interface (AWS CLI). For language-specific instructions, see the following topics.

- [Deploy Node.js Lambda functions with .zip file archives \(p. 488\)](#)
- [Deploy Python Lambda functions with .zip file archives \(p. 514\)](#)
- [Deploy Ruby Lambda functions with .zip file archives \(p. 550\)](#)
- [Deploy Java Lambda functions with .zip or JAR file archives \(p. 577\)](#)
- [Deploy Go Lambda functions with .zip file archives \(p. 616\)](#)
- [Deploy C# Lambda functions with .zip file archives \(p. 644\)](#)
- [Deploy PowerShell Lambda functions with .zip file archives \(p. 668\)](#)

Compiling the .zip file archive for your layer

You build your layer code into a .zip file archive using the same procedure as you would for a function. 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 \(p. 195\)](#).

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 \(p. 195\)](#), 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)		
Python	python		
	python/lib/python3.8/site-packages(site directories)		
Java	java/lib (CLASSPATH)		
Ruby	ruby/gems/2.5.0 (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/2.5.0/
| 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 \(p. 80\)](#).

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 runtimes**, choose up to 15 runtimes.
7. (Optional) For **License**, enter any necessary license information.
8. Choose **Create**.

To create a layer (API)

To create a layer, use the **publish-layer-version** command with a name, description, .zip file archive, and list of [runtimes \(p. 195\)](#) that are compatible with the layer. The list of runtimes is optional.

```
aws lambda publish-layer-version --layer-name my-layer --description "My layer" \
--license-info "MIT" --content S3Bucket=lambda-layers-us-
east-2-123456789012,S3Key=layer.zip \
--compatible-runtimes python3.6 python3.7 python3.8
```

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": "tv9jJO+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,  
    "LicenseInfo": "MIT",  
    "CompatibleRuntimes": [  
        "python3.6",  
        "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 210987654321 --version-number 1 --output text
```

You should see output similar to the following:

```
e210ffdc-e901-43b0-824b-5fcd0dd26d16 {"Sid":"xaccount","Effect":"Allow","Principal": {"AWS":"arn:aws:iam::210987654321:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:lambda:eu-central-1:123456789012: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 \(p. 54\)](#).

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: nodejs12.x
      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::Serverless::LayerVersion
    Properties:
      LayerName: blank-nodejs-lib
      Description: Dependencies for the blank sample app.
      ContentUri: lib/
      CompatibleRuntimes:
        - nodejs12.x
```

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 \(p. 41\)](#). For a function [defined as a container image \(p. 247\)](#), 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

- [Configuring a function to use layers \(p. 106\)](#)
- [Accessing the contents of a layer \(p. 107\)](#)
- [Finding layer information \(p. 107\)](#)
- [Updating a layer version that your function uses \(p. 108\)](#)
- [Adding layer permissions \(p. 108\)](#)
- [Using AWS SAM to add a layer to a function \(p. 105\)](#)
- [Sample applications \(p. 109\)](#)

Configuring a function to use layers

A Lambda function can use up to five layers at a time. 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 \(p. 44\)](#).

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.

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-2:210987654321: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": "nodejs12.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-2:210987654321:layer:their-layer:2",
      "CodeSize": 169
    }
  ]
}
```

```
        "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. Or, 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.

Accessing the contents of a layer

When you include a layer in a Lambda function, Lambda extracts the layer's contents into the /opt directory in the function execution environment. Lambda extracts layers in the order that you specified them, merging any folders with the same name. If the same file appears in multiple layers, the function uses the version in the last extracted layer.

Each [Lambda runtime \(p. 195\)](#) 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 \(p. 80\)](#).

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'
```

Updating a layer version that your function uses

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.

To add an updated layer version to your function, use the **update-function-configuration** command. Use the `--layers` option with this command to list all of the layer versions that you want to add. If the function already has layers, the new list overwrites the previous list.

To update only one of the layer versions, you must include the ARNs of the existing layer versions with the `--layers` option.

The following procedure assumes that you have packaged the updated layer code into a local file named `layer.zip`.

Add an updated layer version to your function

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-config --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
```

Adding layer permissions

To use a Lambda function with a layer, you need permission to call the [GetLayerVersion \(p. 826\)](#) API operation on the layer version. For functions in your AWS account, you can add this permission from your [user policy \(p. 56\)](#).

To use a layer in another account, the owner of that account must grant your account permission in a [resource-based policy \(p. 51\)](#).

For examples, see [Granting layer access to other accounts \(p. 54\)](#).

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 \(p. 465\)](#). For other samples, see [Samples \(p. 463\)](#).

Configuring Lambda function memory

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).

This page describes how to update the memory allotted to your function in the Lambda console.

Sections

- [Configuring function memory \(console\) \(p. 110\)](#)
- [Accepting function memory recommendations \(console\) \(p. 110\)](#)

Configuring function memory (console)

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

To update the memory of a function

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **General configuration**
4. Under **General configuration**, choose **Edit**.
5. For **Memory (MB)**, set a value from 128 MB to 10,240 MB.
6. 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*.

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](#) on the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **General configuration**
4. Under **General configuration**, choose **Edit**.
5. Under **Memory (MB)**, in the memory alert, choose **Update**.
6. Choose **Save**.

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.

When you connect a function to a VPC, Lambda creates an [elastic network interface](#) for each subnet in your function's VPC configuration. This process can take several minutes.

While Lambda creates a network interface, you can't perform additional operations that target the function, such as [creating versions \(p. 95\)](#) 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 more information about function states, see [Monitoring the state of a function with the Lambda API \(p. 151\)](#).

Multiple functions can share a network interface, if the functions share the same subnet and security group. Connecting additional functions to a subnet that has an existing Lambda-managed network interface is much quicker than having Lambda create additional network interfaces. However, if you have many functions or functions with high network usage, Lambda might still create additional network interfaces.

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.

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

- [Execution role and user permissions \(p. 111\)](#)
- [Configuring VPC access \(console\) \(p. 112\)](#)
- [Configuring VPC access \(API\) \(p. 112\)](#)
- [Using IAM condition keys for VPC settings \(p. 113\)](#)
- [Internet and service access for VPC-connected functions \(p. 116\)](#)
- [VPC tutorials \(p. 116\)](#)
- [Sample VPC configurations \(p. 116\)](#)

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 \(p. 47\)](#) must have the following permissions:

Execution role permissions

- `ec2:CreateNetworkInterface`
- `ec2:DescribeNetworkInterfaces`
- `ec2:DeleteNetworkInterface`

These permissions are included in the AWS managed policy [AWSLambdaVPCAccessExecutionRole](#).

When you configure VPC connectivity, Lambda uses your permissions to verify network resources. To configure a function to connect to a VPC, your AWS Identity and Access Management (IAM) user needs the following permissions:

User permissions

- `ec2:DescribeSecurityGroups`
- `ec2:DescribeSubnets`
- `ec2:DescribeVpcs`

Configuring VPC access (console)

If your [IAM permissions \(p. 113\)](#) 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](#) on 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 \(NAT\) \(p. 116\)](#). 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](#) on 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 \(NAT\) \(p. 116\)](#). 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 \(p. 762\)](#)

- [UpdateFunctionConfiguration \(p. 935\)](#)

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 nodejs12.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 \(p. 762\)](#) and [UpdateFunctionConfiguration \(p. 935\)](#) 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 IAM 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. 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: Configuring a Lambda function to access Amazon RDS in an Amazon VPC \(p. 400\)](#)
- [Tutorial: Configuring a Lambda function to access Amazon ElastiCache in an Amazon VPC \(p. 345\)](#)

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 \(p. 118\)](#)
- [Creating an interface endpoint for Lambda \(p. 119\)](#)
- [Creating an interface endpoint policy for Lambda \(p. 119\)](#)

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.

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`.

```
{  
    "Statement": [  
        {  
            "Principal": "  
                "AWS": "arn:aws:iam::123412341234:user/MyUser"  
            },  
            "Effect": "Allow",  
            "Action": [  
                "lambda:InvokeFunction"  
            ],  
            "Resource": [  
                "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. This enables a function to reach high [concurrency \(p. 31\)](#) levels without exhausting database connections.

Sections

- [Creating a database proxy \(console\) \(p. 121\)](#)
- [Using the function's permissions for authentication \(p. 122\)](#)
- [Sample application \(p. 122\)](#)

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](#) on 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()
}
```

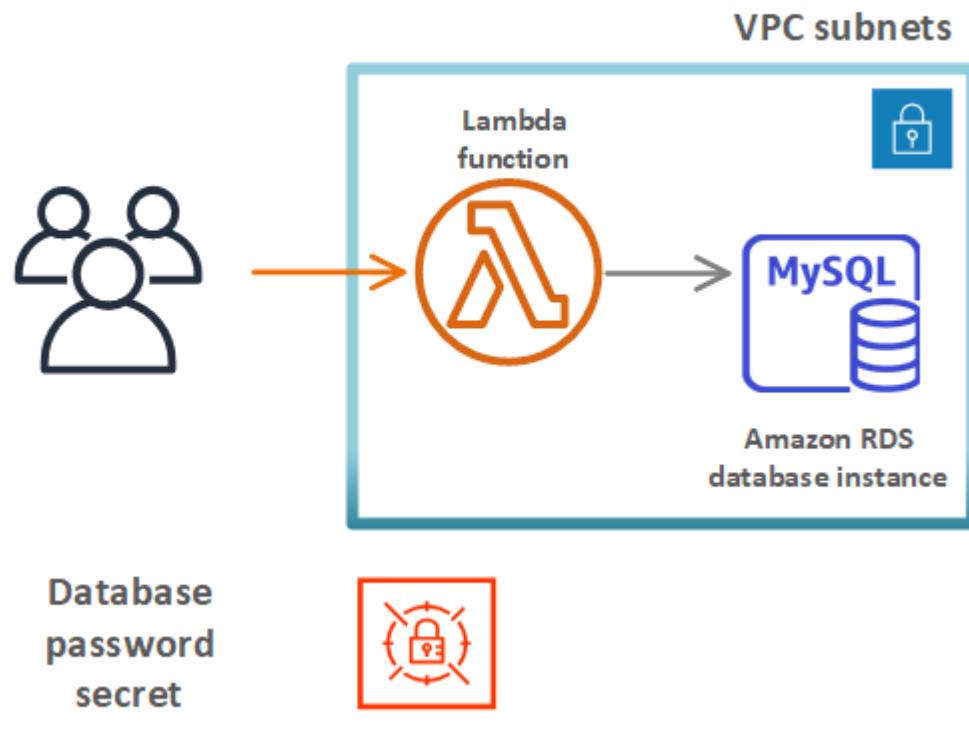
For more information, see [IAM database authentication](#) in the Amazon RDS User Guide.

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:

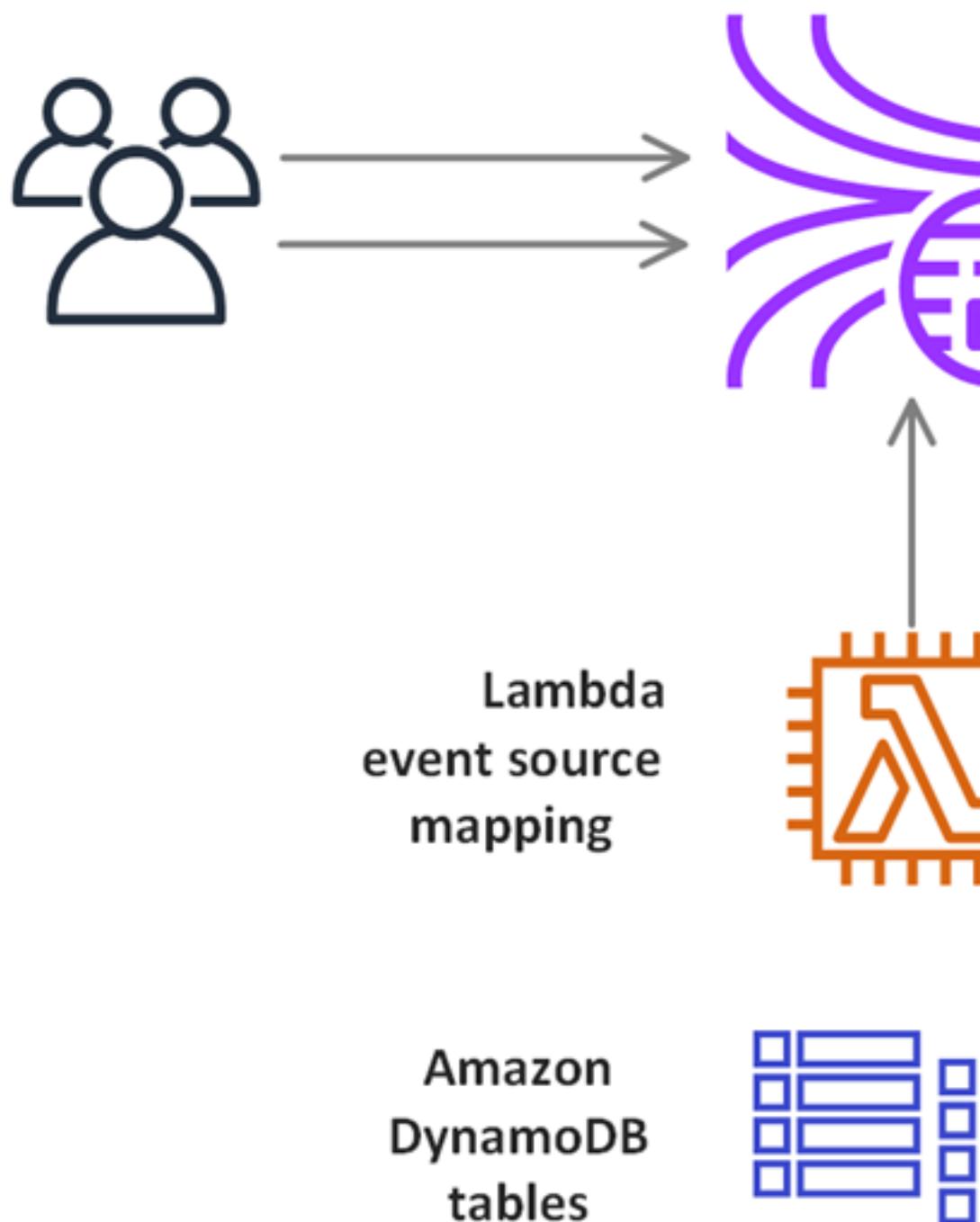
- [RDS MySQL](#) – The AWS CloudFormation template `template-vpcrds.yml` creates a MySQL 5.7 database in a private VPC. In the sample application, a Lambda function proxies queries to the database. The function and database templates both use Secrets Manager to access database credentials.

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

- [Connecting to a file system \(console\) \(p. 126\)](#)
- [Configuring a file system and access point \(p. 126\)](#)
- [Execution role and user permissions \(p. 127\)](#)
- [Configuring file system access with the Lambda API \(p. 128\)](#)
- [AWS CloudFormation and AWS SAM \(p. 128\)](#)
- [Sample applications \(p. 129\)](#)

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 \(p. 111\)](#).

To configure file system access

1. Open the [Functions page](#) on 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 \(p. 351\)](#).

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. User and owner IDs are required, but they don't need to have a specific value. 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](#)

Execution role and user permissions

Lambda uses your function's permissions to mount file systems. To connect to a file system, your function's execution role must have the following permissions in addition to the [permissions required to connect to the file system's VPC \(p. 111\)](#):

Execution role permissions

- `elasticfilesystem:ClientMount`
- `elasticfilesystem:ClientWrite` (not required for read-only connections)

These permissions are included in the `AmazonElasticFileSystemClientReadWriteAccess` managed policy.

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 IAM user needs the following permissions:

User permissions

- `elasticfilesystem:DescribeMountTargets`

Configuring file system access with the Lambda API

Use the following API operations to connect your Lambda function to a file system:

- [CreateFunction \(p. 762\)](#)
- [UpdateFunctionConfiguration \(p. 935\)](#)

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 \
--fs-config FileSystemArn=$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

```
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: "eu-central-1a"
EfsSecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
    VpcId:
      Ref: VPC
    GroupDescription: "mnt target sg"
    SecurityGroupEgress:
      - IpProtocol: -1
        CidrIp: "0.0.0.0/0"
FileSystem:
  Type: AWS::EFS::FileSystem
  Properties:
    PerformanceMode: generalPurpose
MountTarget1:
  Type: AWS::EFS::MountTarget
  Properties:
    FileSystemId:
      Ref: FileSystem
    SubnetId:
      Ref: Subnet1
    SecurityGroups:
      - Ref: EfsSecurityGroup
MyFunctionWithEfs:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: function/
    Description: Use a file system.
    FileSystemConfigs:
      -
        Arn: !Sub
          - "arn:aws:elasticfilesystem:eu-central-1:123456789101:access-point/
fsap-015cxmplb72b405fd"
        LocalMountPath: "/mnt/efs0"
    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 \(p. 303\)](#).

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 \(p. 130\)](#)
- [Configuration prerequisites \(p. 131\)](#)
- [Creating code signing configurations \(p. 131\)](#)
- [Updating a code signing configuration \(p. 131\)](#)
- [Deleting a code signing configuration \(p. 132\)](#)
- [Enabling code signing for a function \(p. 132\)](#)
- [Configuring IAM policies \(p. 132\)](#)
- [Configuring code signing with the Lambda API \(p. 133\)](#)

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](#) on 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 \(p. 133\)](#), attach one or more policy statements to the user policy. For more information about user policies, see [Identity-based IAM policies for Lambda \(p. 56\)](#).

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:DeleteCodeSigningConfig"  
            ]  
        }  
    ]  
}
```

```
        "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 \(p. 762\)](#)
- [GetFunctionCodeSigningConfig](#)
- [PutFunctionCodeSigningConfig](#)
- [DeleteFunctionCodeSigningConfig](#)
- [ListFunctionsByCodeSigningConfig](#)

Tagging Lambda Functions

You can tag Lambda functions to organize them by owner, project or department. Tags are freeform key-value pairs that are supported across AWS services for use in filtering resources and adding detail to billing reports.

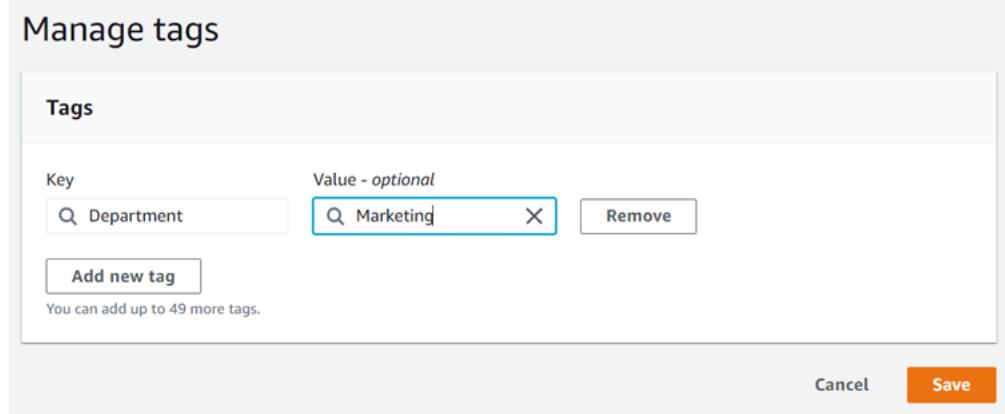
Sections

- [Adding tags to a function \(console\) \(p. 134\)](#)
- [Using tags to filter functions \(console\) \(p. 134\)](#)
- [Using Tags with the AWS CLI \(p. 135\)](#)
- [Tag Key and Value Requirements \(p. 136\)](#)

Adding tags to a function (console)

To add tags to a function

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Tags**.
4. Under **Tags**, choose **Manage tags**.
5. Enter a key and value. To add additional tags, choose **Add new tag**.



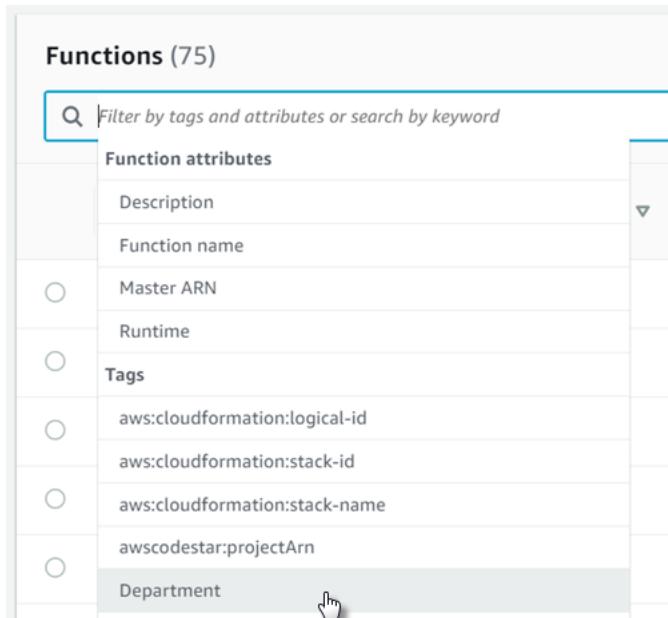
6. Choose **Save**.

Using tags to filter functions (console)

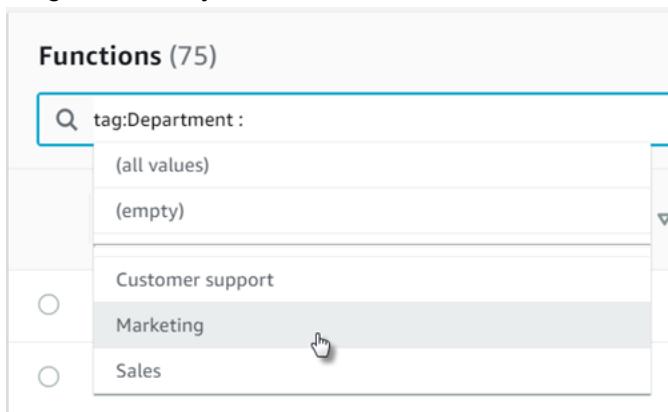
You can filter functions based on the presence or value of a tag with the Lambda console or with the AWS Resource Groups API. Tags apply at the function level, not to versions or aliases. Tags are not part of the version-specific configuration that is snapshotted when you publish a version.

To filter functions with tags

1. Open the [Functions page](#) on the Lambda console.
2. Click within 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 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. Type tag to see just a list of tag keys, or start typing the name of a key to find it in the list.

With AWS Billing and Cost Management, you can use tags to customize billing reports and create cost-allocation reports. For more information, see [Monthly Cost Allocation Report](#) and [Using Cost Allocation Tags](#) in the *AWS Billing and Cost Management User Guide*.

Using Tags with the AWS CLI

When you create a new Lambda function, you can include tags with the `--tags` option.

```
aws lambda create-function --function-name my-function
--handler index.js --runtime nodejs12.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 function arn \
--tag-keys Department
```

If you want to view the tags that are applied to a specific Lambda function, you can use either of the following Lambda API commands:

- [ListTags \(p. 872\)](#) – You supply your Lambda function ARN (Amazon Resource Name) to view a list of the tags associated with this function:

```
aws lambda list-tags --resource function arn
```

- [GetFunction \(p. 807\)](#) – You supply your Lambda function name to view a list of the tags associated with this function:

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

You can also use the AWS Tagging Service's [GetResources](#) API to filter your resources by tags. The GetResources API 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 the AWS Tagging Service, see [Working with Resource Groups](#).

Tag Key and Value Requirements

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 your tagging schema will be used 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: + - = . _ : / @.

Invoking AWS Lambda functions

You can invoke Lambda functions directly [with the Lambda console \(p. 10\)](#), the Lambda API, the AWS SDK, the AWS CLI, and AWS toolkits. You can also configure other AWS services to invoke your function, or you can configure Lambda to read from a stream or queue and invoke your function.

When you invoke a function, you can choose to invoke it synchronously or asynchronously. With [synchronous invocation \(p. 138\)](#), you wait for the function to process the event and return a response. With [asynchronous \(p. 140\)](#) invocation, 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 \(p. 142\)](#).

To use your function to process data automatically, add one or more triggers. A trigger is a Lambda resource or a resource in another service that you configure to invoke your function in response to lifecycle events, external requests, or on a schedule. Your function can have multiple triggers. Each trigger acts as a client invoking your function independently. Each event that Lambda passes to your function only has data from one client or trigger.

To process items from a stream or queue, you can create an [event source mapping \(p. 148\)](#). An event source mapping is a resource in Lambda that reads items from an Amazon SQS queue, an Amazon Kinesis stream, or an Amazon DynamoDB stream, and sends them to your function in batches. Each event that your function processes can contain hundreds or thousands of items.

Other AWS services and resources invoke your function directly. For example, you can configure CloudWatch Events to invoke your function on a timer, or you can configure Amazon S3 to invoke your function when an object is created. Each service varies in the method it uses to invoke your function, the structure of the event, and how you configure it. For more information, see [Using AWS Lambda with other services \(p. 257\)](#).

Depending on who invokes your function and how it's 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 can vary. For details, see [AWS Lambda function scaling \(p. 153\)](#) and [Error handling and automatic retries in AWS Lambda \(p. 158\)](#).

Topics

- [Synchronous invocation \(p. 138\)](#)
- [Asynchronous invocation \(p. 140\)](#)
- [AWS Lambda event source mappings \(p. 148\)](#)
- [Monitoring the state of a function with the Lambda API \(p. 151\)](#)
- [AWS Lambda function scaling \(p. 153\)](#)
- [Error handling and automatic retries in AWS Lambda \(p. 158\)](#)
- [Using Lambda extensions \(p. 160\)](#)
- [Invoking functions defined as container images \(p. 164\)](#)
- [Invoking Lambda functions with the AWS Mobile SDK for Android \(p. 165\)](#)

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 --payload '{ "key": "value" }' response.json
```

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 \(p. 97\)](#)), 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 --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 CLI – Install 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":  
        "U1RBULQgUmVxdWVzdElkOia4N2QwNDRiOC1mMTU0LTEzTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",  
    "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 \(p. 839\)](#).

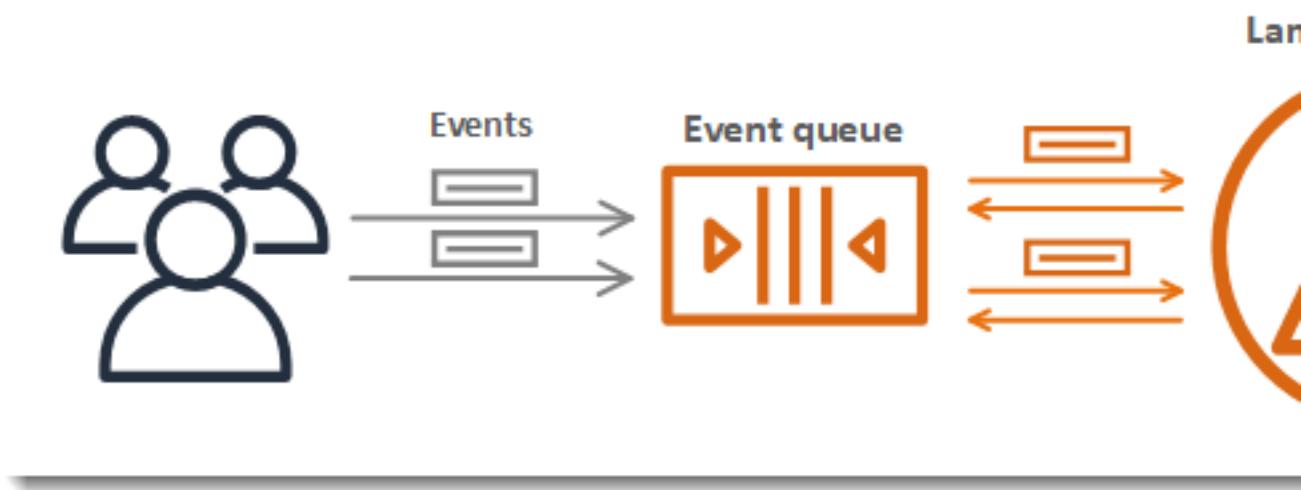
When you invoke a function directly, you can check the response for errors and retry. The AWS CLI and AWS SDK also automatically retry on client timeouts, throttling, and service errors. For more information, see [Error handling and automatic retries in AWS Lambda \(p. 158\)](#).

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 resource to chain together components of your application.

The following diagram shows clients invoking a Lambda function asynchronously. Lambda queues the events before sending them to the function.

Asynchronous Invocation



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
```

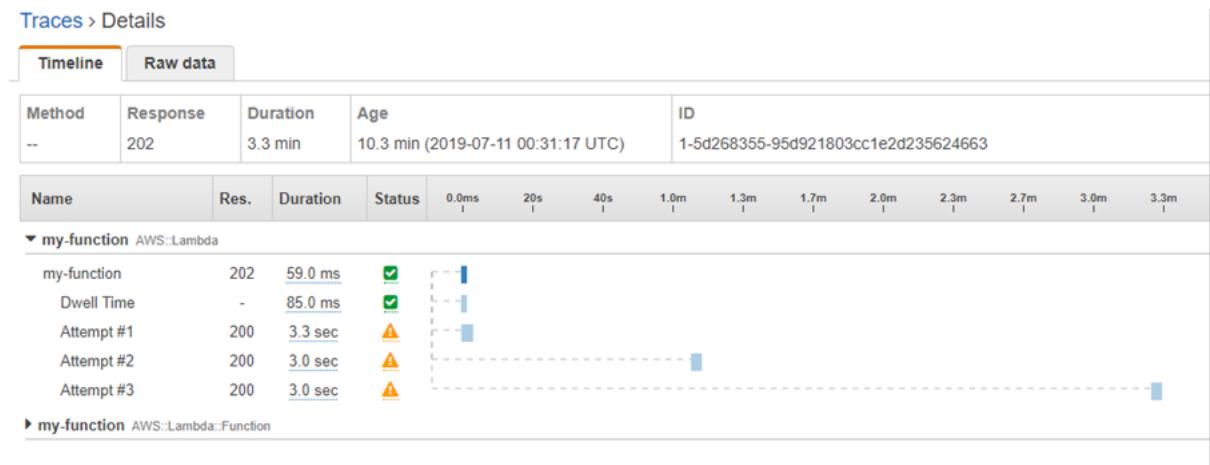
The **cli-binary-format** option is required if you are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#).

You should see the following output:

```
{  
    "StatusCode": 202  
}
```

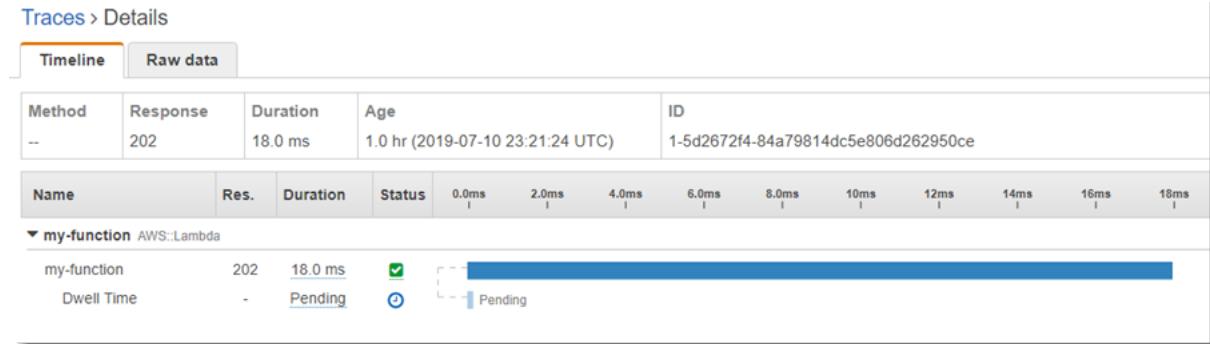
The output file (`response.json`) doesn't contain any information, but is still created when you run this command. If Lambda isn't able to add the event to the queue, the error message appears in the command output.

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. However, it might be longer if the queue is backed up. Lambda also reduces the rate at which it reads events from the queue.

The following example shows an event that was successfully added to the queue, but is still pending one hour later due to throttling.



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 backed up, 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 \(p. 142\)](#) 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 \(p. 142\)](#) for asynchronous invocation.

- **Amazon SQS** – A standard SQS queue.

- **Amazon SNS** – An 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 an SQS queue or SNS topic as a [dead-letter queue \(p. 145\)](#) for discarded events. For dead-letter queues, Lambda only sends the content of the event, without details about the response.

Sections

- [Configuring error handling for asynchronous invocation \(p. 142\)](#)
- [Configuring destinations for asynchronous invocation \(p. 142\)](#)
- [Asynchronous invocation configuration API \(p. 144\)](#)
- [AWS Lambda function dead-letter queues \(p. 145\)](#)

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](#) on 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**.

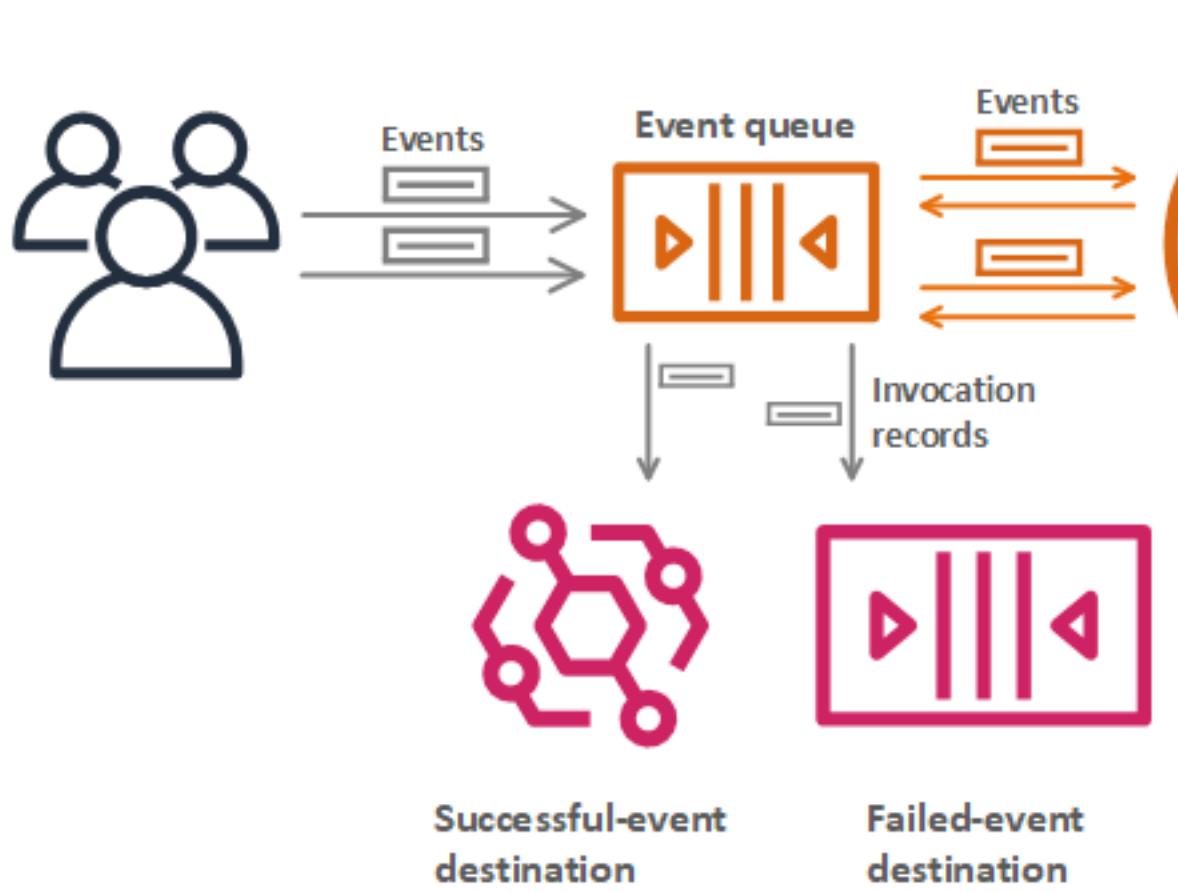
When an invocation event exceeds the maximum age or fails all retry attempts, Lambda discards it. To retain a copy of discarded events, configure a failed-event destination.

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 an 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 \(p. 47\)](#). Each destination service requires a different permission, as follows:

- **Amazon SQS** – [sq:SendMessage](#)
- **Amazon SNS** – [sns:Publish](#)
- **Lambda** – [lambda:InvokeFunction](#)
- **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](#) on 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. 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.

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 an 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"
        }
    }
}
```

AWS Lambda function dead-letter queues

As an alternative to an [on-failure destination](#) (p. 142), 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.

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. You can retrieve events manually, or you can [configure Lambda to read from the queue \(p. 438\)](#) and invoke a function.

Create a queue in the [Amazon SQS console](#).

- **Amazon SNS topic** – A topic relays failed events to one or more destinations. You can configure a topic to send events to an email address, a Lambda function, or an HTTP endpoint.

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 \(p. 47\)](#).

- **Amazon SQS** – [sq: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](#) on 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**.

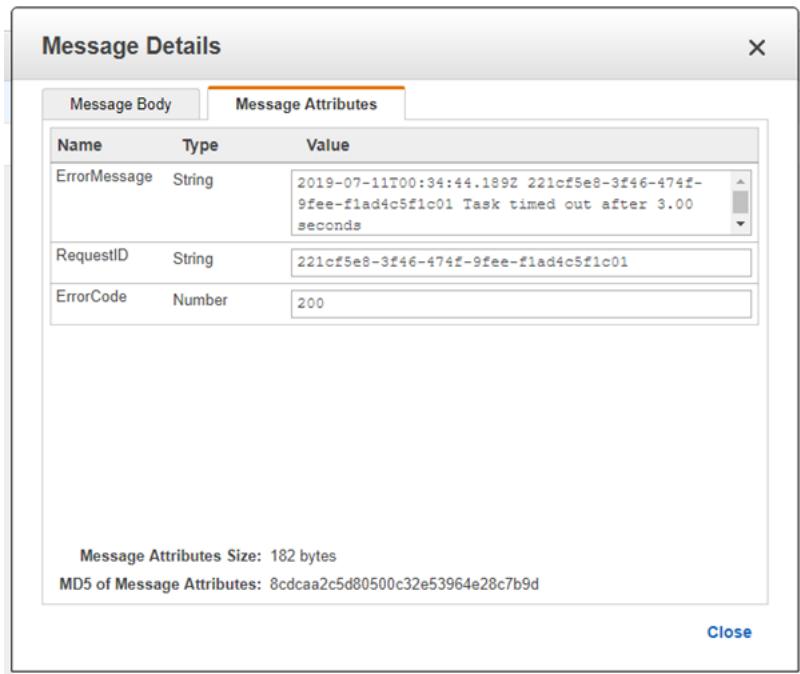
To configure a dead-letter queue with the AWS CLI, use the `update-function-configuration` command.

```
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 \(p. 472\)](#).
- **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 \(p. 691\)](#) 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 AWS Lambda with Amazon SQS \(p. 438\)](#).

AWS Lambda event source mappings

An event source mapping is an AWS 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. Lambda provides event source mappings for the following services.

Services that Lambda reads events from

- [Amazon DynamoDB \(p. 313\)](#)
- [Amazon Kinesis \(p. 365\)](#)
- [Amazon MQ \(p. 389\)](#)
- [Amazon Managed Streaming for Apache Kafka \(p. 395\)](#)
- [self-managed Apache Kafka \(p. 356\)](#)
- [Amazon Simple Queue Service \(p. 438\)](#)

An event source mapping uses permissions in the function's [execution role \(p. 47\)](#) 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 CLI](#) or [AWS SDK](#), you can use the following API operations:

- [CreateEventSourceMapping \(p. 753\)](#)
- [ListEventSourceMappings \(p. 852\)](#)
- [GetEventSourceMapping \(p. 802\)](#)
- [UpdateEventSourceMapping \(p. 918\)](#)
- [DeleteEventSourceMapping \(p. 778\)](#)

The following example uses the AWS CLI to map a function named `my-function` to a DynamoDB stream that is specified by its Amazon Resource Name (ARN), with a batch size of 500.

```
aws lambda create-event-source-mapping --function-name my-function --batch-size 500 --  
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:

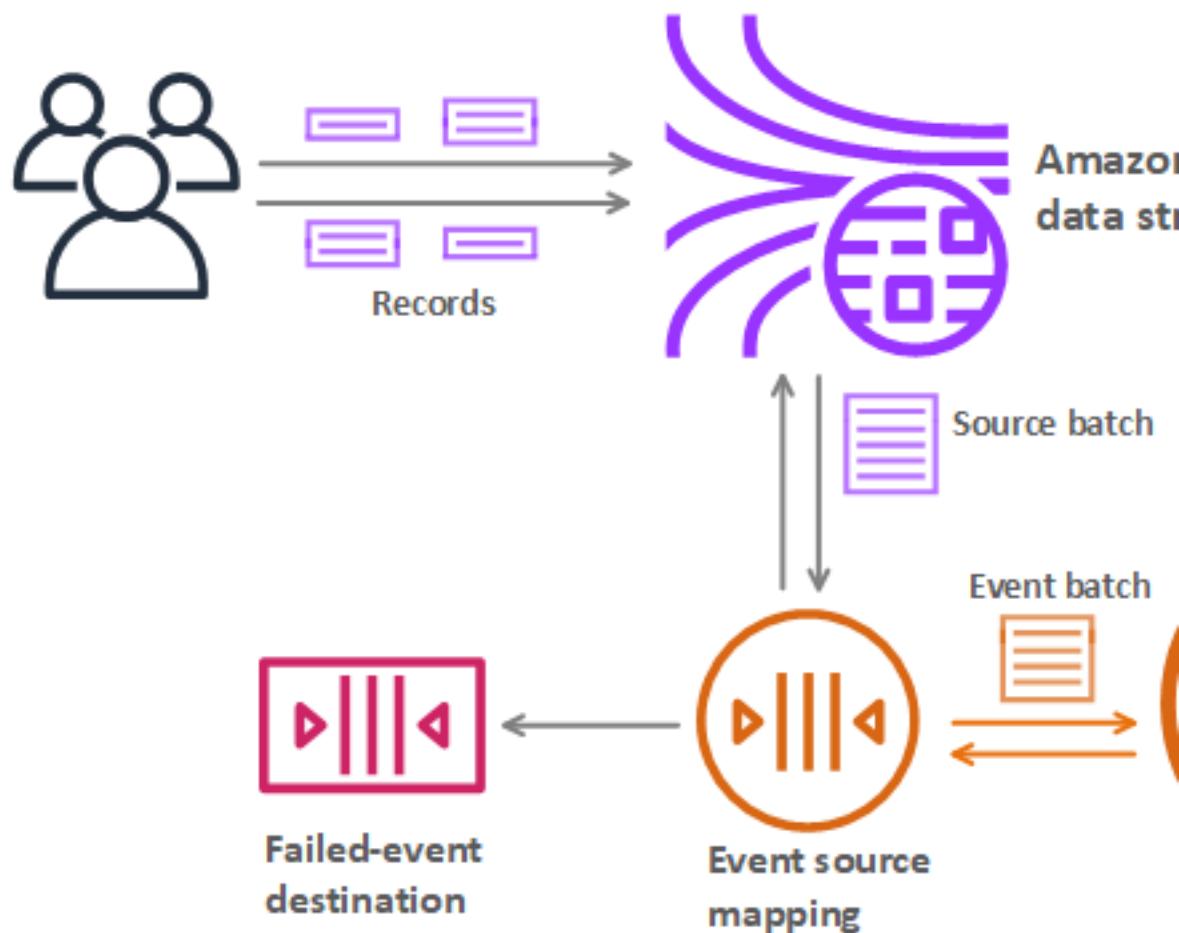
```
{  
    "UUID": "14e0db71-5d35-4eb5-b481-8945cf9d10c2",  
    "BatchSize": 500,  
    "MaximumBatchingWindowInSeconds": 0,  
    "ParallelizationFactor": 1,  
    "EventSourceArn": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/  
stream/2019-06-10T19:26:16.525",  
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",  
    "LastModified": 1560209851.963,  
    "LastProcessingResult": "No records processed",  
    "State": "Creating",  
    "StateTransitionReason": "User action",  
    "DestinationConfig": {},  
    "MaximumRecordAgeInSeconds": 604800,  
    "BisectBatchOnFunctionError": false,
```

```
    "MaximumRetryAttempts": 10000  
}
```

Event source mappings read items from a stream or queue in batches. They include multiple items in the event that your function receives. You can configure the size of the batch that the event source mapping sends to your function, up to a maximum that varies by service. The number of items in the event can be smaller than the batch size if there aren't enough items available, or if the batch is too large to send in one event and has to be split up.

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 from a stream or queue. Batch size and other settings only apply to the event batch.

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 can be processed by other functions or consumers.

By default, if your function returns an error, the entire batch is reprocessed until the function succeeds, or the items in the batch expire. To ensure in-order processing, processing for the affected shard is paused until the error is resolved. You can configure the event source mapping to discard old events, restrict the number of retries, or process multiple batches in parallel. If you process multiple batches in parallel, in-order processing is still guaranteed for each partition key, but multiple partition keys in the same shard are processed simultaneously.

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 \(p. 142\)](#) 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 \(p. 438\)](#), 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, batches are returned to the queue as individual items and might be processed 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 they can't be processed.

For information about services that invoke Lambda functions directly, see [Using AWS Lambda with other services \(p. 257\)](#).

Monitoring the state of a function with the Lambda API

When you create or update a function, Lambda provisions the compute and networking resources that enable it to run. In most cases, this process is very fast, and your function is ready to be invoked or modified right away.

If you configure your function to connect to a virtual private cloud (VPC), the process can take longer. When you first connect a function to a VPC, Lambda provisions network interfaces, which takes about a minute. To communicate the current state of your function, Lambda includes additional fields in the [function configuration \(p. 974\)](#) document that is returned by several Lambda API actions.

When you create a function, the function is initially in the `Pending` state. When the function is ready to be invoked, the state changes from `Pending` to `Active`. While the state is `Pending`, invocations and other API actions that operate on the function return an error. If you build automation around creating and updating functions, wait for the function to become active before performing additional actions that operate on the function.

You can use the Lambda API to get information about a function's state. State information is included in the [FunctionConfiguration \(p. 974\)](#) document returned by several API actions. To view the function's state with the AWS CLI, 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": "nodejs12.x",  
    "Role": "arn:aws:iam::123456789012:role/lambda-role",  
    "TracingConfig": {  
        "Mode": "Active"  
    },  
    "State": "Pending",  
    "StateReason": "The function is being created.",  
    "StateReasonCode": "Creating",  
    ...  
}
```

The `StateReason` and `StateReasonCode` contain additional information about the state when it is not `Active`. The following operations fail while function creation is pending:

- [Invoke \(p. 839\)](#)
- [UpdateFunctionCode \(p. 926\)](#)
- [UpdateFunctionConfiguration \(p. 935\)](#)
- [PublishVersion \(p. 882\)](#)

When you update a function's configuration, the update can trigger an asynchronous operation to provision resources. While this is in progress, you can invoke the function, but other operations on the function fail. Invocations that occur while the update is in progress run against the previous configuration. The function's state is `Active`, but its `LastUpdateStatus` is `InProgress`.

Example Function configuration – Connecting to a VPC

```
{
```

```
"FunctionName": "my-function",
"FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
"Runtime": "nodejs12.x",
"VpcConfig": {
    "SubnetIds": [
        "subnet-071f712345678e7c8",
        "subnet-07fd123456788a036",
        "subnet-0804f77612345cacf"
    ],
    "SecurityGroupIds": [
        "sg-085912345678492fb"
    ],
    "VpcId": "vpc-08e1234569e011e83"
},
"State": "Active",
"LastUpdateStatus": "InProgress",
...
}
```

The following operations fail while an asynchronous update is in progress:

- [UpdateFunctionCode \(p. 926\)](#)
- [UpdateFunctionConfiguration \(p. 935\)](#)
- [PublishVersion \(p. 882\)](#)

Other operations, including invocation, work while updates are in progress.

For example, when you connect your function to a virtual private cloud (VPC), Lambda provisions an elastic network interface for each subnet. This process can leave your function in a pending state for a minute or so. Lambda also reclaims network interfaces that are not in use, placing your function in an `Inactive` state. When the function is inactive, an invocation causes it to enter the `Pending` state while network access is restored. The invocation that triggers restoration, and further invocations while the operation is pending, fail with `ResourceNotReadyException`.

If Lambda encounters an error when restoring a function's network interface, the function goes back to the `Inactive` state. The next invocation can trigger another attempt. For some configuration errors, Lambda waits at least 5 minutes before attempting to create another network interface. These errors have the following `LastUpdateStatusReasonCode` values:

- `InsufficientRolePermission` – Role doesn't exist or is missing permissions.
- `SubnetOutOfIPAddresses` – All IP addresses in a subnet are in use.

For more information on how states work with VPC connectivity, see [Configuring a Lambda function to access resources in a VPC \(p. 111\)](#).

AWS Lambda function scaling

The first time you invoke your function, AWS Lambda creates an instance of the function and runs its handler method to process the event. When the function returns a response, it stays active and waits to process additional events. If you invoke the function again while the first event is being processed, Lambda initializes another instance, and the function processes the two events concurrently. As more events come in, Lambda routes them to available instances and creates new instances as needed. When the number of requests decreases, Lambda stops unused instances to free up scaling capacity for other functions.

Your functions' *concurrency* is the number of instances that serve requests at a given time. 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 of 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

Concurrency
limit

Burst limit

Legend

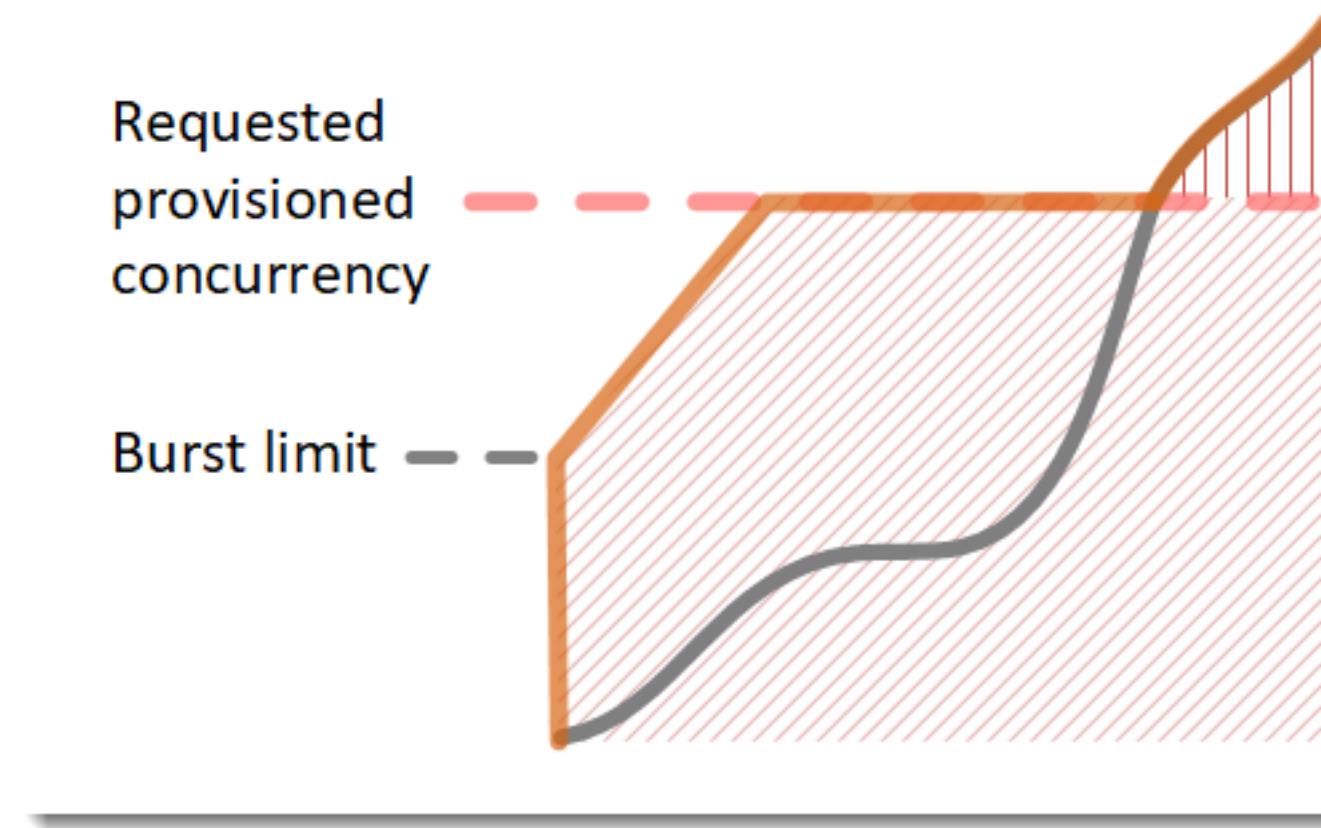
- Function instances
- Open requests
- Throttling possible

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.

The regional concurrency limit starts at 1,000. You can increase the limit by submitting a request in the [Support Center console](#). To allocate capacity on a per-function basis, you can configure functions with [reserved concurrency \(p. 84\)](#). Reserved concurrency creates a pool that can only be used by its function, and also prevents its function from using unreserved concurrency.

When your function scales up, the first request served by each instance is impacted by the time it takes to load and initialize your code. If your [initialization code \(p. 32\)](#) takes a long time, the impact on average and percentile latency can be significant. To enable your function to scale without fluctuations in latency, use [provisioned concurrency \(p. 84\)](#). The following example shows a function with provisioned concurrency processing a spike in traffic.

Function Scaling with Provisioned Concurrency

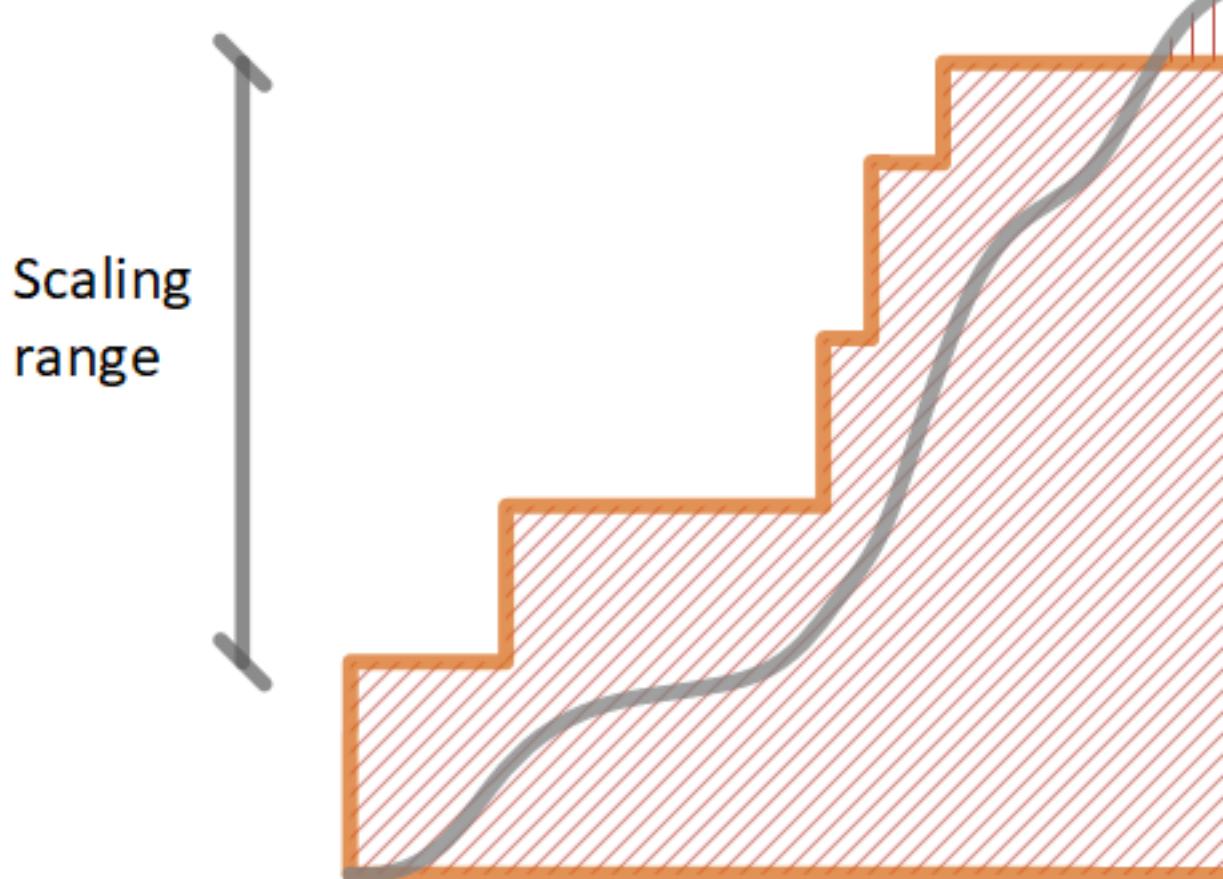


When you allocate provisioned concurrency, your function is ready to serve a burst of incoming requests with very low latency. When all provisioned concurrency is in use, the function scales up normally to handle any additional requests.

Application Auto Scaling takes this a step further by providing autoscaling for provisioned concurrency. With Application Auto Scaling, you can create a target tracking scaling policy that adjusts provisioned concurrency levels automatically, based on the utilization metric that Lambda emits. [Use the Application Auto Scaling API \(p. 90\)](#) to register an alias as a scalable target and create a scaling policy.

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

When you invoke your function asynchronously, by using an event source mapping or another AWS service, scaling behavior varies. For example, event source mappings that read from a stream are limited by the number of shards in the stream. Scaling capacity that is unused by an event source is available for use by other clients and event sources. For more information, see the following topics.

- [Asynchronous invocation \(p. 140\)](#)
- [AWS Lambda event source mappings \(p. 148\)](#)
- [Error handling and automatic retries in AWS Lambda \(p. 158\)](#)
- [Using AWS Lambda with other services \(p. 257\)](#)

You can monitor concurrency levels in your account by using the following metrics:

Concurrency metrics

- `ConcurrentExecutions`
- `UnreservedConcurrentExecutions`
- `ProvisionedConcurrentExecutions`
- `ProvisionedConcurrencyInvocations`
- `ProvisionedConcurrencySpilloverInvocations`
- `ProvisionedConcurrencyUtilization`

For more information, see [Working with AWS Lambda function metrics \(p. 691\)](#).

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 \(p. 195\)](#) 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 \(p. 839\)](#).

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 \(p. 500\)](#)
- [AWS Lambda function errors in Python \(p. 538\)](#)
- [AWS Lambda function errors in Ruby \(p. 561\)](#)
- [AWS Lambda function errors in Java \(p. 596\)](#)
- [AWS Lambda function errors in Go \(p. 628\)](#)
- [AWS Lambda function errors in C# \(p. 657\)](#)
- [AWS Lambda function errors in PowerShell \(p. 678\)](#)

When you invoke a function directly, you determine the strategy for handling errors. You can retry, send the 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 \(p. 140\)](#).
- **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 \(p. 691\)](#) 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 [AWS Lambda event source mappings \(p. 148\)](#) and the service-specific topics under [Using AWS Lambda with other services \(p. 257\)](#).

- **AWS services** – AWS services can invoke your function [synchronously \(p. 138\)](#) 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 \(p. 257\)](#) and the invoking service's documentation.

- **Other accounts and clients** – When you grant access to other accounts, you can use [resource-based policies \(p. 51\)](#) 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 \(p. 261\)](#).

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 applications \(p. 682\)](#).

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 \(p. 472\)](#).

Using 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 \(p. 209\)](#).

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.

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 \(p. 231\)](#).

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.

The following [Lambda runtimes \(p. 195\)](#) support extensions:

- .NET Core 3.1 (C#/PowerShell) (`dotnetcore3.1`)
- Custom runtime ([provided](#))
- Custom runtime on Amazon Linux 2 (`provided.al2`)
- Java 11 (Corretto) (`java11`)
- Java 8 (Corretto) (`java8.al2`)
- Node.js 14.x (`nodejs14.x`)
- Node.js 12.x (`nodejs12.x`)
- Node.js 10.x (`nodejs10.x`)
- Python 3.8 (`python3.8`)
- Python 3.7 (`python3.7`)
- Ruby 2.7 (`ruby2.7`)
- Ruby 2.5 (`ruby2.5`)

Note that the Go 1.x runtime does not support extensions. To support extensions, you can create Go functions on the `provided.al2` runtime. For more information, see [Migrating Lambda functions to Amazon Linux 2](#)

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 \(p. 161\)](#)
- [Impact on performance and resources \(p. 161\)](#)
- [Permissions \(p. 161\)](#)
- [Configuring extensions \(.zip file archive\) \(p. 162\)](#)
- [Using extensions in container images \(p. 162\)](#)
- [Next steps \(p. 162\)](#)

Execution environment

Lambda invokes your function in an [execution environment \(p. 200\)](#), 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 \(p. 87\)](#).

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.

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 \(p. 691\)](#). 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 extensions (.zip file archive)

You can add an extension to your function as a [Lambda layer \(p. 101\)](#). 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 \(p. 106\)](#).

Add an extension to your function (console)

1. Open the [Functions page](#) on 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 \(p. 247\)](#). The ENTRYPOINT container image setting specifies the main process for the function. Configure the ENTRYPOINT 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 \(p. 209\)](#) 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 ./index.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.

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 \(p. 164\)](#)
- [Invoking the function \(p. 164\)](#)
- [Image security \(p. 164\)](#)

Function lifecycle

Lambda optimizes a new or updated container image in preparation for the function to receive 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, 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 [Monitoring the state of a function with the Lambda API \(p. 151\)](#).

Invoking the function

When you invoke the function, Lambda deploys the container image to an execution environment. Lambda initializes any [extensions \(p. 162\)](#) 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 `ENTRYPOINT` and `CMD` [container image settings \(p. 249\)](#)).

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 customer managed AWS Key Management Service (AWS KMS) customer master keys (CMKs). To track and audit Lambda's usage of CMKs, you can view the [AWS CloudTrail logs \(p. 285\)](#).

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 \(p. 165\)](#).

When you invoke a function from a mobile application, you choose the event structure, [invocation type \(p. 137\)](#), and permission model. You can use [aliases \(p. 97\)](#) 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.

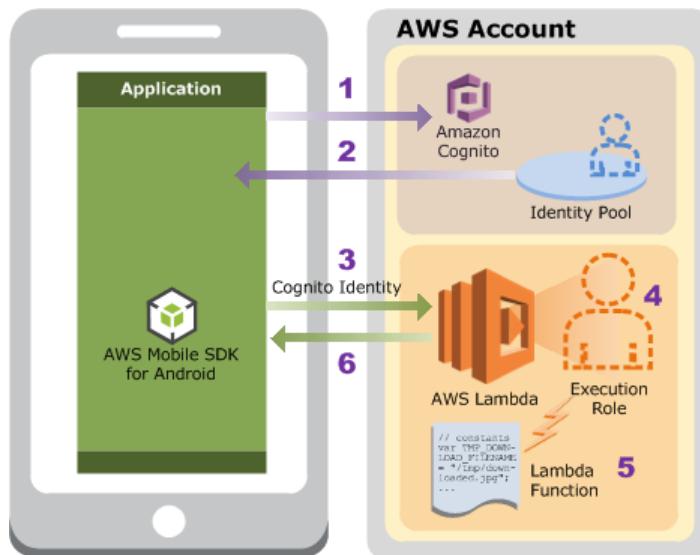
To create a fully-featured web API for your mobile and web applications, use Amazon API Gateway. With API Gateway, you can add custom authorizers, throttle requests, and cache results for all of your functions. For more information, see [Using AWS Lambda with Amazon API Gateway \(p. 261\)](#).

Topics

- [Tutorial: Using AWS Lambda with the Mobile SDK for Android \(p. 165\)](#)
- [Sample function code \(p. 171\)](#)

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 [Getting started with Lambda \(p. 9\)](#) to create your first Lambda function.

To complete the following steps, you need a command line terminal or shell to run commands. 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. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the execution role

Create the [execution role \(p. 47\)](#) 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 \(p. 171\)](#).

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 fileb://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 outputfile.txt
```

The **cli-binary-format** option is required if you are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#)

.

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 \(p. 171\)](#)
- [Java \(p. 171\)](#)

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 \(p. 488\)](#).

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 \(p. 577\)](#).

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 (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 \(p. 5\)](#) extends AWS CloudFormation with a simplified syntax focused on Lambda application development.

The [AWS CLI \(p. 5\)](#) and [SAM CLI \(p. 6\)](#) 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 like uploading deployment packages and updating templates. The AWS SAM CLI provides additional functionality, including validating templates and testing locally.

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

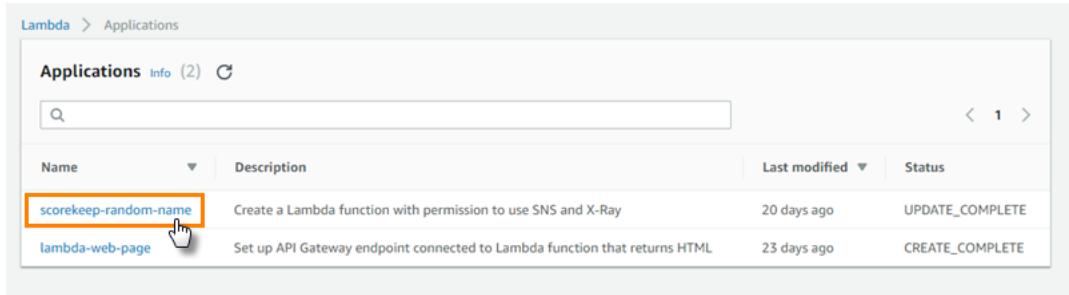
- [Managing applications in the AWS Lambda console \(p. 175\)](#)
- [Creating an application with continuous delivery in the Lambda console \(p. 178\)](#)
- [Rolling deployments for Lambda functions \(p. 187\)](#)
- [Common Lambda application types and use cases \(p. 189\)](#)
- [Best practices for working with AWS Lambda functions \(p. 192\)](#)

Managing applications in the AWS Lambda console

The AWS Lambda console helps you monitor and manage your [Lambda applications \(p. 174\)](#). 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`.

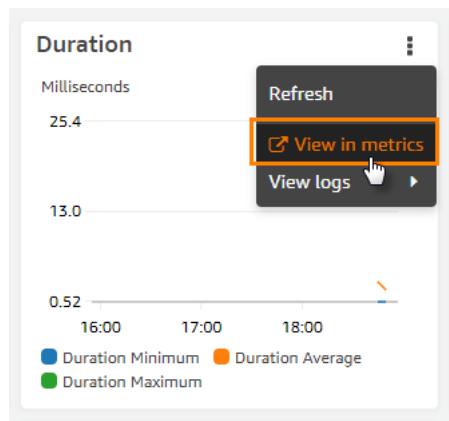
Example function dashboard template

```
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 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 \(p. 47\)](#) 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 \(p. 179\)](#)
- [Create an application \(p. 179\)](#)
- [Invoke the function \(p. 180\)](#)
- [Add an AWS resource \(p. 181\)](#)
- [Update the permissions boundary \(p. 183\)](#)
- [Update the function code \(p. 183\)](#)
- [Next steps \(p. 184\)](#)
- [Troubleshooting \(p. 185\)](#)
- [Clean up \(p. 186\)](#)

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 [Getting started with Lambda \(p. 9\)](#) to create your first Lambda function.

To complete the following steps, you need a command line terminal or shell to run commands. 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. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

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 10.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**.

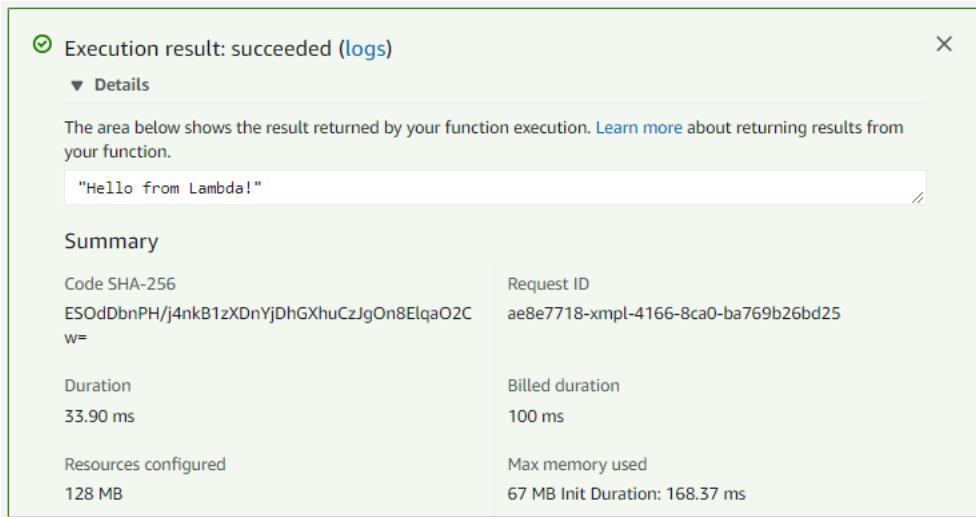
Invoke the function

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 \(p. 179\)](#).
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: ./
    Handler: src/handlers/hello-from-lambda.helloFromLambdaHandler
    Runtime: nodejs10.x
    MemorySize: 128
    Timeout: 60
  Description: A Lambda function that returns a static string.
Environment:
  Variables:
    DDB_TABLE: !RefddbTable
Policies:
  - DynamoDBCrudPolicy:
      TableName: !RefddbTable
  - 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.

The screenshot shows the AWS Lambda Application console for the application 'my-app'. The 'Deployments' tab is selected. The 'Application pipeline' section shows one pipeline named 'my-app-Pipeline' with the most recent action being a 'Build: PackageExport' that occurred 8 seconds ago, currently in progress. Below this, the 'SAM template' section shows a CloudFormation stack. The 'Deployment history' section shows two entries: one deployment completed 6 minutes ago and another completed 2 hours ago, both for Lambda applications.

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 \(p. 67\)](#).

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: nodejs10.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**.

The screenshot shows the AWS DynamoDB Items page for the table 'my-app-ddbTable-6WYEXMPL8K55'. The table has one item, which is displayed in the main table area. The item has an 'id' attribute with the value '2020/05/08[\$LATEST]e2fbe7b2ce58490486c4c57da0f2aa7b' and an 'invocations' attribute with the value '3'.

id	invocations
2020/05/08[\$LATEST]e2fbe7b2ce58490486c4c57da0f2aa7b	3

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 [AWS Lambda function scaling \(p. 153\)](#).

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](#)

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 \(p. 56\)](#) 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 \(p. 47\)](#) 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 \(p. 111\)](#) (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-YV1VXMPK7QK`).
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](#).

Common Lambda application types and use cases

When building applications on AWS Lambda the core components are Lambda functions and triggers. A trigger is the AWS service or application that invokes a function, and a Lambda function is the code and runtime that process events. To illustrate, consider the following scenarios:

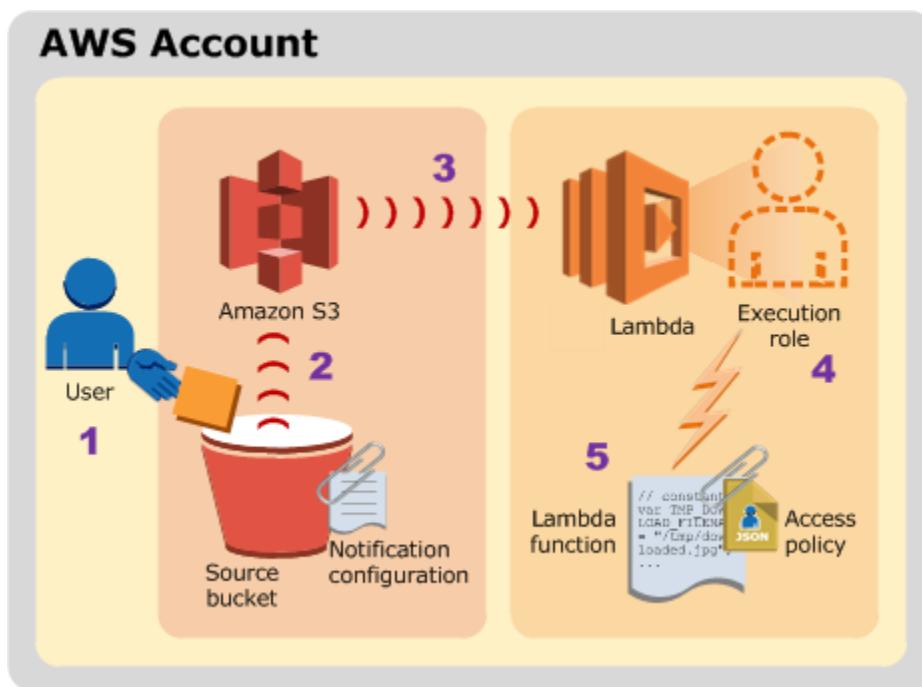
- **File processing** – Suppose you have a photo sharing application. People use your application to upload photos, and the application stores these user photos in an Amazon S3 bucket. Then, your application creates a thumbnail version of each user's photos and displays them on the user's profile page. 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, in this scenario 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 \(p. 257\)](#). 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, as shown in the following illustration.



The diagram illustrates the following 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 \(p. 47\)](#).
4. AWS Lambda runs the Lambda function, specifying the event as a parameter.

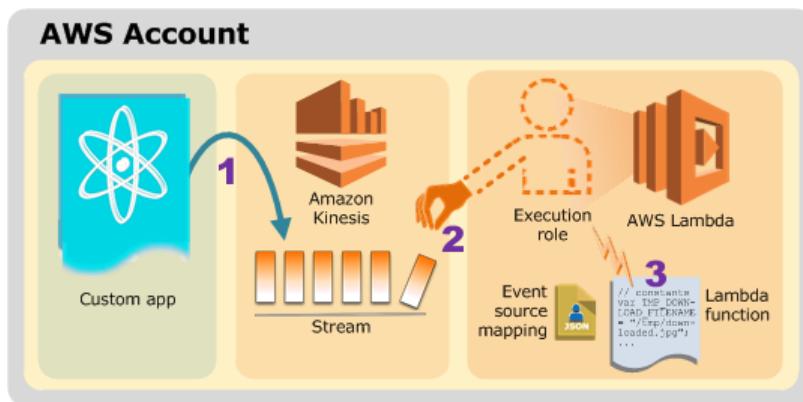
You configure Amazon S3 to invoke your function as a bucket notification action. To grant Amazon S3 permission to invoke the function, update the function's [resource-based policy \(p. 51\)](#).

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 \(p. 753\)](#)
- [UpdateEventSourceMapping \(p. 918\)](#)

The following diagram shows how a custom application writes records to a Kinesis stream.



The diagram illustrates the following sequence:

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.

Best practices for working with AWS Lambda functions

The following are recommended best practices for using AWS Lambda:

Topics

- [Function code \(p. 192\)](#)
- [Function configuration \(p. 193\)](#)
- [Metrics and alarms \(p. 194\)](#)
- [Working with streams \(p. 194\)](#)

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. 77) 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 \(p. 195\)](#)). 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 \(p. 577\)](#) for instructions.
- **Minimize the complexity of your dependencies.** Prefer simpler frameworks that load quickly on [execution environment \(p. 200\)](#) 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.

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 [AWS Lambda permissions \(p. 46\)](#).
- **Be familiar with Lambda quotas (p. 44).** 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 \(p. 762\)](#) and [UpdateFunctionConfiguration \(p. 935\)](#).
 - 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 AWS Lambda function metrics \(p. 691\)](#) 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.)

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. [BatchSize \(p. 754\)](#) 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 in the stream. If the batch that Lambda reads from the stream only has 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 five minutes by configuring a *batch window*. Before invoking the function, Lambda continues to read records from the stream until it has gathered a full batch, or until the batch window expires.

- **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).

Lambda runtimes

Lambda supports multiple languages through the use of [runtimes \(p. 30\)](#). For a function defined as a [container image \(p. 72\)](#), you choose a runtime and the Linux distribution when you [create the container image \(p. 248\)](#). 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 \(p. 70\)](#). The runtime is paired with one of the Amazon Linux distributions. The underlying execution environment provides additional libraries and [environment variables \(p. 77\)](#) that you can access from your function code.

Amazon Linux

- Image – [amzn-ami-hvm-2018.03.0.20181129-x86_64-gp2](#)
- Linux kernel – 4.14.171-105.231.amzn1.x86_64

Amazon Linux 2

- Image – Custom
- Linux kernel – 4.14.165-102.205.amzn2.x86_64

When your function is invoked, Lambda attempts to re-use the execution environment from a previous invocation if one is available. This saves time preparing the execution environment, and it allows you to save resources such as database connections and temporary files in the [execution environment \(p. 200\)](#) to avoid creating them every time your function runs.

A runtime can support a single version of a language, multiple versions of a language, or multiple languages. Runtimes specific to a language or framework version are [deprecated \(p. 198\)](#) when the version reaches end of life.

Node.js runtimes

Name	Identifier	SDK for JavaScript	Operating system
Node.js 14	nodejs14.x	2.880.0	Amazon Linux 2
Node.js 12	nodejs12.x	2.880.0	Amazon Linux 2
Node.js 10	nodejs10.x	2.880.0	Amazon Linux 2

Note

For end of support information about Node.js 10, see [the section called “Runtime support policy” \(p. 198\)](#).

Python runtimes

Name	Identifier	AWS SDK for Python	Operating system	
Python 3.8	python3.8	boto3-1.17.42 botocore-1.20.42	Amazon Linux 2	
Python 3.7	python3.7	boto3-1.17.42 botocore-1.20.42	Amazon Linux	

Name	Identifier	AWS SDK for Python	Operating system	
Python 3.6	python3.6	boto3-1.17.42 botocore-1.20.42	Amazon Linux	
Python 2.7	python2.7	boto3-1.17.42 botocore-1.20.42	Amazon Linux	

Important

Python 2.7 reached end of life on January 1, 2020. End of support (phase 1) for the Python 2.7 runtime starts on July 15, 2021. For more information, see [Announcing end of support for Python 2.7 in AWS Lambda](#) on the AWS Compute Blog.

Ruby runtimes

Name	Identifier	SDK for Ruby	Operating system	
Ruby 2.7	ruby2.7	3.0.1	Amazon Linux 2	
Ruby 2.5	ruby2.5	3.0.1	Amazon Linux	

Note

For end of support information about Ruby 2.5, see [the section called “Runtime support policy” \(p. 198\)](#).

Java runtimes

Name	Identifier	JDK	Operating system	
Java 11	java11	amazon-corretto-11	Amazon Linux 2	
Java 8	java8.al2	amazon-corretto-8	Amazon Linux 2	
Java 8	java8	java-1.8.0-openjdk	Amazon Linux	

Go runtimes

Name	Identifier	Operating system	
Go 1.x	go1.x	Amazon Linux	

.NET runtimes

Name	Identifier	Operating system	
.NET Core 3.1	dotnetcore3.1	Amazon Linux 2	
.NET Core 2.1	dotnetcore2.1	Amazon Linux	

Note

For end of support information about .NET Core 2.1, see [the section called “Runtime support policy” \(p. 198\)](#).

To use other languages in Lambda, you can implement a [custom runtime \(p. 236\)](#). The Lambda execution environment provides a [runtime interface \(p. 205\)](#) for getting invocation events and sending responses. You can deploy a custom runtime alongside your function code, or in a [layer \(p. 101\)](#).

Custom runtime

Name	Identifier	Operating system
Custom Runtime	<code>provided.al2</code>	Amazon Linux 2
Custom Runtime	<code>provided</code>	Amazon Linux

Topics

- [Runtime support policy \(p. 198\)](#)
- [AWS Lambda execution environment \(p. 200\)](#)
- [Runtime support for Lambda container images \(p. 203\)](#)
- [AWS Lambda runtime API \(p. 205\)](#)
- [Lambda Extensions API \(p. 209\)](#)
- [Lambda Logs API \(p. 223\)](#)
- [Modifying the runtime environment \(p. 231\)](#)
- [Custom AWS Lambda runtimes \(p. 236\)](#)
- [Tutorial – Publishing a custom runtime \(p. 239\)](#)
- [Using AVX2 vectorization in Lambda \(p. 245\)](#)

Runtime support policy

Lambda runtimes (p. 195) 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. In 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 version, and rolling back to the previous runtime version. Note that functions that use a deprecated runtime are no longer eligible for technical support.

In phase 2, which starts at least 30 days after the start of phase 1, 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 version. After you migrate the function to a supported runtime version, you cannot rollback the function to the previous runtime.

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

Important

Python 2.7 reached end of life on January 1, 2020. End of support (phase 1) for the Python 2.7 runtime starts on July 15, 2021. For more information, see [Announcing end of support for Python 2.7 in AWS Lambda](#) on the AWS Compute Blog.

The following runtimes have reached or are scheduled for end of support:

Runtime end of support dates

Name	Identifier	Operating system	End of support phase 1 start	End of support phase 2 start
.NET Core 2.1	dotnetcore2.1	Amazon Linux	Aug 23, 2021	Sept 23, 2021
Python 2.7	python2.7	Amazon Linux	July 15, 2021	Sept 30, 2021
Ruby 2.5	ruby2.5	Amazon Linux	July 30, 2021	Aug 30, 2021
Node.js 10.x	nodejs10.x	Amazon Linux 2	July 30, 2021	Aug 30, 2021
Node.js 8.10	nodejs8.10	Amazon Linux		March 6, 2020
Node.js 6.10	nodejs6.10	Amazon Linux		August 12, 2019
Node.js 4.3 edge	nodejs4.3-edge	Amazon Linux		April 30, 2019
Node.js 4.3	nodejs4.3	Amazon Linux		March 6, 2020
Node.js 0.10	nodejs	Amazon Linux		October 31, 2016
.NET Core 2.0	dotnetcore2.0	Amazon Linux		May 30, 2019
.NET Core 1.0	dotnetcore1.0	Amazon Linux		July 30, 2019

In almost all cases, the end-of-life date of a language version or operating system is known well in advance. Lambda notifies you by email if you have functions using a runtime that is scheduled for end of

support in the next 60 days. In rare cases, advance notice of support ending might not be possible. For example, security issues that require a backwards-incompatible update, or a runtime component that doesn't provide a long-term support (LTS) schedule.

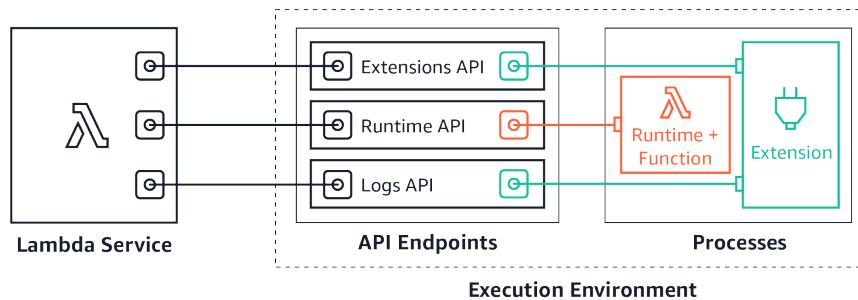
Language and framework support policies

- **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

AWS 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 \(p. 160\)](#) associated with your function.

The function's runtime communicates with Lambda using the [Runtime API \(p. 205\)](#). Extensions communicate with Lambda using the [Extensions API \(p. 209\)](#). Extensions can also receive log messages from the function by subscribing to logs using the [Logs API \(p. 223\)](#).



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.

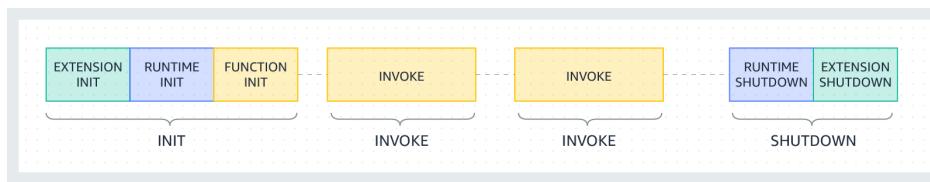
Lambda execution environment lifecycle

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 \(p. 87\)](#).

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 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 \(p. 201\)](#)
- [Invoke phase \(p. 201\)](#)
- [Shutdown phase \(p. 202\)](#)

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`)

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.

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.

If the Lambda function crashes or times out during the `Invoke` phase, Lambda resets the execution environment. The reset behaves like a `Shutdown` event. First, Lambda shuts down the runtime. Then Lambda sends a `Shutdown` event to each registered external extension. The event includes the reason for the shutdown. If another `Invoke` event results in this execution environment being reused, Lambda initializes the runtime and extensions as part of the next invocation.



Shutdown phase

When Lambda is about to shut down the runtime, it sends a `Shutdown` event to the runtime and to each 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 512 MB 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 \(p. 44\)](#).
- 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.

Runtime support for Lambda container images

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.

Topics

- [AWS base images for Lambda \(p. 203\)](#)
- [Runtime interface clients \(p. 203\)](#)
- [Runtime interface emulator \(p. 204\)](#)

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 an AWS base images when any new managed runtime becomes available.

Lambda provides base images for the following runtimes:

- [Node.js \(p. 491\)](#)
- [Python \(p. 528\)](#)
- [Java \(p. 584\)](#)
- [.NET \(p. 650\)](#)
- [Go \(p. 619\)](#)
- [Ruby \(p. 553\)](#)

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

DockerHub: [amazon/aws-lambda-provided](#)

ECR Public: [public.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 \(p. 205\)](#), along with the [Extensions API \(p. 209\)](#), 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 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 \(p. 491\)](#)
- [Python \(p. 529\)](#)
- [Java \(p. 584\)](#)
- [.NET \(p. 650\)](#)
- [Go \(p. 619\)](#)
- [Ruby \(p. 553\)](#)

Runtime interface emulator

Lambda provides a runtime interface emulator (RIE) for you to test your function locally. The AWS base images for Lambda include the RIE. For other base images, you can download the [Runtime interface emulator](#) from the AWS GitHub repository.

AWS Lambda runtime API

AWS Lambda provides an HTTP API for [custom runtimes \(p. 236\)](#) to receive invocation events from Lambda and send response data back within the Lambda [execution environment \(p. 195\)](#).

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 \(p. 205\)](#)
- [Invocation response \(p. 206\)](#)
- [Initialization error \(p. 206\)](#)
- [Invocation error \(p. 207\)](#)

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-e9f4-11e8-9739-2dfe598c3fcd`.
- `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 boots 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 extension 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`
- `Extension.APIKeyNotFound`
- `Extension.ConfigInvalid`
- `Extension.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. Extension 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 extension 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`
- `Extension.APIKeyNotFound`
- `Extension.ConfigInvalid`

- Extension.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. Extension 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"
```

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, and open-source projects. For more information on using extensions, see [Introducing AWS Lambda Extensions](#) on the AWS Compute Blog.

As an extension author, you can use the Lambda Extensions API to integrate deeply into the Lambda [execution environment](#) (p. 200). 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. 223) 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 all of the supported runtimes. If you use a non-compiled language, ensure that you include a compatible runtime in the extension.

The following [Lambda runtimes](#) (p. 195) support extensions:

- .NET Core 3.1 (C#/PowerShell) (`dotnetcore3.1`)
- Custom runtime ([provided](#))
- Custom runtime on Amazon Linux 2 (`provided.al2`)
- Java 11 (Corretto) (`java11`)
- Java 8 (Corretto) (`java8.al2`)
- Node.js 14.x (`nodejs14.x`)
- Node.js 12.x (`nodejs12.x`)
- Node.js 10.x (`nodejs10.x`)
- Python 3.8 (`python3.8`)
- Python 3.7 (`python3.7`)
- Ruby 2.7 (`ruby2.7`)
- Ruby 2.5 (`ruby2.5`)

Note that the Go 1.x runtime does not support extensions. To support extensions, you can create Go functions on the `provided.al2` runtime. For more information, see [Migrating Lambda functions to Amazon Linux 2](#)

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. 231). 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. 41), you deploy your extension as a [layer](#) (p. 101). For a function defined as a container image, you add [the extensions](#) (p. 162) 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 \(p. 210\)](#)
- [Extensions API reference \(p. 218\)](#)

Lambda execution environment lifecycle

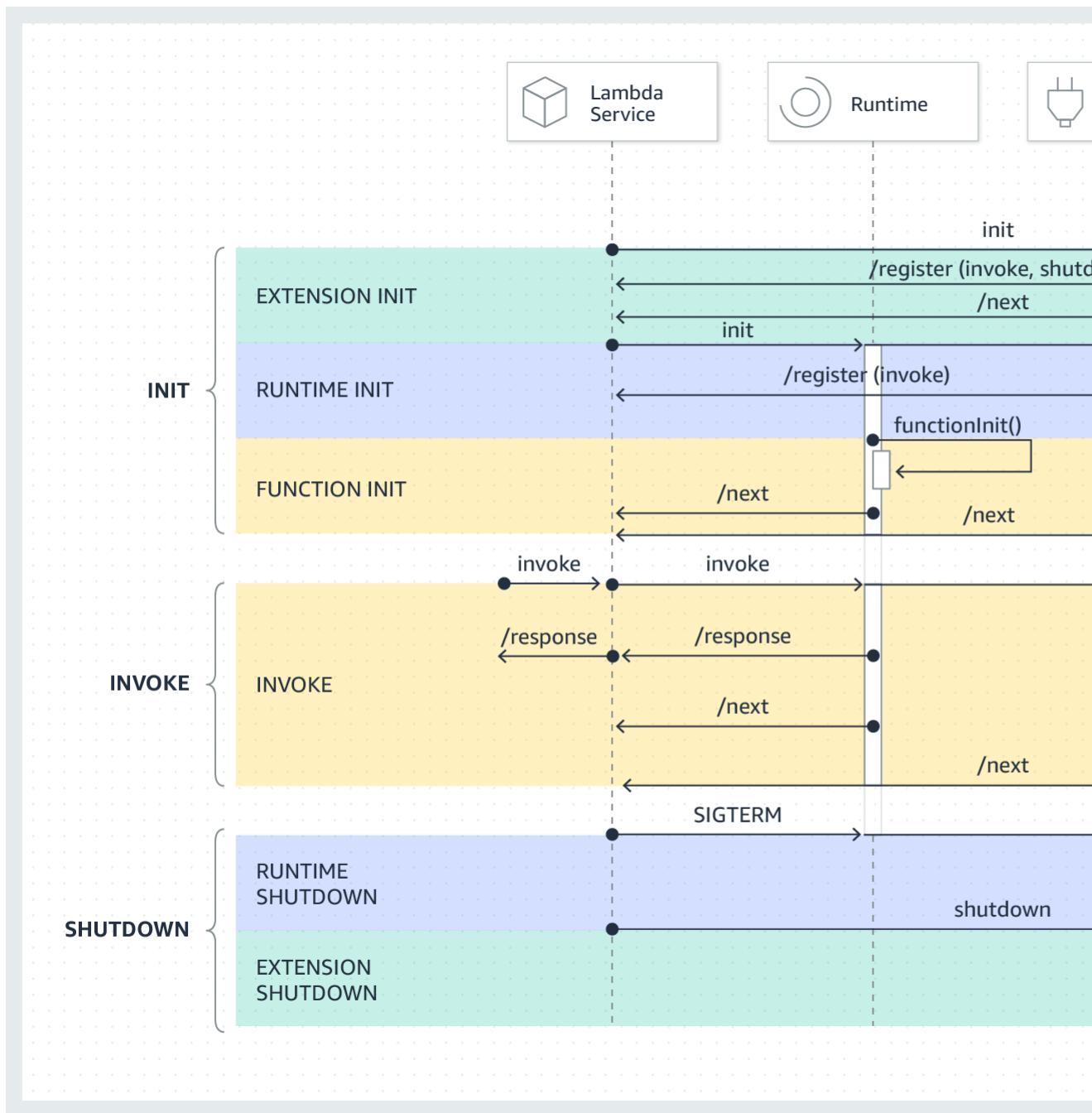
The lifecycle of the execution environment includes the following phases:

- **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 \(p. 87\)](#).

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 \(p. 212\)](#)
- [Invoke phase \(p. 201\)](#)
- [Shutdown phase \(p. 202\)](#)
- [Permissions and configuration \(p. 216\)](#)
- [Failure handling \(p. 217\)](#)
- [Troubleshooting extensions \(p. 218\)](#)

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.

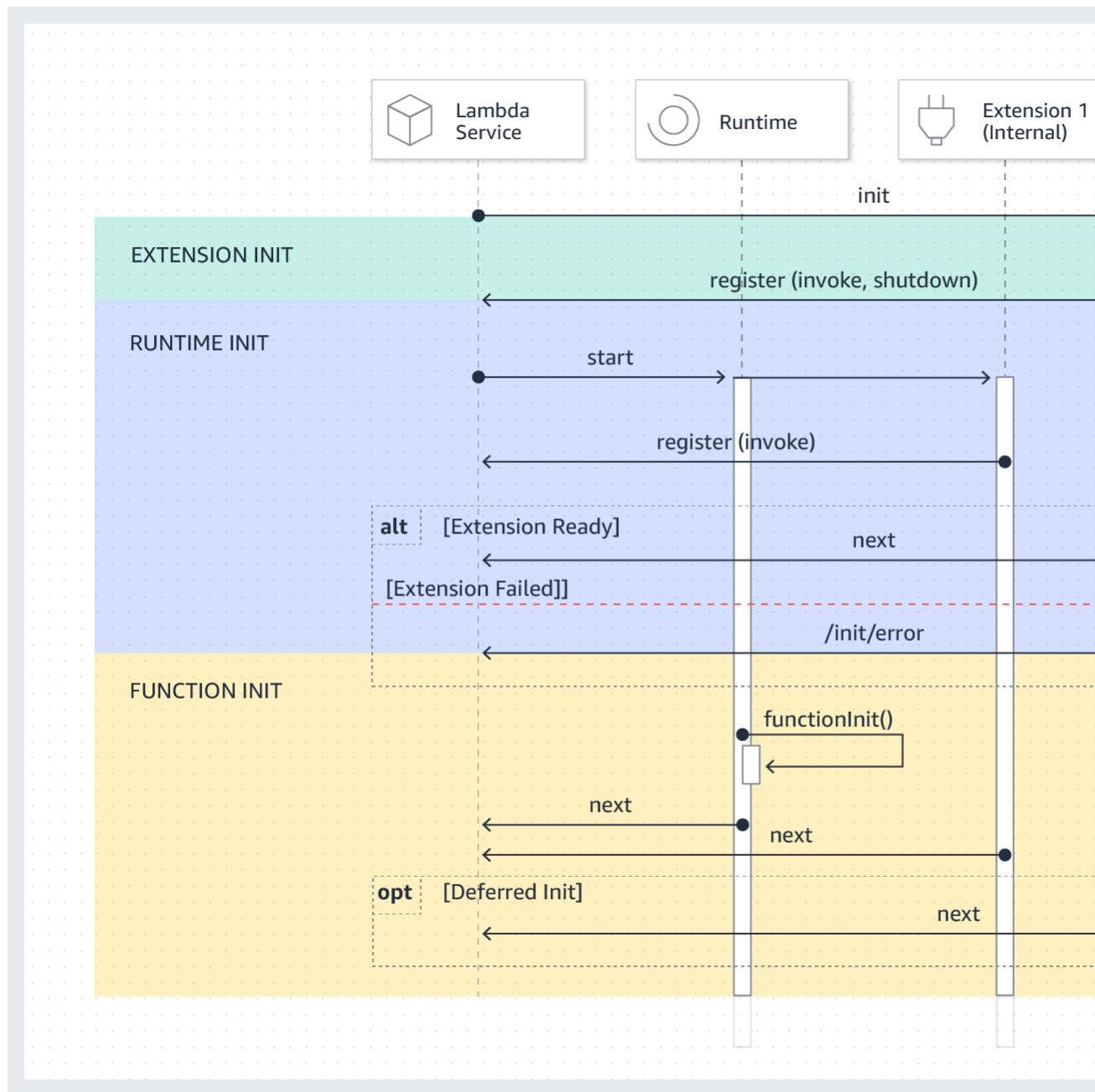
You can register up to 10 extensions for a function. This limit is enforced through the `Register` API call.

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.



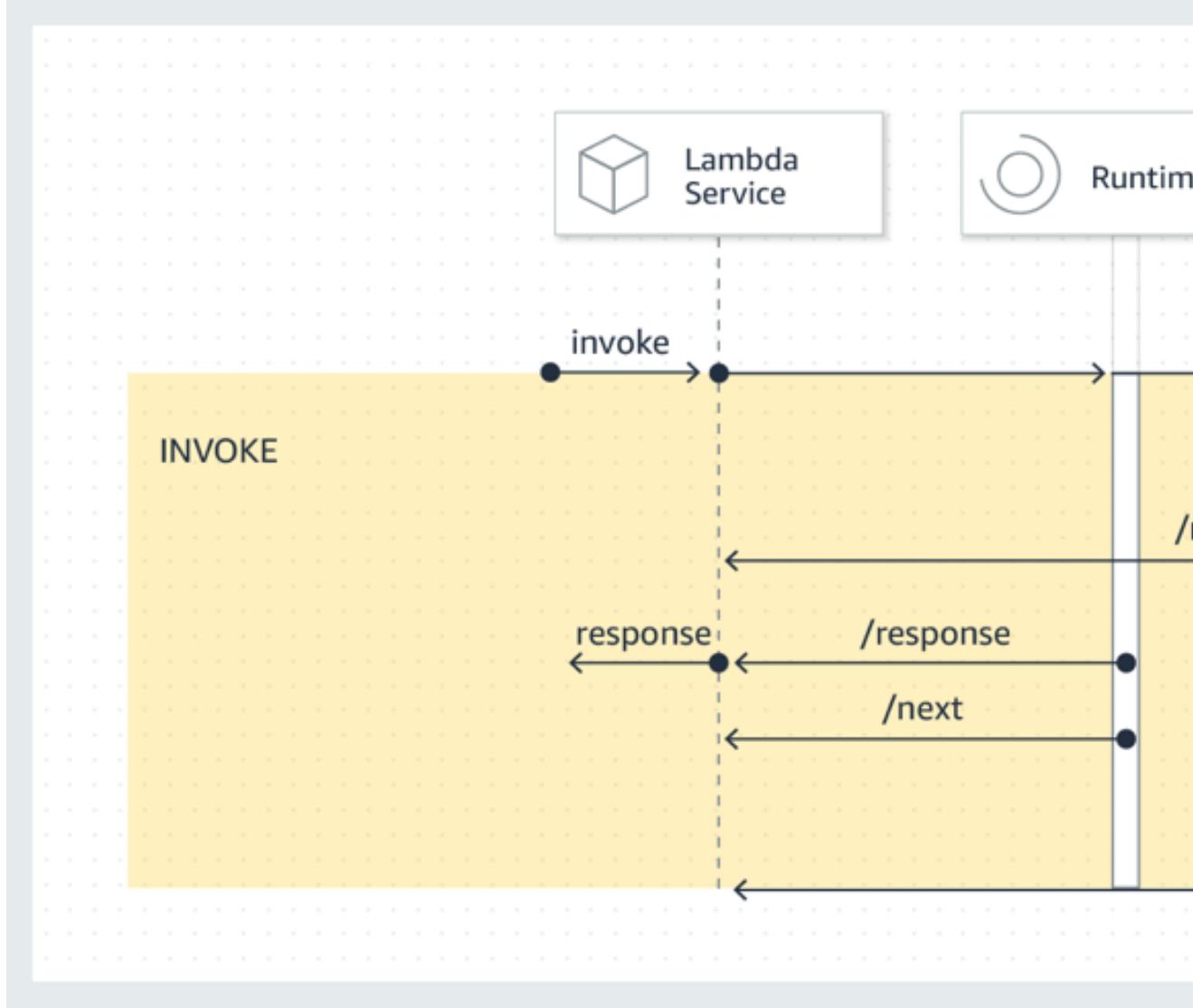
Invoke phase

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": "INVOCATION",
    "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 \(p. 70\)](#) 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 AWS Lambda function metrics \(p. 691\)](#).

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

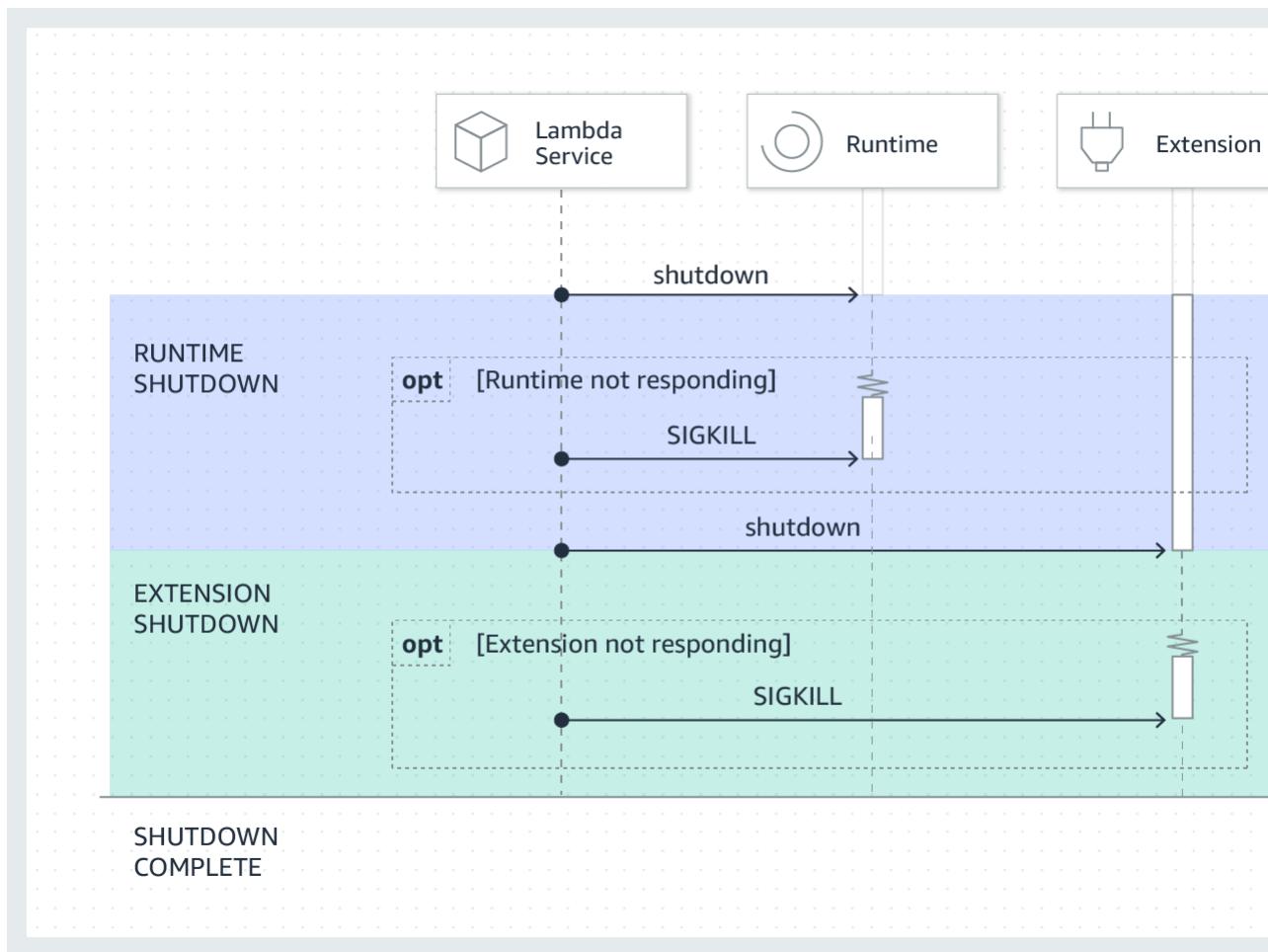
When Lambda is about to shut down the runtime, it sends a `Shutdown` event to the runtime and then to each registered external extension. Extensions can use this time for final cleanup tasks. The `Shutdown` event is sent in response to a `Next` API request.

Duration limit: The maximum duration of the `Shutdown` phase depends on the configuration of registered extensions:

- 300 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 \(p. 77\)](#), except for the following variables that are specific to the runtime process:

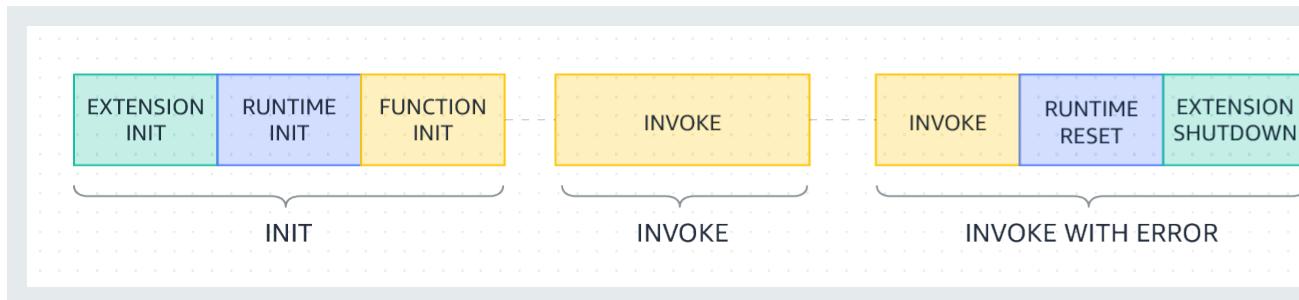
- AWS_EXECUTION_ENV
- AWS_LAMBDA_LOG_GROUP_NAME
- AWS_LAMBDA_LOG_STREAM_NAME
- AWS_XRAY_CONTEXT_MISSING
- AWS_XRAY_DAEMON_ADDRESS
- LAMBDA_RUNTIME_DIR
- LAMBDA_TASK_ROOT
- _AWS_XRAY_DAEMON_ADDRESS
- _AWS_XRAY_DAEMON_PORT
- _HANDLER

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.



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 \(p. 218\)](#)
- [Next \(p. 219\)](#)
- [Init error \(p. 219\)](#)
- [Exit error \(p. 221\)](#)

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

Headers

`Lambda-Extension-Name` – The full file name of the extension. Required: yes. Type: string.

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"  
}
```

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

Parameters

`Lambda-Extension-Identifier` – Unique identifier for extension (UUID string). Required: yes. Type: UUID string.

Response header

- `Lambda-Extension-Identifier` – Unique agent identifier (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

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:

- Runtime.NoSuchHandler
- Extension.APIKeyNotFound
- Extension.ConfigInvalid
- Extension.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

- **Lambda-Extension-Identifier** – Unique agent identifier (UUID string).

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

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:

- Runtime.NoSuchHandler
- Extension.APIKeyNotFound
- Extension.ConfigInvalid
- Extension.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

- **Lambda-Extension-Identifier** – Unique agent identifier (UUID string).

Response codes

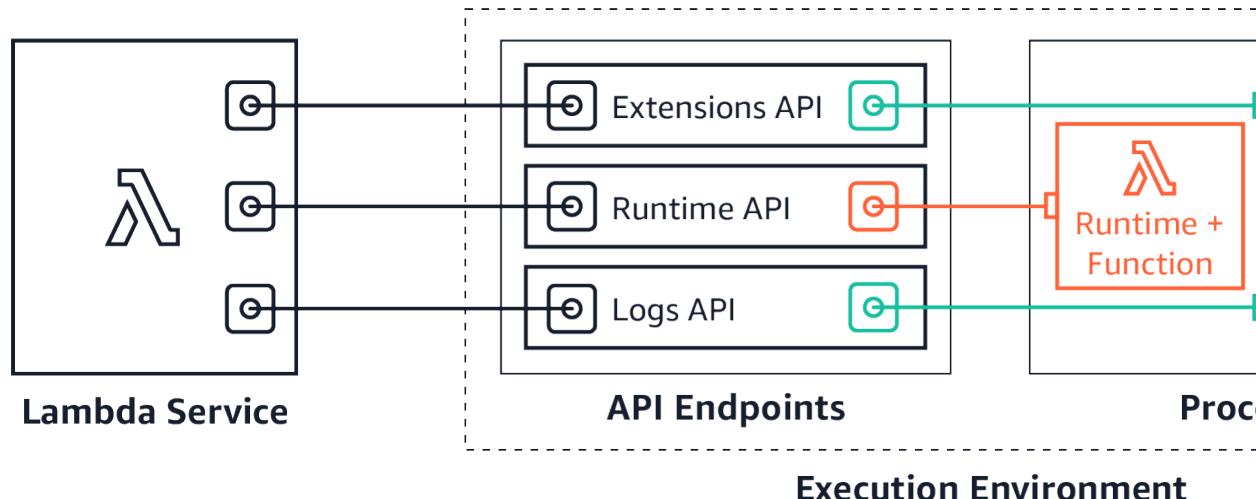
- 202 – Accepted

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

Lambda Logs API

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 \(p. 209\)](#) can use the Lambda Runtime Logs API to subscribe to log streams directly from within the Lambda [execution environment \(p. 200\)](#). 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 \(p. 224\)](#)
- [Memory usage \(p. 224\)](#)
- [Destination protocols \(p. 224\)](#)
- [Buffering configuration \(p. 224\)](#)
- [Example subscription \(p. 225\)](#)
- [Sample code for Logs API \(p. 225\)](#)
- [Logs API reference \(p. 226\)](#)
- [Log messages \(p. 227\)](#)

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 \(p. 218\)](#) to receive the extension identifier. Then subscribe to the Logs API during [initialization \(p. 201\)](#). 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 \(p. 224\)](#). 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": 10240,  
    "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. 209).

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” \(p. 224\)](#). Required: yes. Type: strings.

`buffering` – See [the section called “Buffering configuration” \(p. 224\)](#). 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 \(p. 229\)](#) 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 \(p. 227\)](#)
- [Extension logs \(p. 227\)](#)
- [Platform logs \(p. 227\)](#)

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 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
             }
}
```

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"  
    }  
}
```

Modifying the runtime environment

You can use [internal extensions \(p. 160\)](#) 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 \(p. 77\)](#) that you can set to add options and tools to the runtime. Lambda also provides [wrapper scripts \(p. 233\)](#), 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 11 and Java 8 (java8.a12), 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 \(p. 218\)](#). 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.

The following [Lambda runtimes \(p. 195\)](#) support wrapper scripts:

- Node.js 14.x
- Node.js 12.x

- Node.js 10.x
- Python 3.8
- Ruby 2.7
- Java 11
- Java 8 (java8.al2)
- .NET Core 3.1

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 \(p. 101\)](#).
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 \(p. 17\)](#), 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
...
```

Custom AWS 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 in the form of an executable file named `bootstrap`.

A runtime is responsible for running the function's setup code, reading the handler name from an environment variable, and reading 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 \(p. 195\)](#). 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 \(p. 239\)](#). You can also explore a custom runtime implemented in C++ at [awslabs/aws-lambda-cpp](#) on GitHub.

Topics

- [Using a custom runtime \(p. 236\)](#)
- [Building a custom runtime \(p. 236\)](#)

Using a custom runtime

To use a custom runtime, set your function's runtime to `provided`. The runtime can be included in your function's deployment package, or in a [layer \(p. 101\)](#).

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 terminated. The initialization tasks run once [per instance of the function \(p. 200\)](#) 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.

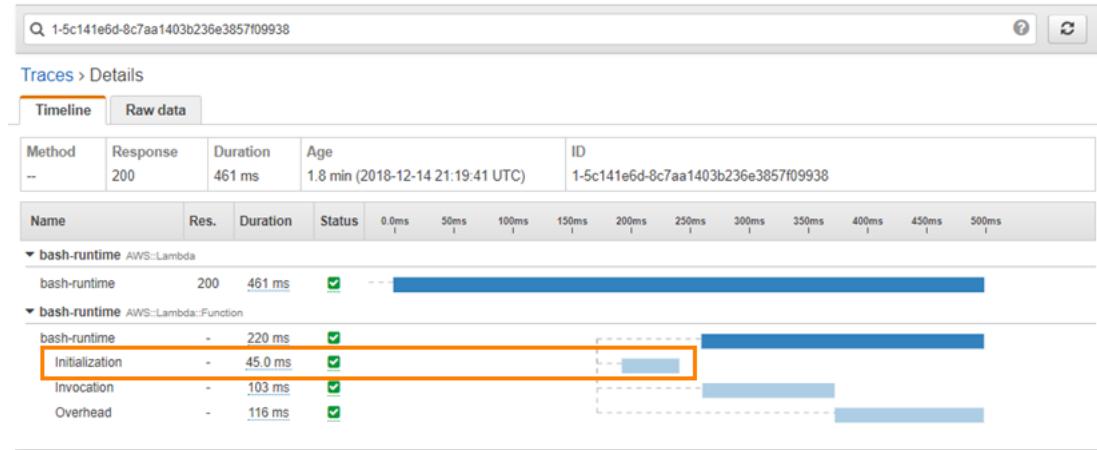
See [Defined runtime environment variables \(p. 80\)](#) for a full list of available variables.

- **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 \(p. 206\)](#) 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 \(p. 450\)](#).

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 \(p. 205\)](#) 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 \(p. 205\)](#) 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 \(p. 206\)](#) API to post the response from the handler.

- **Handle errors** – If an error occurs, call the [invocation error \(p. 207\)](#) 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 \(p. 239\)](#).

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 [Getting started with Lambda \(p. 9\)](#) to create your first Lambda function.

To complete the following steps, you need a command line terminal or shell to run commands. 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. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

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 -LD "$HEADERS" -X GET "http://${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 \(p. 236\)](#).

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-west-2:123456789012:function:bash-runtime",
    "Runtime": "provided",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Handler": "function.handler",
    "CodeSha256": "mv/xRv84LPCxdpcbKvmwuuFzwo7sLwUO1VxcUv3wK1M=",
    "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-west-2-layers.s3.us-west-2.amazonaws.com/
snapshots/123456789012/bash-runtime-018c209b...", 
        "CodeSha256": "bXVLhHi+D3H1QbDARUVPrDwlC7bssPxySQqt1QZqusE=",
        "CodeSize": 584,
        "UncompressedCodeSize": 0
    },
    "LayerArn": "arn:aws:lambda:us-west-2:123456789012:layer:bash-runtime",
    "LayerVersionArn": "arn:aws:lambda:us-west-2:123456789012: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-west-2:123456789012:layer:bash-runtime:1  
{  
    "FunctionName": "bash-runtime",  
    "Layers": [  
        {  
            "Arn": "arn:aws:lambda:us-west-2:123456789012: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-west-2:123456789012: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  
{  
    "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-west-2:123456789012: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-west-2:123456789012: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.

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.

Using container images with Lambda

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.

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

- [Creating Lambda container images \(p. 248\)](#)
- [Testing Lambda container images locally \(p. 254\)](#)

Creating Lambda container images

You can package your Lambda function code and dependencies as a container image, using tools such as the Docker CLI. You can then upload the image to your container registry hosted on Amazon Elastic Container Registry (Amazon ECR). Note that you must create the Lambda function from the same account as the container registry in Amazon ECR.

AWS provides a set of open-source [base images \(p. 203\)](#) that you can use to create your container image. These base images include a [runtime interface client \(p. 203\)](#) to manage the interaction between Lambda and your function code.

You can also use an alternative base image from another container registry. Lambda provides open-source runtime interface clients that you add to an alternative base image to make it compatible with Lambda.

For example applications, including a Node.js example and a Python example, see [Container image support for Lambda](#) on the AWS Blog.

After you create a container image in the Amazon ECR container registry, you can [create and run \(p. 72\)](#) the Lambda function.

Topics

- [Image types \(p. 248\)](#)
- [Container tools \(p. 248\)](#)
- [Lambda requirements for container images \(p. 249\)](#)
- [Container image settings \(p. 249\)](#)
- [Create an image from an AWS base image for Lambda \(p. 249\)](#)
- [Create an image from an alternative base image \(p. 251\)](#)
- [Create an image using the AWS SAM toolkit \(p. 253\)](#)

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 images up to 10 GB in size.

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.

Lambda requirements for container images

To deploy a container image to Lambda, note the following requirements:

1. The container image must implement the Lambda [Runtime API \(p. 205\)](#). The AWS open-source [runtime interface clients \(p. 203\)](#) implement the API. You can add a runtime interface client to your preferred base image to make it compatible with Lambda.
2. The container image must be able to run on a read-only file system. Your function code can access a writable `/tmp` directory with 512 MB of storage.
3. 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.
4. Lambda supports only Linux-based 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 `ENTRYPOINT`.
- `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 `ENTRYPOINT` and `CMD`, see [Demystifying ENTRYPOINT 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 `ENTRYPOINT` 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.

Create an image from an AWS base image for Lambda

To build a container image for a new Lambda function, you can start with an AWS base image for 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 Docker Hub. To use the updated base image, you must rebuild your container image and [update the function code \(p. 74\)](#).

Prerequisites

- The AWS Command Line Interface (AWS CLI)

The following instructions use the AWS CLI to call AWS service API operations. To install the AWS CLI, see [Installing, updating, and uninstalling the AWS CLI](#) in the *AWS Command Line Interface User Guide*.

- Docker Desktop

The following instructions use Docker CLI commands to create the container image. To install the Docker CLI, see [Get Docker](#) on the Docker Docs website.

- Your function code

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`

The following shows an example Dockerfile for Node.js version 14. :

```
FROM public.ecr.aws/lambda/nodejs:14
# Alternatively, you can pull the base image from Docker Hub: amazon/aws-lambda-
nodejs:12

# Assumes your function is named "app.js", and there is a package.json file in the app
# directory.
COPY app.js package.json /var/task/

# 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" ]
```

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 \(p. 254\)](#). 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.

7. Authenticate the Docker CLI to your Amazon ECR registry.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 123456789012.dkr.ecr.us-east-1.amazonaws.com
```

8. 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
```

9. Tag your image to match your repository name, and deploy the image to Amazon ECR using the `docker push` command.

```
docker tag hello-world:latest 123456789012.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
docker push 123456789012.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

Now that your container image resides in the Amazon ECR container registry, you can [create and run](#) (p. 72) the Lambda function.

Create an image from an alternative base image

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 \(p. 254\)](#).
7. Authenticate the Docker CLI to your Amazon ECR registry.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-
stdin 123456789012.dkr.ecr.us-east-1.amazonaws.com
```

8. 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
```

9. Tag your image to match your repository name, and deploy the image to Amazon ECR using the `docker push` command.

```
docker tag hello-world:latest 123456789012.dkr.ecr.us-east-1.amazonaws.com/hello-
world:latest
docker push 123456789012.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

Now that your container image resides in the Amazon ECR container registry, you can [create and run \(p. 72\)](#) the Lambda function.

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*.

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.

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 \(p. 254\)](#)
- [Environment variables \(p. 254\)](#)
- [Test an image with RIE included in the image \(p. 255\)](#)
- [Build RIE into your base image \(p. 255\)](#)
- [Test an image without adding RIE to the image \(p. 256\)](#)

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.
- The emulator supports only Linux x86-64 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 \(p. 77\)](#) 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 if you built 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. 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 \(p. 203\)](#). 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](#) from GitHub into your project directory.
3. Copy the script, install the emulator package, and change `ENTRYPOINT` to run the new script by adding the following lines to your Dockerfile:

```
COPY ./entry_script.sh /entry_script.sh
ADD aws-lambda-rie /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 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 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.

Using AWS Lambda with other services

AWS Lambda integrates with other AWS services to invoke functions. You can configure triggers to invoke a function in response to resource lifecycle events, respond to incoming HTTP requests, consume events from a queue, or [run on a schedule \(p. 295\)](#).

Each service that integrates with Lambda sends data to your function in JSON as an event. The structure of the event document is different for each event type, and contains data about the resource or request that triggered the function. Lambda runtimes convert the event into an object and pass it to your function.

The following example shows a test event from an [Application Load Balancer \(p. 349\)](#) that represents a GET request to `/lambda?query=1234ABCD`.

Example event from an Application Load Balancer

```
{
  "requestContext": {
    "elb": {
      "targetGroupArn": "arn:aws:elasticloadbalancing:us-east-2:123456789012: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-2.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
}
```

Note

The Lambda runtime converts the event document into an object and passes it to your [function handler \(p. 29\)](#). For compiled languages, Lambda provides definitions for event types in a library. See the following topics for more information.

- [Building Lambda functions with Java \(p. 569\)](#)

- [Building Lambda functions with Go \(p. 609\)](#)
- [Building Lambda functions with C# \(p. 637\)](#)
- [Building Lambda functions with PowerShell \(p. 666\)](#)

For services that generate a queue or data stream, you create an [event source mapping \(p. 148\)](#) in Lambda and grant Lambda permission to access the other service in the [execution role \(p. 47\)](#). Lambda reads data from the other service, creates an event, and invokes your function.

Services that Lambda reads events from

- [Amazon DynamoDB \(p. 313\)](#)
- [Amazon Kinesis \(p. 365\)](#)
- [Amazon MQ \(p. 389\)](#)
- [Amazon Managed Streaming for Apache Kafka \(p. 395\)](#)
- [self-managed Apache Kafka \(p. 356\)](#)
- [Amazon Simple Queue Service \(p. 438\)](#)

Other services invoke your function directly. You grant the other service permission in the function's [resource-based policy \(p. 51\)](#), and configure the other service to generate events and invoke your function. Depending on the service, the invocation can be synchronous or asynchronous. For synchronous invocation, the other service waits for the response from your function and might [retry on errors \(p. 158\)](#).

For more information about Lambda service architectures, see [Event driven architectures](#) in the *Lambda operator guide*.

Services that invoke Lambda functions synchronously

- [Elastic Load Balancing \(Application Load Balancer\) \(p. 349\)](#)
- [Amazon Cognito \(p. 310\)](#)
- [Amazon connect \(p. 312\)](#)
- [Amazon Lex \(p. 386\)](#)
- [Amazon Alexa \(p. 260\)](#)
- [Amazon API Gateway \(p. 261\)](#)
- [Amazon CloudFront \(Lambda@Edge\) \(p. 305\)](#)
- [Amazon Kinesis Data Firehose \(p. 364\)](#)
- [Amazon Simple Storage Service Batch \(p. 426\)](#)
- [Secrets Manager \(p. 428\)](#)

For asynchronous invocation, Lambda queues the event before passing it to your function. The other service gets a success response as soon as the event is queued and isn't aware of what happens afterwards. If an error occurs, Lambda handles [retries \(p. 158\)](#), and can send failed events to a [destination \(p. 142\)](#) that you configure.

Services that invoke Lambda functions asynchronously

- [Amazon Simple Storage Service \(p. 406\)](#)
- [Amazon Simple Notification Service \(p. 431\)](#)
- [Amazon Simple Email Service \(p. 429\)](#)
- [AWS CloudFormation \(p. 303\)](#)
- [Amazon CloudWatch Logs \(p. 302\)](#)

- [Amazon CloudWatch Events \(p. 295\)](#)
- [AWS CodeCommit \(p. 307\)](#)
- [AWS Config \(p. 311\)](#)
- [AWS IoT \(p. 353\)](#)
- [AWS IoT Events \(p. 354\)](#)
- [AWS CodePipeline \(p. 308\)](#)

Additionally, some services integrate with Lambda in other ways that don't involve invoking functions.

Services that integrate with Lambda in other ways

- [Amazon Elastic File System \(p. 351\)](#)
- [AWS X-Ray \(p. 450\)](#)

See the topics in this chapter for more details about each service, and example events that you can use to test your function.

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 Node.js](#)
- [Alexa Skills Kit SDK for Java](#)

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.

To add a public endpoint to your Lambda function

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Under **Functional overview**, choose **Add trigger**.
4. Select **API Gateway**.
5. For **API**, choose **Create an API**.
6. For **Security**, choose **Open**.
7. Choose **Add**.

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.

Amazon API Gateway invokes your function [synchronously](#) (p. 138) 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
}

```

This example shows an event for a GET request to the root path of the Prod stage of a REST API. Event shape and contents vary by [API type \(p. 265\)](#) and configuration.

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-cccecxmplbae116148e52f036

```

```
<
{
  "TotalCodeSize": 104330022,
  "FunctionCount": 26
}
```

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.

Sections

- [Permissions \(p. 263\)](#)
- [Handling errors with an API Gateway API \(p. 264\)](#)
- [Choosing an API type \(p. 265\)](#)
- [Sample applications \(p. 267\)](#)
- [Tutorial: Using AWS Lambda with Amazon API Gateway \(p. 267\)](#)
- [Sample function code \(p. 277\)](#)
- [Create a simple microservice using Lambda and API Gateway \(p. 280\)](#)
- [AWS SAM template for an API Gateway application \(p. 281\)](#)

Permissions

Amazon API Gateway gets permission to invoke your function from the function's [resource-based policy \(p. 51\)](#). 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 example shows a function policy with a statement that was added by an AWS SAM template.

Example function policy

```
{
  "Version": "2012-10-17",
  "Id": "default",
  "Statement": [
    {
      "Sid": "nodejs-apig-functiongetEndpointPermissionProd-BWDBXMPLX2F",
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-2:123456789012:function:nodejs-apig-
function-1G3MXMPLXVXYI",
      "Condition": {
        "ArnLike": {
          "AWS:SourceArn": "arn:aws:execute-api:us-east-2:123456789012:ktyvxmpls1/*/*"
        }
      }
    }
  ]
}
```

Confirm the function policy ([p. 51](#)) in the **Permissions** tab of the Lambda console.

You can manage function policy permissions manually with the following API operations:

- [AddPermission \(p. 742\)](#)
- [RemovePermission \(p. 905\)](#)
- [GetPolicy \(p. 834\)](#)

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

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 `mnh1xmpli7`. You can use an asterisk in the source ARN to grant permissions to multiple stages, methods, or resources.

Resource patterns

- `mnh1xmpli7/*/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 \(p. 55\)](#).

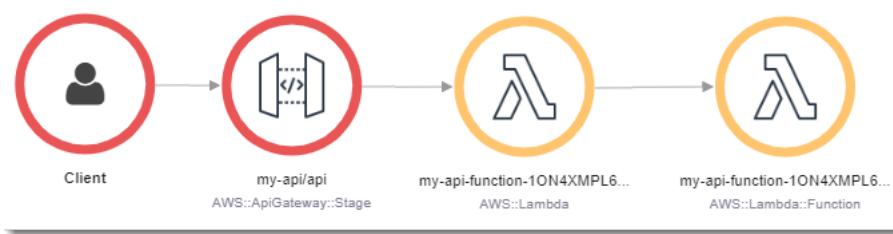
Handling errors with an API Gateway API

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.



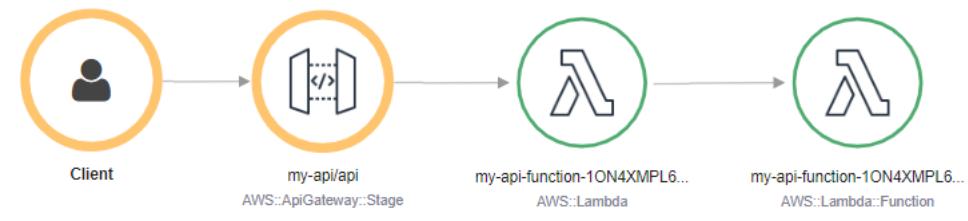
To customize the error response, you must catch errors in your code and format a response in the required format.

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": "JKJaXmPLvHcESEA=",
    "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 \(p. 39\)](#) and [templates \(p. 40\)](#) that you can use to create an API Gateway application in the Lambda console.

Tutorial: Using AWS Lambda with Amazon API Gateway

In this example you create a simple API using Amazon API Gateway. An Amazon API Gateway is a collection of resources and methods. For this tutorial, you create one resource (`DynamoDBManager`) and define one method (`POST`) on it. The method is backed by a Lambda function (`LambdaFunctionOverHttps`). That is, when you call the API through an HTTPS endpoint, Amazon API Gateway invokes the Lambda function.

The `POST` method on the `DynamoDBManager` resource supports the following DynamoDB operations:

- Create, update, and delete an item.
- Read an item.
- Scan an item.
- Other operations (echo, ping), not related to DynamoDB, that you can use for testing.

The request payload you send in the `POST` request identifies the DynamoDB operation and provides necessary data. For example:

- The following is a sample request payload for a DynamoDB create item operation:

```
{
    "operation": "create",
    "tableName": "lambda-apigateway",
    "payload": {
        "Item": {
            "id": "1",
            "name": "Bob"
        }
    }
}
```

- The following is a sample request payload for a DynamoDB read item operation:

```
{  
    "operation": "read",  
    "tableName": "lambda-apigateway",  
    "payload": {  
        "Key": {  
            "id": "1"  
        }  
    }  
}
```

- The following is a sample request payload for an echo operation. You send an HTTP POST request to the endpoint, using the following data in the request body.

```
{  
    "operation": "echo",  
    "payload": {  
        "somekey1": "somevalue1",  
        "somekey2": "somevalue2"  
    }  
}
```

Note

API Gateway offers advanced capabilities, such as:

- Pass through the entire request** – A Lambda function can receive the entire HTTP request (instead of just the request body) and set the HTTP response (instead of just the response body) using the AWS_PROXY integration type.
- Catch-all methods** – Map all methods of an API resource to a single Lambda function with a single mapping, using the ANY catch-all method.
- Catch-all resources** – Map all sub-paths of a resource to a Lambda function without any additional configuration using the new path parameter ({proxy+}).

To learn more about these API Gateway features, see [Configure proxy integration for a proxy resource](#).

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 [Getting started with Lambda \(p. 9\)](#) to create your first Lambda function.

To complete the following steps, you need a command line terminal or shell to run commands. 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. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the execution role

Create the [execution role](#) (p. 47). This role uses a custom policy to give your function permission to access the required AWS resources. You first create the policy and then create the execution role.

To create a custom policy

1. Open the [policy page](#) in the IAM console.
2. Choose **Create Policy**.
3. Choose the **JSON** tab. Paste the following custom policy into the input box.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "Stmt1428341300017",  
            "Action": [  
                "dynamodb>DeleteItem",  
                "dynamodb>GetItem",  
                "dynamodb>PutItem",  
                "dynamodb>Query",  
                "dynamodb>Scan",  
                "dynamodb>UpdateItem"  
            ],  
            "Effect": "Allow",  
            "Resource": "*"  
        },  
        {  
            "Sid": "",  
            "Resource": "*",  
            "Action": [  
                "logs>CreateLogGroup",  
                "logs>CreateLogStream",  
                "logs>PutLogEvents"  
            ],  
            "Effect": "Allow"  
        }  
    ]  
}
```

4. Choose **Next: Tags**.
5. Choose **Next: Review**.
6. For the policy name, enter **lambda-apigateway-policy**.
7. Enter **Create policy**.

This policy includes permissions for the function to access DynamoDB and CloudWatch Logs.

To create an execution role

1. Open the [roles page](#) in the IAM console.
2. Choose **Create role**.
3. For the trusted entity, choose **AWS Service**, and for the use case choose **Lambda**.
4. Choose **Next: Permissions**.
5. In the policy search box, enter **lambda-apigateway-policy**.
6. In the results, select **lambda-apigateway-policy** and choose **Next: Tags**.
7. Choose **Next: Review**.
8. For the role name, enter **lambda-apigateway-role**.

9. Enter **Create role**.

Note the Amazon Resource Name (ARN) of the execution role for later use.

Create the function

The following example code receives a API Gateway 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 \(p. 277\)](#).

Example index.js

```
console.log('Loading function');

var AWS = require('aws-sdk');
var dynamo = new AWS.DynamoDB.DocumentClient();

/**
 * Provide an event that contains the following keys:
 *
 * - operation: one of the operations in the switch statement below
 * - tableName: required for operations that interact with DynamoDB
 * - payload: a parameter to pass to the operation being performed
 */
exports.handler = function(event, context, callback) {
    //console.log('Received event:', JSON.stringify(event, null, 2));

    var operation = event.operation;

    if (event.tableName) {
        event.payload.TableName = event.tableName;
    }

    switch (operation) {
        case 'create':
            dynamo.put(event.payload, callback);
            break;
        case 'read':
            dynamo.get(event.payload, callback);
            break;
        case 'update':
            dynamo.update(event.payload, callback);
            break;
        case 'delete':
            dynamo.delete(event.payload, callback);
            break;
        case 'list':
            dynamo.scan(event.payload, callback);
            break;
        case 'echo':
            callback(null, "Success");
            break;
        case 'ping':
            callback(null, "pong");
            break;
        default:
            callback(`Unknown operation: ${operation}`);
    }
};
```

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. For the `role` parameter, enter the ARN of the execution role that you created earlier.

```
aws lambda create-function --function-name LambdaFunctionOverHttps \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs12.x \
--role arn:aws:iam::123456789012:role/service-role/lambda-apigateway-role
```

Test the Lambda function

Invoke the function manually using the sample event data. We recommend that you invoke the function using the console because the console UI provides a user-friendly interface for reviewing the execution results, including the execution summary, logs written by your code, and the results returned by the function (because the console always performs synchronous execution—invokes the Lambda function using the `RequestResponse` invocation type).

To test the Lambda function

1. Copy the following JSON into a file and save it as `input.txt`.

```
{
  "operation": "echo",
  "payload": {
    "somekey1": "somevalue1",
    "somekey2": "somevalue2"
  }
}
```

2. Run the following `invoke` command:

```
aws lambda invoke --function-name LambdaFunctionOverHttps \
--payload file://input.txt outputfile.txt
```

The `cli-binary-format` option is required if you are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#).

.

Create an API using Amazon API Gateway

In this step, you associate your Lambda function with a method in the API that you created using Amazon API Gateway and test the end-to-end experience. That is, when an HTTP request is sent to an API method, Amazon API Gateway invokes your Lambda function.

First, you create an API (`DynamoDBOperations`) using Amazon API Gateway with one resource (`DynamoDBManager`) and one method (`POST`). You associate the `POST` method with your Lambda function. Then, you test the end-to-end experience.

Create the API

Run the following `create-rest-api` command to create the `DynamoDBOperations` API for this tutorial.

```
aws apigateway create-rest-api --name DynamoDBOperations
```

You should see the following output:

```
{  
    "id": "bs8fqo6bp0",  
    "name": "DynamoDBOperations",  
    "createdDate": 1539803980,  
    "apiKeySource": "HEADER",  
    "endpointConfiguration": {  
        "types": [  
            "EDGE"  
        ]  
    }  
}
```

Save the API ID for use in further commands. You also need the ID of the API root resource. To get the ID, run the `get-resources` command.

```
API=bs8fqo6bp0  
aws apigateway get-resources --rest-api-id $API
```

You should see the following output:

```
{  
    "items": [  
        {  
            "path": "/",  
            "id": "e8kitthgdb"  
        }  
    ]  
}
```

At this time you only have the root resource, but you add more resources in the next step.

Create a resource in the API

Run the following `create-resource` command to create a resource (`DynamoDBManager`) in the API that you created in the preceding section.

```
aws apigateway create-resource --rest-api-id $API --path-part DynamoDBManager \  
--parent-id e8kitthgdb
```

You should see the following output:

```
{  
    "path": "/DynamoDBManager",  
    "pathPart": "DynamoDBManager",  
    "id": "iuig5w",  
    "parentId": "e8kitthgdb"  
}
```

Note the ID in the response. This is the ID of the `DynamoDBManager` resource that you created.

Create POST method on the resource

Run the following `put-method` command to create a `POST` method on the `DynamoDBManager` resource in your API.

```
RESOURCE=iuig5w
aws apigateway put-method --rest-api-id $API --resource-id $RESOURCE \
--http-method POST --authorization-type NONE
```

You should see the following output:

```
{
  "apiKeyRequired": false,
  "httpMethod": "POST",
  "authorizationType": "NONE"
}
```

We specify `NONE` for the `--authorization-type` parameter, which means that unauthenticated requests for this method are supported. This is fine for testing but in production you should use either the key-based or role-base authentication.

Set the Lambda function as the destination for the POST method

Run the following command to set the Lambda function as the integration point for the `POST` method. This is the method Amazon API Gateway invokes when you make an HTTP request for the `POST` method endpoint. This command and others use ARNs that include your account ID and region. Save these to variables (you can find your account ID in the role ARN that you used to create the function).

```
REGION=us-east-2
ACCOUNT=123456789012
aws apigateway put-integration --rest-api-id $API --resource-id $RESOURCE \
--http-method POST --type AWS --integration-http-method POST \
--uri arn:aws:apigateway:$REGION:lambda:path/2015-03-31/functions/arn:aws:lambda:$REGION:
$ACCOUNT:function:LambdaFunctionOverHttps/invocations
```

You should see the following output:

```
{
  "type": "AWS",
  "httpMethod": "POST",
  "uri": "arn:aws:apigateway:us-east-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-2:123456789012:function:LambdaFunctionOverHttps/invocations",
  "passthroughBehavior": "WHEN_NO_MATCH",
  "timeoutInMillis": 29000,
  "cacheNamespace": "iuig5w",
  "cacheKeyParameters": []
}
```

`--integration-http-method` is the method that API Gateway uses to communicate with AWS Lambda. `--uri` is unique identifier for the endpoint to which Amazon API Gateway can send request.

Set `content-type` of the `POST` method response and integration response to JSON as follows:

- Run the following command to set the `POST` method response to JSON. This is the response type that your API method returns.

```
aws apigateway put-method-response --rest-api-id $API \
```

```
--resource-id $RESOURCE --http-method POST \
--status-code 200 --response-models application/json=Empty
```

You should see the following output:

```
{
  "statusCode": "200",
  "responseModels": {
    "application/json": "Empty"
  }
}
```

- Run the following command to set the `POST` method integration response to JSON. This is the response type that Lambda function returns.

```
aws apigateway put-integration-response --rest-api-id $API \
--resource-id $RESOURCE --http-method POST \
--status-code 200 --response-templates application/json=""
```

You should see the following output:

```
{
  "statusCode": "200",
  "responseTemplates": {
    "application/json": null
  }
}
```

Note

If you encounter an error running this command, you can use escape characters around the response template field for more clarity. The text `application/json=""` becomes `"{\"application/json\": \"\"}""`.

Deploy the API

In this step, you deploy the API that you created to a stage called `prod`.

```
aws apigateway create-deployment --rest-api-id $API --stage-name prod
```

You should see the following output:

```
{
  "id": "20vgsz",
  "createdDate": 1539820012
}
```

Grant invoke permission to the API

Now that you have an API created using Amazon API Gateway and you've deployed it, you can test. First, you need to add permissions so that Amazon API Gateway can invoke your Lambda function when you send HTTP request to the `POST` method.

To do this, you need to add a permission to the permissions policy associated with your Lambda function. Run the following `add-permission` AWS Lambda command to grant the Amazon API Gateway service principal (`apigateway.amazonaws.com`) permissions to invoke your Lambda function (`LambdaFunctionOverHttps`).

```
aws lambda add-permission --function-name LambdaFunctionOverHttps \
--statement-id apigateway-test-2 --action lambda:InvokeFunction \
--principal apigateway.amazonaws.com \
--source-arn "arn:aws:execute-api:$REGION:$ACCOUNT:$API/*/POST/DynamoDBManager"
```

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:LambdaFunctionOverHttps\",\"Condition\":  
  \"\":{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:execute-api:us-east-2:123456789012:mnh1yprki7/  
  */POST/DynamoDBManager\"}}}"  
}
```

You must grant this permission to enable testing (if you go to the Amazon API Gateway and choose **Test** to test the API method, you need this permission). Note the --source-arn specifies a wildcard character (*) as the stage value (indicates testing only). This allows you to test without deploying the API.

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.

Now, run the same command again, but this time you grant to your deployed API permissions to invoke the Lambda function.

```
aws lambda add-permission --function-name LambdaFunctionOverHttps \
--statement-id apigateway-prod-2 --action lambda:InvokeFunction \
--principal apigateway.amazonaws.com \
--source-arn "arn:aws:execute-api:$REGION:$ACCOUNT:$API/prod/POST/DynamoDBManager"
```

You should see the following output:

```
{  
  "Statement": "{\"Sid\":\"apigateway-prod-2\",\"Effect\":\"Allow\",\"Principal\":  
  {\"Service\":\"apigateway.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",\"Resource\":  
  \"arn:aws:lambda:us-east-2:123456789012:function:LambdaFunctionOverHttps\",\"Condition\":  
  \"\":{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:execute-api:us-east-2:123456789012:mnh1yprki7/  
  prod/POST/DynamoDBManager\"}}}"  
}
```

You grant this permission so that your deployed API has permissions to invoke the Lambda function. Note that the --source-arn specifies a prod which is the stage name we used when deploying the API.

Create a Amazon DynamoDB table

Create the DynamoDB table that the Lambda function uses.

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-apigateway**
 - **Primary key – id (string)**
4. Choose **Create**.

Trigger the function with an HTTP request

In this step, you are ready to send an HTTP request to the `POST` method endpoint. You can use either Curl or a method (`test-invoke-method`) provided by Amazon API Gateway.

You can use Amazon API Gateway CLI commands to send an HTTP `POST` request to the resource (`DynamoDBManager`) endpoint. Because you deployed your Amazon API Gateway, you can use Curl to invoke the methods for the same operation.

The Lambda function supports using the `create` operation to create an item in your DynamoDB table. To request this operation, use the following JSON:

Example `create-item.json`

```
{  
    "operation": "create",  
    "tableName": "lambda-apigateway",  
    "payload": {  
        "Item": {  
            "id": "1234ABCD",  
            "number": 5  
        }  
    }  
}
```

Save the test input to a file named `create-item.json`. Run the `test-invoke-method` Amazon API Gateway command to send an HTTP `POST` method request to the resource (`DynamoDBManager`) endpoint.

```
aws apigateway test-invoke-method --rest-api-id $API \  
--resource-id $RESOURCE --http-method POST --path-with-query-string "" \  
--body file://create-item.json
```

Or, you can use the following Curl command:

```
curl -X POST -d "{\"operation\":\"create\", \"tableName\":\"lambda-apigateway\",  
\"payload\":{\"Item\":{\"id\":\"1\", \"name\":\"Bob\"}}}" https://$API.execute-api.  
$REGION.amazonaws.com/prod/DynamoDBManager
```

To send request for the `echo` operation that your Lambda function supports, you can use the following request payload:

Example `echo.json`

```
{  
    "operation": "echo",  
    "payload": {  
        "somekey1": "somevalue1",  
        "somekey2": "somevalue2"  
    }  
}
```

Save the test input to a file named `echo.json`. Run the `test-invoke-method` Amazon API Gateway CLI command to send an HTTP `POST` method request to the resource (`DynamoDBManager`) endpoint using the preceding JSON in the request body.

```
aws apigateway test-invoke-method --rest-api-id $API \  
--body file://echo.json
```

```
--resource-id $RESOURCE --http-method POST --path-with-query-string "" \
--body file://echo.json
```

Or, you can use the following Curl command:

```
curl -X POST -d "{\"operation\":\"echo\", \"payload\":{\"somekey1\":\"somevalue1\",
\"somekey2\":\"somevalue2\"}}" https://$API.execute-api.$REGION.amazonaws.com/prod/
DynamoDBManager
```

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. 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 role**.
4. Choose **Yes, 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**.

Sample function code

Sample code is available for the following languages.

Topics

- [Node.js \(p. 278\)](#)
- [Python 3 \(p. 279\)](#)
- [Go \(p. 279\)](#)

Node.js

The following example processes messages from API Gateway, and manages DynamoDB documents based on the request method.

Example index.js

```
console.log('Loading function');

var AWS = require('aws-sdk');
var dynamo = new AWS.DynamoDB.DocumentClient();

/**
 * Provide an event that contains the following keys:
 *
 * - operation: one of the operations in the switch statement below
 * - tableName: required for operations that interact with DynamoDB
 * - payload: a parameter to pass to the operation being performed
 */
exports.handler = function(event, context, callback) {
    //console.log('Received event:', JSON.stringify(event, null, 2));

    var operation = event.operation;

    if (event.tableName) {
        event.payload.TableName = event.tableName;
    }

    switch (operation) {
        case 'create':
            dynamo.put(event.payload, callback);
            break;
        case 'read':
            dynamo.get(event.payload, callback);
            break;
        case 'update':
            dynamo.update(event.payload, callback);
            break;
        case 'delete':
            dynamo.delete(event.payload, callback);
            break;
        case 'list':
            dynamo.scan(event.payload, callback);
            break;
        case 'echo':
            callback(null, "Success");
            break;
        case 'ping':
            callback(null, "pong");
            break;
        default:
            callback(`Unknown operation: ${operation}`);
    }
};
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Node.js Lambda functions with .zip file archives \(p. 488\)](#).

Python 3

The following example processes messages from API Gateway, and manages DynamoDB documents based on the request method.

Example LambdaFunctionOverHttps.py

```
from __future__ import print_function

import boto3
import json

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
        - tableName: required for operations that interact with DynamoDB
        - payload: a parameter to pass to the operation being performed
    '''
    #print("Received event: " + json.dumps(event, indent=2))

    operation = event['operation']

    if 'tableName' in event:
        dynamo = boto3.resource('dynamodb').Table(event['tableName'])

    operations = {
        'create': lambda x: dynamo.put_item(**x),
        'read': lambda x: dynamo.get_item(**x),
        'update': lambda x: dynamo.update_item(**x),
        'delete': lambda x: dynamo.delete_item(**x),
        'list': lambda x: dynamo.scan(**x),
        'echo': lambda x: x,
        'ping': lambda x: 'pong'
    }

    if operation in operations:
        return operations[operation](event.get('payload'))
    else:
        raise ValueError('Unrecognized operation "{}".format(operation)')
```

Zip up the sample code to create a deployment package. For instructions, see [Deploy Python Lambda functions with .zip file archives \(p. 514\)](#).

Go

The following example processes messages from API Gateway, and logs information about the request.

Example LambdaFunctionOverHttps.go

```
import (
    "context"
    "fmt"
    "github.com/aws/aws-lambda-go/events"
    runtime "github.com/aws/aws-lambda-go/lambda"
)

func main() {
```

```
    runtime.Start(handleRequest)
}

func handleRequest(ctx context.Context, request events.APIGatewayProxyRequest)
(events.APIGatewayProxyResponse, error) {
    fmt.Printf("Processing request data for request %s.\n",
request.RequestContext.RequestID)
    fmt.Printf("Body size = %d.\n", len(request.Body))

    fmt.Println("Headers:")
    for key, value := range request.Headers {
        fmt.Printf("    %s: %s\n", key, value)
    }

    return events.APIGatewayProxyResponse{Body: request.Body, StatusCode: 200}, nil
}
```

Build the executable with `go build` and create a deployment package. For instructions, see [Deploy Go Lambda functions with .zip file archives \(p. 616\)](#).

Create a simple microservice using Lambda and API Gateway

In this tutorial you will use the Lambda console to create a Lambda function, and an Amazon API Gateway endpoint to trigger that function. You will be able to call the endpoint with any method (GET, POST, PATCH, etc.) to trigger your Lambda function. When the endpoint is called, the entire request will be passed through to your Lambda function. Your function action will depend on the method you call your endpoint with:

- DELETE: delete an item from a DynamoDB table
- GET: scan table and return all items
- POST: Create an item
- PUT: Update an item

Create an API using Amazon API Gateway

Follow the steps in this section to create a new Lambda function and an API Gateway endpoint to trigger it:

To create an API

1. Sign in to the AWS Management Console and open the AWS Lambda console.
2. Choose **Create Lambda function**.
3. Choose **Blueprint**.
4. Enter **microservice** in the search bar. Choose the **microservice-http-endpoint** blueprint and then choose **Configure**.
5. Configure the following settings.
 - **Name** – `lambda-microservice`.
 - **Role** – **Create a new role from AWS policy templates**.
 - **Role name** – `lambda-apigateway-role`.
 - **Policy templates** – **Simple microservice permissions**.
 - **API** – **Create a new API**.
 - **Security** – **Open**.

Choose **Create function**.

When you complete the wizard and create your function, Lambda creates a proxy resource named `lambda-microservice` under the API name you selected. For more information about proxy resources, see [Configure proxy integration for a proxy resource](#).

A proxy resource has an `AWS_PROXY` integration type and a catch-all method `ANY`. The `AWS_PROXY` integration type applies a default mapping template to pass through the entire request to the Lambda function and transforms the output from the Lambda function to HTTP responses. The `ANY` method defines the same integration setup for all the supported methods, including `GET`, `POST`, `PATCH`, `DELETE` and others.

Test sending an HTTP request

In this step, you will use the console to test the Lambda function. In addition, you can run a `curl` command to test the end-to-end experience. That is, [send an HTTP request](#) to your API method and have Amazon API Gateway invoke your Lambda function. In order to complete the steps, make sure you have created a DynamoDB table and named it "MyTable". For more information, see [Create a DynamoDB table with a stream enabled \(p. 327\)](#)

To test the API

1. With your `lambda-microservice` function still open in the console, choose the **Test** tab.
2. Choose **New event**.
3. Choose the **Hello World** template.
4. In **Name**, enter a name for the test event.
5. In the text entry panel, replace the existing text with the following:

```
{  
  "httpMethod": "GET",  
  "queryStringParameters": {  
    "TableName": "MyTable"  
  }  
}
```

6. After entering the text above choose **Create event**.
7. Choose the event that you created and choose **Invoke**.

AWS SAM template for an API Gateway application

Below is a sample AWS SAM template for the Lambda application from the [tutorial \(p. 267\)](#). 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 \(p. 406\)](#).

Topics

- [Logging AWS Lambda API calls with AWS CloudTrail \(p. 285\)](#)
- [Tutorial: Triggering a Lambda function with AWS CloudTrail events \(p. 288\)](#)
- [Sample function code \(p. 292\)](#)

Logging AWS Lambda API calls with AWS CloudTrail

AWS Lambda is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in AWS Lambda. CloudTrail captures API calls for AWS Lambda as events. The calls captured include calls from the AWS Lambda console and code calls to the AWS Lambda API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for AWS 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 AWS Lambda, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, including how to configure and enable it, see the [AWS CloudTrail User Guide](#).

AWS Lambda information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When supported event activity occurs in AWS 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](#).

For an ongoing record of events in your AWS account, including events for AWS 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 Amazon 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:

- [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](#)

AWS Lambda supports logging the following actions as events in CloudTrail log files:

- [AddLayerVersionPermission \(p. 738\)](#)
- [AddPermission \(p. 742\)](#)
- [CreateEventSourceMapping \(p. 753\)](#)
- [CreateFunction \(p. 762\)](#)

(The `ZipFile` parameter is omitted from the CloudTrail logs for `CreateFunction`.)

- [DeleteEventSourceMapping \(p. 778\)](#)
- [DeleteFunction \(p. 783\)](#)
- [GetEventSourceMapping \(p. 802\)](#)
- [GetFunction \(p. 807\)](#)
- [GetFunctionConfiguration \(p. 816\)](#)
- [GetLayerVersionPolicy \(p. 832\)](#)
- [GetPolicy \(p. 834\)](#)
- [ListEventSourceMappings \(p. 852\)](#)

- [ListFunctions \(p. 858\)](#)
- [RemovePermission \(p. 905\)](#)
- [UpdateEventSourceMapping \(p. 918\)](#)
- [UpdateFunctionCode \(p. 926\)](#)

(The `ZipFile` parameter is omitted from the CloudTrail logs for `UpdateFunctionCode`.)

- [UpdateFunctionConfiguration \(p. 935\)](#)

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 root or IAM 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 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 AWS Lambda log files from multiple AWS regions and multiple AWS accounts into a single S3 bucket. For more information, see [Working with CloudTrail log files](#).

Understanding AWS 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.

```
{  
  "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  
    }  
  ]  
}
```

```

    "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"
}
]
}

```

Note

The `eventName` may include date and version information, such as "GetFunction20150331", but it is still referring to the same public API. For more information, see [Services supported by CloudTrail event history](#) in the *AWS CloudTrail User Guide*.

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](#).

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"  
}
```

Tutorial: Triggering a Lambda function with AWS CloudTrail events

You can configure Amazon S3 to publish events to AWS Lambda when AWS CloudTrail stores API call logs. Your Lambda function can read the log object and process the access records logged by CloudTrail.

Use the following instructions to create a Lambda function that notifies you when a specific API call is made in your account. The function processes notification events from Amazon S3, reads logs from a bucket, and publishes alerts through an Amazon SNS topic. For this tutorial, you create:

- A CloudTrail trail and an S3 bucket to save logs to.
- An Amazon SNS topic to publish alert notifications.
- An IAM user role with permissions to read items from an S3 bucket and write logs to Amazon CloudWatch.
- A Lambda function that processes CloudTrail logs and sends a notification whenever an Amazon SNS topic is created.

Requirements

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 [Getting started with Lambda \(p. 9\)](#) to create your first Lambda function.

Before you begin, make sure that you have the following tools:

- [Node.js 12.x with npm](#).
- The Bash shell. For Linux and macOS, this is included by default. In Windows 10, you can install the [Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.
- [The AWS CLI](#).

Step 1: Creating a trail in CloudTrail

When you create a trail, CloudTrail records the API calls in log files and stores them in Amazon S3. A CloudTrail log is an unordered array of events in JSON format. For each call to a supported API action, CloudTrail records information about the request and the entity that made it. Log events include the action name, parameters, response values, and details about the requester.

To create a trail

1. Open the [Trails page of the CloudTrail console](#).
2. Choose **Create trail**.
3. For **Trail name**, enter a name.
4. For **S3 bucket**, enter a name.
5. Choose **Create**.
6. Save the bucket Amazon Resource Name (ARN) to add it to the IAM execution role, which you create later.

Step 2: Creating an Amazon SNS topic

Create an Amazon SNS topic to send out a notification when new object events have occurred.

To create a topic

1. Open the [Topics page of the Amazon SNS console](#).
2. Choose **Create topic**.
3. For **Topic name**, enter a name.
4. Choose **Create topic**.
5. Record the topic ARN. You will need it when you create the IAM execution role and Lambda function.

Step 3: Creating an IAM execution role

An [execution role \(p. 47\)](#) gives your function permission to access AWS resources. Create an execution role that grants the function permission to access CloudWatch Logs, Amazon S3, and Amazon SNS.

To create an execution role

1. Open the [Roles page](#) of the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties:
 - For **Trusted entity**, choose **Lambda**.
 - For **Role name**, enter **lambda-cloudtrail-role**.
 - For **Permissions**, create a custom policy with the following statements. Replace the highlighted values with the names of your bucket and topic.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "logs:*"  
            ],  
            "Resource": "arn:aws:logs:*:*::*"  
        },  
        {  
            "Effect": "Allow",  
            "Action": [  
                "s3:GetObject"  
            ],  
            "Resource": "arn:aws:s3:::my-bucket/*"  
        }  
    ]  
}
```

```
        },
        {
            "Effect": "Allow",
            "Action": [
                "sns:Publish"
            ],
            "Resource": "arn:aws:sns:us-west-2:123456789012:my-topic"
        }
    ]
}
```

4. Record the role ARN. You will need it when you create the Lambda function.

Step 4: Creating the Lambda function

The following Lambda function processes CloudTrail logs, and sends a notification through Amazon SNS when a new Amazon SNS topic is created.

To create the function

1. Create a folder and give it a name that indicates that it's your Lambda function (for example, *lambda-cloudtrail*).
2. In the folder, create a file named `index.js`.
3. Paste the following code into `index.js`. Replace the Amazon SNS topic ARN with the ARN that Amazon S3 created when you created the Amazon SNS topic.

```
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-east-2';
var SNS_TOPIC_ARN = 'arn:aws:sns:us-west-2:123456789012:my-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);
            next();
        }
    ],
    function(err, result) {
        if (err) {
            return callback(err);
        }
        else {
            return callback(null, result);
        }
    }
});
```

```

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.');
    }
    callback(null,"message");
});
};

```

4. In the `Lambda-cloudtrail` folder, run the following script. It creates a `package-lock.json` file and a `node_modules` folder, which handle all dependencies.

```
npm install async
```

5. Run the following script to create a deployment package.

```
zip -r function.zip .
```

6. Create a Lambda function named `CloudTrailEventProcessing` with the `create-function` command by running the following script. Make the indicated replacements.

```
aws lambda create-function --function-name CloudTrailEventProcessing \
--zip-file file://function.zip --handler index.handler --runtime nodejs12.x --timeout
10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/lambda-cloudtrail-role
```

Step 5: Adding permissions to the Lambda function policy

The Lambda function's resource policy needs permissions to allow Amazon S3 to invoke the function.

To give Amazon S3 permissions to invoke the function

1. Run the following add-permission command. Replace the ARN and account ID with your own.

```
aws lambda add-permission --function-name CloudTrailEventProcessing \
--statement-id Id-1 --action "lambda:InvokeFunction" --principal s3.amazonaws.com \
--source-arn arn:aws:s3:::my-bucket \
--source-account 123456789012
```

This command grants the Amazon S3 service principal (`s3.amazonaws.com`) permissions to perform the `lambda:InvokeFunction` action. Invoke permissions are granted to Amazon S3 only if the following conditions are met:

- CloudTrail stores a log object in the specified bucket.
 - The bucket is owned by the specified AWS account. If the bucket owner deletes a bucket, another AWS account can create a bucket with the same name. This condition ensures that only the specified AWS account can invoke your Lambda function.
2. To view the Lambda function's access policy, run the following `get-policy` command, and replace the function name.

```
aws lambda get-policy --function-name function-name
```

Step 6: Configuring notifications on an Amazon S3 bucket

To request that Amazon S3 publishes object-created events to Lambda, add a notification configuration to the S3 bucket. In the configuration, you specify the following:

- Event type – Any event types that create objects.
- Lambda function – The Lambda function that you want Amazon S3 to invoke.

To configure notifications

1. Open the [Amazon S3 console](#).
2. Choose the source bucket.
3. Choose **Properties**.
4. Under **Events**, configure a notification with the following settings:
 - **Name** – `lambda-trigger`
 - **Events** – `All object create events`
 - **Send to** – `Lambda function`
 - **Lambda** – `CloudTrailEventProcessing`

When CloudTrail stores logs in the bucket, Amazon S3 sends an event to the function. The event provides information, including the bucket name and key name of the log object that CloudTrail created.

Sample function code

Sample code is available for the following languages.

Topics

- [Node.js \(p. 293\)](#)

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.');  
  }  
  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 \(p. 488\)](#).

Using AWS Lambda with Amazon CloudWatch Events

Amazon CloudWatch events help you to respond to state changes in your AWS resources. When your resources change state, they automatically send events into an event stream. 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](#).

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 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 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 CloudWatch Events to invoke your function

1. Open the [Functions page](#) on 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 \(p. 300\)](#).

Each AWS account can have up to 100 unique event sources of the **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 \(p. 296\)](#)
- [AWS SAM template for a CloudWatch Events application \(p. 299\)](#)
- [Schedule expressions using rate or cron \(p. 300\)](#)

Tutorial: Using AWS Lambda with scheduled events

In this tutorial, you do the following:

- Create a Lambda function using the **lambda-canary** blueprint. You configure the Lambda function to run every minute. Note that if the function returns an error, AWS Lambda logs error metrics to 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 [Getting started with Lambda \(p. 9\)](#) 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. Enter **canary** in the search bar. Choose the **lambda-canary** blueprint, and then choose **Configure**.
5. Configure the following settings.

- **Name** – `lambda-canary`.
 - **Role** – **Create a new role from AWS policy templates**.
 - **Role name** – `lambda-apigateway-role`.
 - **Policy templates** – **Simple microservice permissions**.
 - **Rule** – **Create a new rule**.
 - **Rule name** – `CheckWebsiteScheduledEvent`.
 - **Rule description** – `CheckWebsiteScheduledEvent trigger`.
 - **Schedule expression** – `rate(1 minute)`.
 - **Enabled** – True (checked).
 - **Environment variables**
 - `site` – <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>.
 - `expected` – What is AWS Lambda?
6. Choose **Create function**.

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 \(p. 300\)](#).

Test the Lambda function

Test the function with a sample event provided by the Lambda console.

1. Open the [Functions page](#) on the Lambda console.
2. Choose the `lambda-canary` function.
3. Choose **Test**.
4. Create a new event using the **CloudWatch Events** event template.
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. Choose **Create topic**.
3. Create a topic with the following settings.
 - **Name** – `lambda-canary-notifications`.
 - **Display name** – `Canary`.
4. Choose **Create subscription**.
5. Create a subscription with the following settings.
 - **Protocol** – `Email`.
 - **Endpoint** – Your email address.

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. Choose **Alarms**.
3. Choose **Create alarm**.
4. Choose **Alarms**.
5. Create an alarm with the following settings.

- **Metrics – lambda-canary Errors.**

Search for `lambda canary errors` to find the metric.

- **Statistic – Sum.**

Choose the statistic from the drop-down menu above the preview graph.

- **Name – lambda-canary-alarm.**
- **Description – Lambda canary alarm.**
- **Threshold – Whenever Errors is ≥ 1 .**
- **Send notification to – lambda-canary-notifications.**

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](#) on 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. Choose **Delete**.

To delete the CloudWatch alarm

1. Open the [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 box.
5. Choose **Delete**.

AWS SAM template for a CloudWatch Events 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 \(p. 296\)](#). 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
Parameters:
  NotificationEmail:
    Type: String
Resources:
  CheckWebsitePeriodically:
    Type: AWS::Serverless::Function
    Properties:
      Handler: LambdaFunctionOverHttps.handler
      Runtime: runtime
      Policies: AmazonDynamoDBFullAccess
      Events:
        CheckWebsiteScheduledEvent:
          Type: Schedule
          Properties:
            Schedule: rate(1 minute)
```

```

AlarmTopic:
  Type: AWS::SNS::Topic
  Properties:
    Subscription:
      - Protocol: email
        Endpoint: !Ref NotificationEmail

Alarm:
  Type: AWS::CloudWatch::Alarm
  Properties:
    AlarmActions:
      - !Ref AlarmTopic
    ComparisonOperator: GreaterThanOrEqualToThreshold
    Dimensions:
      - Name: FunctionName
        Value: !Ref CheckWebsitePeriodically
    EvaluationPeriods: 1
    MetricName: Errors
    Namespace: AWS/Lambda
    Period: 60
    Statistic: Sum
    Threshold: '1'

```

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*.

Schedule expressions using rate or cron

AWS Lambda supports standard rate and cron expressions for frequencies of up to once per minute. 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). 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>

Frequency	Expression
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":  
      "ewogICAgIm1lc3NhZ2VUeXB1IjogIkRBVEFFTUVTUOFHRSIsCiAgICAib3duZXIIoAiMTIzNDU2Nzg5MDEyIiwKICAgICJsb2dH  
  }  
}
```

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 \(p. 472\)](#).

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 \(p. 140\)](#) with an event that includes a callback URL.

Example – AWS CloudFormation message event

```
{
  "RequestType": "Create",
  "ServiceToken": "arn:aws:lambda:us-east-2:123456789012:function:lambda-error-processor-primer-14ROR2T3JKU66",
  "ResponseURL": "https://cloudformation-custom-resource-response-useast2.s3-us-east-2.amazonaws.com/arn%3Aaws%3Acloudformation%3Aus-east-2%3A123456789012%3Astack/lambda-error-processor/1134083a-2608-1e91-9897-022501a2c456%7Cprimerinvoke%7C5d478078-13e9-baf0-464a-7ef285ecc786?
AWSAccessKeyId=AKIAIOSFODNN7EXAMPLE&Expires=1555451971&Signature=28UijZePE5I4dvukKQqM%2F9Rf1o4%3D",
  "StackId": "arn:aws:cloudformation:us-east-2:123456789012: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-2:123456789012:function:lambda-error-processor-primer-14ROR2T3JKU66",
    "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-2:123456789012: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.

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.

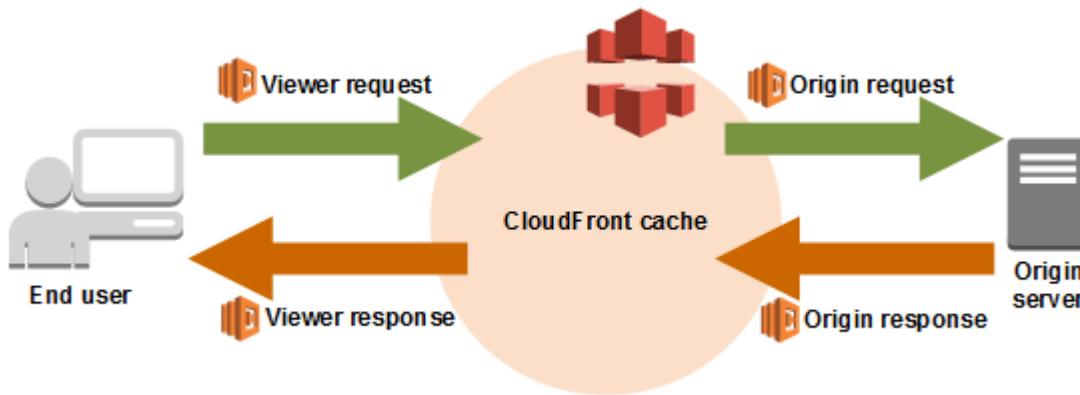
For a sample application that uses a custom resource to ensure that a function's log group is created before another resource that depends on it, see [Error processor sample application for AWS Lambda \(p. 472\)](#).

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/58.0.3029.110 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 \(p. 174\)](#) 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/TdOSFRV"
                        }
                    }
                }
            ],
            "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 \(p. 158\)](#) if your function returns an error. Configure your function with a [destination \(p. 142\)](#) 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 \(p. 858\)](#)
- [InvokeFunction \(p. 839\)](#)

The default [pipeline service role](#) includes these permissions.

To complete a job, the function needs the following permissions in its [execution role \(p. 47\)](#).

- `codepipeline:PutJobSuccessResult`
- `codepipeline:PutJobFailureResult`

These permissions are included in the [AWSCodePipelineCustomActionAccess](#) managed policy.

Using AWS Lambda with Amazon Cognito

The Amazon Cognito Events feature enables you to run Lambda functions in response to events in Amazon Cognito. For example, you can invoke a Lambda function for the Sync Trigger events, 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-00000000\", \"ARN\": \"arn:aws:ec2:us-east-1:000000000000:instance/i-00000000\", \"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::012345678912:role/config-role\",  
    \"configRuleArn\": \"arn:aws:config:us-east-1:012345678912:config-rule/config-rule-0123456\",  
    \"configRuleName\": \"change-triggered-config-rule\",  
    \"configRuleId\": \"config-rule-0123456\",  
    \"accountId\": \"012345678912\",  
    \"version\": \"1.0\"\n}
```

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.

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-XXXXXXXXXXXX",  
            "CustomerEndpoint": {  
                "Address": "+1234567890",  
                "Type": "TELEPHONE_NUMBER"  
            },  
            "InitialContactId": "4a573372-1f28-4e26-b97b-XXXXXXXXXXXX",  
            "InitiationMethod": "INBOUND | OUTBOUND | TRANSFER | CALLBACK",  
            "InstanceARN": "arn:aws:connect:aws-region:1234567890:instance/  
c8c0e68d-2200-4265-82c0-XXXXXXXXXX",  
            "PreviousContactId": "4a573372-1f28-4e26-b97b-XXXXXXXXXXXX",  
            "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 DynamoDB

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 \(p. 138\)](#) with an event that contains stream records. Lambda reads records in batches and invokes your function to process records from the batch.

Example DynamoDB Streams record event

```
{  
    "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": "eventsourcearn",  
            "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"  
                    }  
                }  
            }  
        }  
    ]  
}
```

```
        "Id": {  
            "N": "101"  
        },  
        "StreamViewType": "NEW_AND_OLD_IMAGES"  
    },  
    "awsRegion": "us-west-2",  
    "eventName": "MODIFY",  
    "eventSourceARN": sourcearn,  
    "eventSource": "aws:dynamodb"  
}
```

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 in the stream. If the batch that Lambda reads from the stream only has 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 five minutes by configuring a *batch window*. Before invoking the function, Lambda continues to read records from the stream until it has gathered a full batch, or until the batch window expires.

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 an SQS queue or 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 `ParallelizationFactor` is set 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. For more information, see [New AWS Lambda scaling controls for Kinesis and DynamoDB event sources](#).

Sections

- [Execution role permissions \(p. 314\)](#)
- [Configuring a stream as an event source \(p. 315\)](#)
- [Event source mapping APIs \(p. 316\)](#)
- [Error handling \(p. 318\)](#)
- [Amazon CloudWatch metrics \(p. 319\)](#)
- [Time windows \(p. 319\)](#)
- [Reporting batch item failures \(p. 322\)](#)
- [Tutorial: Using AWS Lambda with Amazon DynamoDB streams \(p. 324\)](#)
- [Sample function code \(p. 329\)](#)
- [AWS SAM template for a DynamoDB application \(p. 332\)](#)

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](#)
- [dynamodb>ListShards](#)

The `AWSLambdaDynamoDBExecutionRole` managed policy includes these permissions. For more information, see [AWS Lambda execution role \(p. 47\)](#).

To send records of failed batches to a queue or 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 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](#) on the Lambda console.
2. Choose 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 \(p. 44\)](#) 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** – An SQS queue or SNS topic for records that can't be processed. When Lambda discards a batch of records because it's too old or has exhausted all retries, it 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.
- **Concurrent batches per shard** – Process multiple batches from the same shard concurrently.
- **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 CLI](#) or [AWS SDK](#), you can use the following API operations:

- [CreateEventSourceMapping \(p. 753\)](#)
- [ListEventSourceMappings \(p. 852\)](#)
- [GetEventSourceMapping \(p. 802\)](#)
- [UpdateEventSourceMapping \(p. 918\)](#)
- [DeleteEventSourceMapping \(p. 778\)](#)

The following example uses the AWS CLI to map a function named `my-function` to a DynamoDB stream that is specified by its Amazon Resource Name (ARN), with a batch size of 500.

```
aws lambda create-event-source-mapping --function-name my-function --batch-size 500 --  
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": 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": 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 an 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 the function is throttled or the Lambda service 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](#) on 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 AWS Lambda function metrics \(p. 691\)](#).

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.

Lambda invocations are stateless—you cannot use them for processing data across multiple continuous invocations without an external database. However, with windowing enabled, 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.

Tumbling windows

Lambda functions can aggregate data using tumbling windows: distinct time windows that open and close at regular intervals. Tumbling windows enable you to process streaming data sources through contiguous, non-overlapping time windows.

Each record of a stream belongs to a specific window. A record is processed only once, when Lambda processes the window that the record belongs to. In each window, you can perform calculations, such as a sum or average, at the [partition key](#) level within a shard.

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. You can also process records and return a new state, which is passed in the next invocation. Lambda returns a `TimeWindowEventResponse` in JSON in the following format:

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!"
          },
          "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": "Original item"
          },
          "Id": {
            "N": "101"
          }
        }
      }
    }
  ]
}
```

```

        "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 \(p. 148\)](#). 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 \(p. 148\)](#).

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>"  
        }  
    ]  
}
```

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 java.io.Serializable;  
import java.util.ArrayList;  
import java.util.List;  
  
public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,  
    Serializable> {  
  
    @Override  
    public Serializable handleRequest(DynamodbEvent input, Context context) {  
  
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new  
        ArrayList<*>();  
        String curRecordSequenceNumber = "";
```

```
        for (DynamodbEvent.DynamodbEventRecord dynamodbEventRecord :  
input.getRecords()) {  
    try {  
        //Process your record  
        DynamodbEvent.Record dynamodbRecord =  
dynamodbEventRecord.getDynamodb();  
        curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();  
  
    } catch (Exception e) {  
        //Return failed record's sequence number  
        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["dynamodb"]["sequenceNumber"]  
        except Exception as e:  
            # Return failed record's sequence number  
            return {"batchItemFailures": [{"itemIdentifier": curRecordSequenceNumber}]}  
  
    return {"batchItemFailures": []}
```

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 [Getting started with Lambda \(p. 9\)](#) to create your first Lambda function.

To complete the following steps, you need a command line terminal or shell to run commands. 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. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the execution role

Create the [execution role \(p. 47\)](#) 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 \(p. 329\)](#).

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": 26  
            }  
        }  
    ]  
}
```

```
        "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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#).

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 **Overview**, choose **Manage stream**.
5. Choose **Enable**.

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. 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 role**.
4. Choose **Yes, 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**.

Sample function code

Sample code is available for the following languages.

Topics

- [Node.js \(p. 329\)](#)
- [Java 11 \(p. 330\)](#)
- [C# \(p. 331\)](#)
- [Python 3 \(p. 331\)](#)
- [Go \(p. 332\)](#)

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 \(p. 488\)](#).

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.DynamodbEvent.DynamodbStreamRecord;

public class DDBEventProcessor implements
    RequestHandler<DynamodbEvent, String> {

    public String handleRequest(DynamodbEvent ddbEvent, Context context) {
        for (DynamodbStreamRecord record : ddbEvent.getRecords()){
            System.out.println(record.getEventID());
            System.out.println(record.getEventName());
            System.out.println(record.getDynamodb().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 \(p. 577\)](#).

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 \(p. 644\)](#).

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 \(p. 514\)](#).

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 \(p. 616\)](#).

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 \(p. 324\)](#). 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 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 CloudWatch Events to forward those events to a Lambda function for processing.

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": "123456798012",  
    "time": "2019-10-02T17:59:30Z",  
    "region": "us-east-2",  
    "resources": [  
        "arn:aws:ec2:us-east-2:123456798012:instance/i-0c314xmplcd5b8173"  
    ],  
    "detail": {  
        "instance-id": "i-0c314xmplcd5b8173",  
        "state": "running"  
    }  
}
```

For details on configuring events in CloudWatch Events, see [Using AWS Lambda with Amazon CloudWatch Events \(p. 295\)](#). For an example function that processes Amazon EBS snapshot notifications, see [Amazon 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. For a tutorial with a sample application in C#, see [Tutorial: Using AWS SDK for .NET to manage Amazon EC2 Spot Instances \(p. 335\)](#).

Permissions

To process lifecycle events from Amazon EC2, CloudWatch Events needs permission to invoke your function. This permission comes from the function's [resource-based policy \(p. 51\)](#). If you use the 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-2:12456789012:function:my-function",  
    "Condition": {  
        "ArnLike": {  
            "AWS:SourceArn": "arn:aws:events:us-east-2:12456789012:rule/*"  
        }  
    }  
}
```

}

To add a statement, use the `add-permission` AWS CLI command.

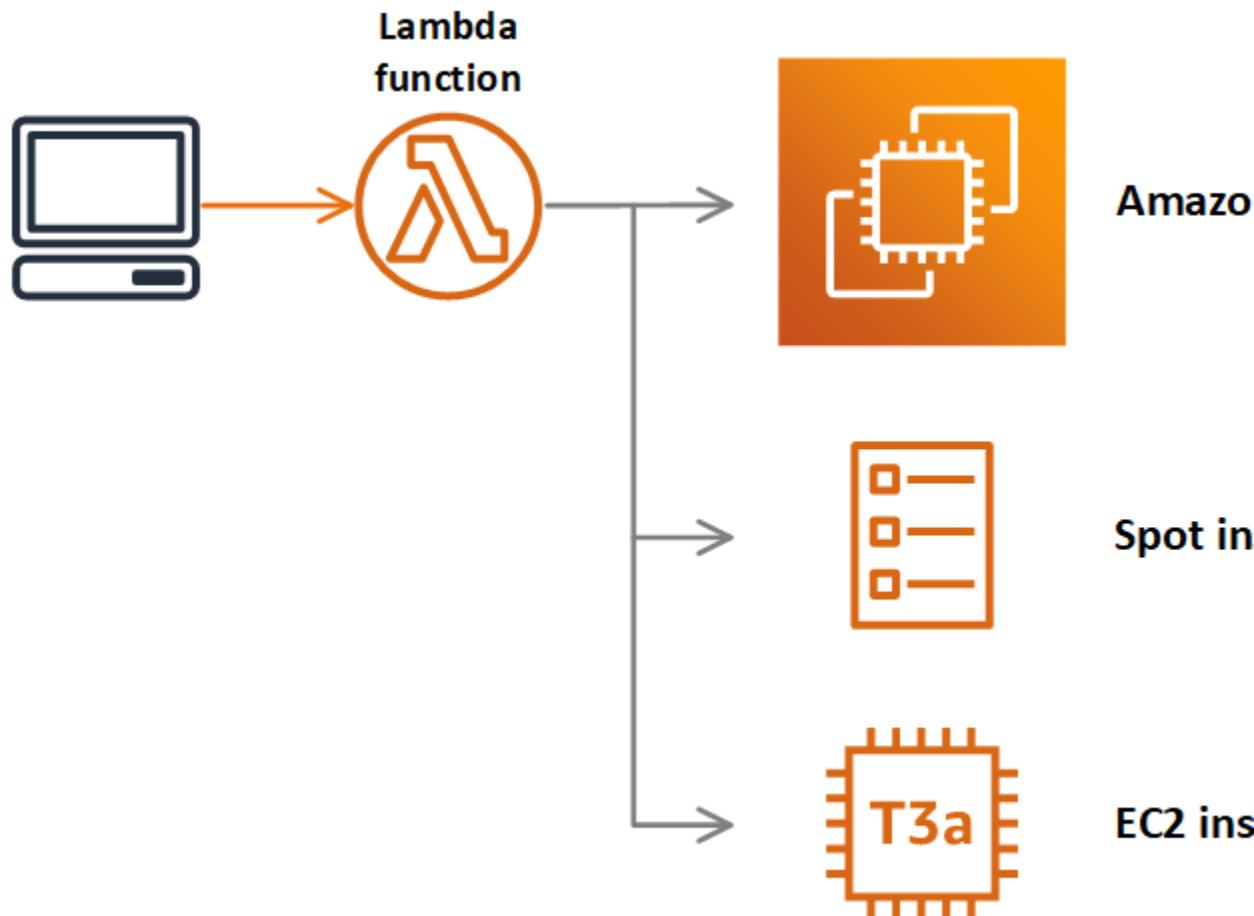
```
aws lambda add-permission --action lambda:InvokeFunction --statement-id ec2-events \
--principal events.amazonaws.com --function-name my-function --source-arn
'arn:aws:events:us-east-2:12456789012:rule/*'
```

If your function uses the AWS SDK to manage Amazon EC2 resources, add Amazon EC2 permissions to the function's [execution role \(p. 47\)](#).

Tutorial: Using AWS SDK for .NET to manage Amazon EC2 Spot Instances

You can use the AWS SDK for .NET to manage Amazon EC2 spot instances with C# code. The SDK enables you to use the Amazon EC2 API to create spot instance requests, determine when the request is fulfilled, delete requests, and identify the instances created.

This tutorial provides code that performs these tasks and a sample application that you can run locally or on AWS. It includes a sample project that you can deploy to AWS Lambda's .NET Core 2.1 runtime.



For more information about spot instances usage and best practices, see [Spot Instances](#) in the Amazon EC2 user guide.

Prerequisites

To complete the following steps, you need a command line terminal or shell to run commands. 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. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

This tutorial uses code from the developer guide's GitHub repository. The repository also contains helper scripts and configuration files that are needed to follow its procedures. Clone the repository at github.com/awsdocs/aws-lambda-developer-guide.

To use the sample code you need the following tools:

- **AWS CLI** – To deploy the sample application to AWS, install the [AWS CLI](#). The AWS CLI also provides credentials to the sample code when you run it locally.
- **.NET Core CLI** – To run and test the code locally, install the [.NET Core SDK 2.1](#).
- **Lambda .NET Core Global Tool** – To build the deployment package for Lambda, install the [.NET Core global tool](#) with the .NET Core CLI.

```
dotnet tool install -g Amazon.Lambda.Tools
```

The code in this tutorial manages spot requests that launch Amazon EC2 instances. To run the code locally, you need SDK credentials with permission to use the following APIs.

- `ec2:RequestSpotInstance`
- `ec2:GetSpotRequestState`
- `ec2:CancelSpotRequest`
- `ec2:TerminateInstances`

To run the sample application in AWS, you need [permission to use Lambda \(p. 46\)](#) and the following services.

- [AWS CloudFormation \(pricing\)](#)
- [Amazon Elastic Compute Cloud \(pricing\)](#)

Standard charges apply for each service.

Review the code

Locate the sample project in the guide repository under [sample-apps/ec2-spot](#). This directory contains Lambda function code, tests, project files, scripts, and a AWS CloudFormation template.

The `Function` class includes a `FunctionHandler` method that calls other methods to create spot requests, check their status, and clean up. It creates an Amazon EC2 client with the AWS SDK for .NET in a static constructor to allow it to be used throughout the class.

Example `Function.cs – FunctionHandler`

```
using Amazon.EC2;
...
public class Function
{
    private static AmazonEC2Client ec2Client;

    static Function()
    {
        AWSSDKHandler.RegisterXRayForAllServices();
        ec2Client = new AmazonEC2Client();
    }

    public async Task<string> FunctionHandler(Dictionary<string, string> input,
ILambdaContext context)
    {
        // More AMI IDs: amazon-linux-2/release-notes/
        // us-east-2 HVM EBS-Backed 64-bit Amazon Linux 2
        string ami = "ami-09d9edae5eb90d556";
        string sg = "default";
        InstanceType type = InstanceType.T3aNano;
        string price = "0.003";
        int count = 1;
        var requestSpotInstances = await RequestSpotInstance(ami, sg, type, price,
count);
        var spotRequestId =
requestSpotInstances.SpotInstanceRequests[0].SpotInstanceRequestId;
    }
}
```

The `RequestSpotInstance` method creates a spot instance request.

Example `Function.cs – RequestSpotInstance`

```
using Amazon;
using Amazon.Util;
using Amazon.EC2;
using Amazon.EC2.Model;
...
public async Task<RequestSpotInstancesResponse> RequestSpotInstance(
    string amiId,
    string securityGroupName,
    InstanceType instanceType,
    string spotPrice,
    int instanceCount)
{
    var request = new RequestSpotInstancesRequest();

    var launchSpecification = new LaunchSpecification();
    launchSpecification.ImageId = amiId;
    launchSpecification.InstanceType = instanceType;
    launchSpecification.SecurityGroups.Add(securityGroupName);

    request.SpotPrice = spotPrice;
    request.InstanceCount = instanceCount;
    request.LaunchSpecification = launchSpecification;

    RequestSpotInstancesResponse response = await
ec2Client.RequestSpotInstancesAsync(request);

    return response;
}
```

```
    }
    ...
}
```

Next, you need to wait until the spot request reaches the `Active` state before proceeding to the last step. To determine the state of your spot request, use the [DescribeSpotInstanceRequests](#) method to obtain the state of the spot request ID to monitor.

```
public async Task<SpotInstanceRequest> GetSpotRequest(string spotRequestId)
{
    var request = new DescribeSpotInstanceRequestsRequest();
    request.SpotInstanceRequestIds.Add(spotRequestId);

    var describeResponse = await ec2Client.DescribeSpotInstanceRequestsAsync(request);

    return describeResponse.SpotInstanceRequests[0];
}
```

The final step is to clean up your requests and instances. It is important to both cancel any outstanding requests and terminate any instances. Just canceling your requests will not terminate your instances, which means that you will continue to be charged for them. If you terminate your instances, your Spot requests may be canceled, but there are some scenarios, such as if you use persistent requests, where terminating your instances is not sufficient to stop your request from being re-fulfilled. Therefore, it is a best practice to both cancel any active requests and terminate any running instances.

You use the [CancelSpotInstanceRequests](#) method to cancel a Spot request. The following example demonstrates how to cancel a Spot request.

```
public async Task CancelSpotRequest(string spotRequestId)
{
    Console.WriteLine("Canceling request " + spotRequestId);
    var cancelRequest = new CancelSpotInstanceRequestsRequest();
    cancelRequest.SpotInstanceRequestIds.Add(spotRequestId);

    await ec2Client.CancelSpotInstanceRequestsAsync(cancelRequest);
}
```

You use the [TerminateInstances](#) method to terminate an instance.

```
public async Task TerminateSpotInstance(string instanceId)
{
    Console.WriteLine("Terminating instance " + instanceId);
    var terminateRequest = new TerminateInstancesRequest();
    terminateRequest.InstanceIds = new List<string>() { instanceId };
    try
    {
        var terminateResponse = await ec2Client.TerminateInstancesAsync(terminateRequest);
    }
    catch (AmazonEC2Exception ex)
    {
        // Check the ErrorCode to see if the instance does not exist.
        if ("InvalidInstanceID.NotFound" == ex.ErrorCode)
        {
            Console.WriteLine("Instance {0} does not exist.", instanceId);
        }
        else
        {
            // The exception was thrown for another reason, so re-throw the exception.
            throw;
        }
    }
}
```

Run the code locally

Run the code on your local machine to create a spot instance request. After the request is fulfilled, the code deletes the request and terminates the instance.

To run the application code

1. Navigate to the `ec2Spot.Tests` directory.

```
cd test/ec2Spot.Tests
```

2. Use the .NET CLI to run the project's unit tests.

```
dotnet test
```

You should see the following output:

```
Starting test execution, please wait...
sir-x5tgs5ij
open
open
open
open
open
active
Canceling request sir-x5tgs5ij
Terminating instance i-0b3fdff0e12e0897e
Complete

Test Run Successful.
Total tests: 1
    Passed: 1
    Total time: 7.6060 Seconds
```

The unit test invokes the `FunctionHandler` method to create a spot instance request, monitor it, and clean up. It is implemented in the [xUnit.net](#) testing framework.

Deploy the application

Run the code in Lambda as a starting point for creating a serverless application.

To deploy and test the application

1. Set your region to `us-east-2`.

```
export AWS_DEFAULT_REGION=us-east-2
```

2. Create a bucket for deployment artifacts.

```
./create-bucket.sh
```

You should see the following output:

```
make_bucket: lambda-artifacts-63d5cbbf18fa5ecc
```

3. Create a deployment package and deploy the application.

```
./deploy.sh
```

You should see the following output:

```
Amazon Lambda Tools for .NET Core applications (3.3.0)
Project Home: https://github.com/aws/aws-extensions-for-dotnet-cli, https://github.com/
aws/aws-lambda-dotnet

Executing publish command
...
Created publish archive (ec2spot.zip)
Lambda project successfully packaged: ec2spot.zip
Uploading to ebd38e401cedd7d676d05d22b76f0209 1305107 / 1305107.0 (100.00%)
Successfully packaged artifacts and wrote output template to file out.yaml.
Run the following command to deploy the packaged template
aws cloudformation deploy --template-file out.yaml --stack-name <YOUR STACK NAME>

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - ec2-spot
```

4. Open the [Applications](#) page of the Lambda console.

The screenshot shows the AWS Lambda console for the 'ec2-spot' function. At the top, there are three tabs: 'Overview' (highlighted in orange), 'Deployments', and 'Monitoring'. Below the tabs, a section titled 'Getting started' is shown. Under 'Resources (2)', there is a search bar with the placeholder 'Filter by attributes or search by keyword'. A table lists two resources: 'function' (Logical ID: ec2-spot-function-17Z36VYL3PS14, Physical ID: ec2-spot-function-17Z36VYL3PS14, Type: Lambda Function) and 'role' (Logical ID: ec2-spot-role-1TDCWJ2ZNNF1M, Physical ID: ec2-spot-role-1TDCWJ2ZNNF1M, Type: IAM Role). A 'Edit permissions' button is located in the top right corner of the resources section.

Logical ID ▲	Physical ID	Type
function	ec2-spot-function-17Z36VYL3PS14	Lambda Function
role	ec2-spot-role-1TDCWJ2ZNNF1M	IAM Role

5. Under **Resources**, choose **function**.
6. Choose **Test** and create a test event from the default template.
7. Choose **Test** again to invoke the function.

View the logs and trace information to see the spot request ID and sequence of calls to Amazon EC2.

To view the service map, open the [Service map page](#) in the X-Ray console.



Choose a node in the service map and then choose **View traces** to see a list of traces. Choose a trace from the list to see the timeline of calls that the function made to Amazon EC2.

Timeline	Raw data				
Method		Response	Duration	Age	ID
--		200	7.6 sec	58.4 sec (2019-08-28 23:51:17 UTC)	1-5
Name	Res.	Duration	Status	0.0ms	1.0s
▼ ec2-spot-function-17Z36VYL3PS14 AWS::Lambda					
ec2-spot-function-17Z36VYL3PS14	200	7.6 sec	✓	---	
▼ ec2-spot-function-17Z36VYL3PS14 AWS::Lambda::Function					
ec2-spot-function-17Z36VYL3PS14	-	7.0 sec	✓		
Initialization	-	358 ms	✓		
EC2	200	3.5 sec	✓		
EC2	200	140 ms	✓		
EC2	200	50.9 ms	✓		
EC2	200	55.3 ms	✓		
EC2	200	49.9 ms	✓		
EC2	200	47.5 ms	✓		
EC2	200	50.3 ms	✓		
EC2	200	52.2 ms	✓		
EC2	200	44.8 ms	✓		
EC2	200	50.6 ms	✓		
EC2	200	80.4 ms	✓		
EC2	200	51.4 ms	✓		
EC2	200	197 ms	✓		
EC2	200	188 ms	✓		

Clean up

The code provided in this tutorial is designed to create and delete spot instance requests, and to terminate the instances that they launch. However, if an error occurs, the requests and instances might not be cleaned up automatically. View the spot requests and instances in the Amazon EC2 console.

To confirm that Amazon EC2 resources are cleaned up

1. Open the [Spot Requests page](#) in the Amazon EC2 console.
2. Verify that the state of the requests is **Cancelled**.
3. Choose the instance ID in the **Capacity** column to view the instance.
4. Verify that the state of the instances is **Terminated** or **Shutting down**.

To clean up the sample function and support resources, delete its AWS CloudFormation stack and the artifacts bucket that you created.

```
./cleanup.sh
```

You should see the following output:

```
Delete deployment artifacts and bucket (lambda-artifacts-63d5cbbf18fa5ecc)?y
delete: s3://lambda-artifacts-63d5cbbf18fa5ecc/ebd38e401cedd7d676d05d22b76f0209
remove_bucket: lambda-artifacts-63d5cbbf18fa5ecc
```

The function's log group is not deleted automatically. You can delete it in the [CloudWatch Logs console](#). Traces in X-Ray expire after a few weeks and are deleted automatically.

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 \(p. 111\)](#).

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 [Getting started with Lambda \(p. 9\)](#) to create your first Lambda function.

To complete the following steps, you need a command line terminal or shell to run commands. 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. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the execution role

Create the [execution role \(p. 47\)](#) 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-elasticahe-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.
    uid_inserted = uuid.uuid4().hex
    #Put the UUID to the cache.
    memcache_client.set('uuid', uid_inserted)
    #Get item (UUID) from the cache.
    uid_obtained = memcache_client.get('uuid')
    if uid_obtained.decode("utf-8") == uid_inserted:
        # this print should go to the CloudWatch Logs and Lambda console.
        print ("Success: Fetched value %s from memcache" %(uid_inserted))
    else:
        raise Exception("Value is not the same as we put :(. Expected %s got %s"
        %(uid_inserted, uid_obtained))

    return "Fetched value from memcache: " + uid_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 \(p. 514\)](#).

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 \(p. 257\)](#).

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. 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 role**.
4. Choose **Yes, 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-2:123456789012: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-2.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\"}"  
}
```

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 \(p. 126\)](#).

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 \(p. 351\)](#)
- [Throughput \(p. 352\)](#)
- [IOPS \(p. 352\)](#)

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 \(p. 84\)](#).

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 \(p. 187\)](#). 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 \(p. 153\)](#) 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 \(p. 84\)](#), 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 \(p. 140\)](#) 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 \(p. 158\)](#) if your function returns an error. Configure your function with a [destination \(p. 142\)](#) 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-west-2:123456789012: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-west-2:123456789012: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

Lambda supports [Apache Kafka](#) as an [event source \(p. 148\)](#). 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 \(p. 395\)](#).

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 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.

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": "aws:kafka",
  "eventSourceArn": "arn:aws:kafka:sa-east-1:123456789012:cluster/vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
  "records": {
    "mytopic-0": [
      {
        "topic": "mytopic",
        "partition": "0",
        "offset": 15,
        "timestamp": 1545084650987,
        "timestampType": "CREATE_TIME",
        "value": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
      }
    ]
  }
}
```

Topics

- [Managing access and permissions \(p. 357\)](#)
- [Kafka authentication \(p. 358\)](#)
- [Network configuration \(p. 359\)](#)
- [Adding a Kafka cluster as an event source \(p. 359\)](#)
- [Using a Kafka cluster as an event source \(p. 361\)](#)
- [Auto scaling of the Kafka event source \(p. 361\)](#)
- [Event source API operations \(p. 361\)](#)
- [Event source mapping errors \(p. 362\)](#)

- Event source configuration parameters ([p. 362](#))

Managing access and permissions

For Lambda to poll your Apache Kafka topic and update other cluster resources, your Lambda function—as well as your AWS Identity and Access Management (IAM) users and roles—must have the following permissions.

Required Lambda function permissions

To create and store logs to a log group in Amazon CloudWatch Logs, your Lambda function must have the following permissions in its [execution role \(p. 47\)](#):

- [logs>CreateLogGroup](#)
- [logs>CreateLogStream](#)
- [logsPutLogEvents](#)

Optional Lambda function permissions

Your Lambda function might need permission to describe your AWS Secrets Manager secret or your AWS Key Management Service (AWS KMS) [customer managed customer master key \(CMK\)](#), or to access your virtual private cloud (VPC).

Secrets Manager and AWS KMS permissions

If your Apache Kafka users access your Kafka brokers over the internet, you must specify a Secrets Manager secret. Your Lambda function might need permission to describe your Secrets Manager secret or to decrypt your AWS KMS customer managed CMK. To access these resources, your function's [execution role \(p. 47\)](#) must have the following permissions:

- [secretsmanagerGetSecretValue](#)
- [kmsDecrypt](#)

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 Virtual Private Cloud (Amazon VPC) resources. These resources include your VPC, subnets, security groups, and network interfaces. To access these resources, your function's [execution role \(p. 47\)](#) must have the following permissions:

- [ec2CreateNetworkInterface](#)
- [ec2DescribeNetworkInterfaces](#)
- [ec2DescribeVpcs](#)
- [ec2DeleteNetworkInterface](#)
- [ec2DescribeSubnets](#)
- [ec2DescribeSecurityGroups](#)

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 \(p. 47\)](#).

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": "arn:aws:ec2:us-east-1:01234567890:instance/my-instance-name"  
        }  
    ]  
}
```

For information about creating a JSON policy document in the IAM console, see [Creating policies on the JSON tab](#) in the *IAM User Guide*.

Adding users to an IAM policy

By default, IAM users and roles do not have permission to perform [event source API operations \(p. 361\)](#). To grant access to users in your organization or account, you might need to create an identity-based policy. For more information, see [Controlling access to AWS resources using policies](#) in the *IAM User Guide*.

Kafka 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 the following authentication methods that Lambda supports:

- VPC

If only Kafka users within your VPC access your Kafka brokers, you must configure the event source with VPC access.

- SASL/SCRAM

With Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) authentication, the cluster encrypts the credentials using SSL encryption. Lambda sends the encrypted credentials to authenticate with the cluster. Because the credentials are encrypted, the connection to the cluster does not need to be encrypted. For more information about SASL/SCRAM authentication, see [RFC 5802](#).

- SASL/PLAIN

With SASL/PLAIN authentication, credentials are sent as clear text (unencrypted) to the server. Because the credentials are clear text, the connection to the server must use TLS encryption.

For SASL authentication, you must store the user name and password as a secret in Secrets Manager. For more information, see [Tutorial: Creating and retrieving a secret](#) in the *AWS Secrets Manager User Guide*.

Network configuration

If you configure Amazon VPC access to your Kafka brokers, Lambda must have access to the Amazon VPC resources.

Lambda must have access to the Amazon Virtual Private Cloud (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 authentication is required, 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 \(p. 116\)](#).

You must configure your Amazon VPC security groups with the following rules (at minimum):

- Inbound rules – Allow all traffic on all ports for the security group specified as your event source.
- Outbound rules – Allow all traffic on all ports for all destinations.

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 \(p. 148\)](#), add your Kafka cluster as a Lambda function [trigger \(p. 29\)](#) 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.

Prerequisites

- A self-managed Apache Kafka cluster. Lambda supports Apache Kafka version 0.10.0.0 and later.
- A Lambda execution role with permission to access the AWS resources that your self-managed Kafka cluster uses.

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. (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.
5. Under **Authentication method**, choose the access or authentication protocol of the Kafka brokers in your cluster. If only users within your VPC access your Kafka brokers, you must configure VPC access. If users access your Kafka brokers over the internet, you must configure SASL authentication.
 - To configure VPC access, choose the **VPC** for your Kafka cluster, then choose **VPC subnets** and **VPC security groups**.
 - To configure SASL authentication, under **Secret key**, choose **Add**, then do the following:
 - a. Choose the key type. If your Kafka broker uses SASL plaintext, choose **BASIC_AUTH**. Otherwise, choose one of the **SASL_SCRAM** options.
 - b. Choose the name of the Secrets Manager secret key that contains the credentials for your Kafka cluster.
6. 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, you must 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](#) 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](#) 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-343a-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 \(p. 148\)](#).

Lambda reads event data from the Kafka topics that you specify in [Topics](#) based on the starting position that you specify in [StartingPosition](#). After successful processing, your Kafka topic is committed to your Kafka cluster.

If you specify `LATEST` as the starting position, 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 does not 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 Lambda function. When more records are available, Lambda continues processing records in batches, based on the value that you specify in [BatchSize](#), until the function catches up with the topic.

If your Lambda 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.

The maximum amount of time that Lambda lets a function run before stopping it is 14 minutes.

Auto scaling of the Kafka event source

When you initially create an Apache Kafka [event source \(p. 148\)](#), Lambda allocates one consumer to process all of the partitions in the Kafka topic. 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.

Every 15 minutes, 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.

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 \(p. 692\)](#) for your function.

Event source API operations

When you add your Kafka cluster as an [event source \(p. 148\)](#) 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 CLI](#) or [AWS SDK](#), you can use the following API operations:

- [CreateEventSourceMapping \(p. 753\)](#)
- [ListEventSourceMappings \(p. 852\)](#)
- [GetEventSourceMapping \(p. 802\)](#)
- [UpdateEventSourceMapping \(p. 918\)](#)
- [DeleteEventSourceMapping \(p. 778\)](#)

Event source mapping errors

When you add your Apache Kafka cluster as an [event source \(p. 148\)](#) 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 is not valid.

EVENT_SOURCE_AUTHN_ERROR

Lambda could not 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 configuration is not valid.

Note

If your Lambda event records exceed the allowed size limit of 6 MB, they can go unprocessed.

Event source configuration parameters

All Lambda event source types share the same [CreateEventSourceMapping \(p. 753\)](#) and [UpdateEventSourceMapping \(p. 918\)](#) 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
DestinationConfig	N	none	
Enabled	N	Enabled	
KafkaBrokerList	Y		Can set only on Create
FunctionName	Y		

Parameter	Required	Default	Notes
MaximumBatchingWindowInSeconds	N	0	Maximum: 300
MaximumRecordAgeInSeconds	N	604,800 (7 days)	Maximum: no limit
MaximumRetryAttempts	N	10,000	Maximum: no limit
ParallelizationFactor	N	1	
SourceAccessConfiguration	N	No credentials	VPC information or authentication credentials for the cluster For SASL_PLAIN, set to BASIC_AUTH
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

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](#).

Lambda reads records from the data stream and invokes your function [synchronously \(p. 138\)](#) with an event that contains stream records. Lambda reads records in batches and invokes your function to process records from the batch.

Example Kinesis record event

```
{  
    "Records": [  
        {  
            "kinesis": {  
                "kinesisSchemaVersion": "1.0",  
                "partitionKey": "1",  
                "sequenceNumber":  
"49590338271490256608559692538361571095921575989136588898",  
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",  
                "approximateArrivalTimestamp": 1545084650.987  
            },  
            "eventSource": "aws:kinesis",  
            "eventVersion": "1.0",  
            "eventID":  
"shardId-000000000006:49590338271490256608559692538361571095921575989136588898",  
            "eventName": "aws:kinesis:record",  
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",  
            "awsRegion": "us-east-2",  
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"  
        },  
        {  
            "kinesis": {  
                "kinesisSchemaVersion": "1.0",  
                "partitionKey": "1",  
                "sequenceNumber":  
"49590338271490256608559692540925702759324208523137515618",  
                "data": "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"
    ]
}
```

By default, Lambda invokes your function as soon as records are available in the stream. If the batch that Lambda reads from the stream only has 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 five minutes by configuring a *batch window*. Before invoking the function, Lambda continues to read records from the stream until it has gathered a full batch, or until the batch window expires.

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 an SQS queue or 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 `ParallelizationFactor` is set 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. For more information, see [New AWS Lambda scaling controls for Kinesis and DynamoDB event sources](#).

Sections

- [Configuring your data stream and function \(p. 366\)](#)
- [Execution role permissions \(p. 367\)](#)
- [Configuring a stream as an event source \(p. 367\)](#)
- [Event source mapping API \(p. 368\)](#)
- [Error handling \(p. 370\)](#)
- [Amazon CloudWatch metrics \(p. 371\)](#)
- [Time windows \(p. 372\)](#)
- [Reporting batch item failures \(p. 374\)](#)
- [Tutorial: Using AWS Lambda with Amazon Kinesis \(p. 376\)](#)
- [Sample function code \(p. 380\)](#)
- [AWS SAM template for a Kinesis application \(p. 383\)](#)

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 \(p. 44\)](#) or [reserve concurrency \(p. 84\)](#) 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 \(p. 47\)](#).

- [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 [AWS Lambda execution role \(p. 47\)](#).

To send records of failed batches to a queue or 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 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.

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](#) on the Lambda console.
2. Choose 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 \(p. 44\)](#) 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** – An SQS queue or SNS topic for records that can't be processed. When Lambda discards a batch of records because it's too old or has exhausted all retries, it 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.
- **Concurrent batches per shard** – Process multiple batches from the same shard concurrently.
- **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.

Event source mapping API

To manage an event source with the [AWS CLI](#) or [AWS SDK](#), you can use the following API operations:

- [CreateEventSourceMapping \(p. 753\)](#)

- [ListEventSourceMappings \(p. 852\)](#)
- [GetEventSourceMapping \(p. 802\)](#)
- [UpdateEventSourceMapping \(p. 918\)](#)
- [DeleteEventSourceMapping \(p. 778\)](#)

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 an 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: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 Kinesis stream invokes your function synchronously and retries on errors. If the function is throttled or the Lambda service 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](#) on 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 `IIteratorAge` 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 AWS Lambda function metrics \(p. 691\)](#).

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.

Lambda invocations are stateless—you cannot use them for processing data across multiple continuous invocations without an external database. However, with windowing enabled, 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.

Tumbling windows

Lambda functions can aggregate data using tumbling windows: distinct time windows that open and close at regular intervals. Tumbling windows enable you to process streaming data sources through contiguous, non-overlapping time windows.

Each record of a stream belongs to a specific window. A record is processed only once, when Lambda processes the window that the record belongs to. In each window, you can perform calculations, such as a sum or average, at the [partition key](#) level within a shard.

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. You can also process records and return a new state, which is passed in the next invocation. Lambda returns a `TimeWindowEventResponse` in JSON in the following format:

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 \(p. 148\)](#). 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 \(p. 148\)](#).

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>"
    }
  ]
}
```

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 java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
Serializable> {

    @Override
    public Serializable 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) {
                //Return failed record's sequence number
                batchItemFailures.add(new
StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

        return new StreamsEventResponse(batchItemFailures);
    }
}
```

Python

Example Handler.py – return batchItemFailures[]

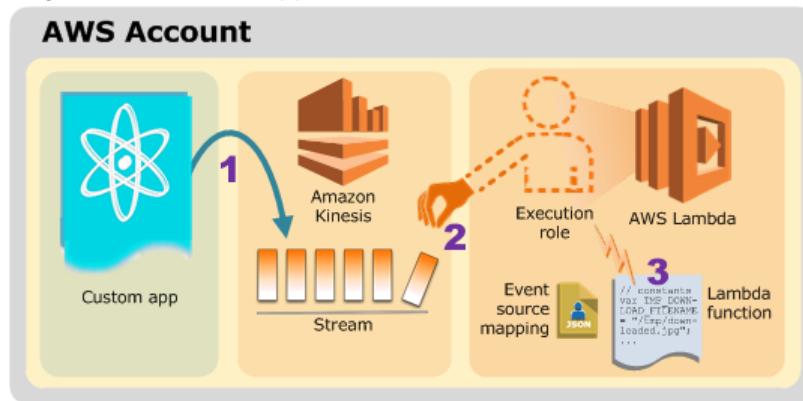
```
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    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": []}
```

Tutorial: Using AWS Lambda with Amazon Kinesis

In this tutorial, you create a Lambda function to consume events from a Kinesis stream. The following diagram illustrates the application flow:



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 [Getting started with Lambda \(p. 9\)](#) to create your first Lambda function.

To complete the following steps, you need a command line terminal or shell to run commands. 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. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the execution role

Create the [execution role \(p. 47\)](#) 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 \(p. 380\)](#).

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 fileb://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": "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-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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#).

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 role**.
4. Choose **Yes, 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. 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 box.
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 \(p. 381\)](#)
- [Java 11 \(p. 381\)](#)
- [C# \(p. 382\)](#)
- [Python 3 \(p. 382\)](#)
- [Go \(p. 383\)](#)

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 \(p. 488\)](#).

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 \(p. 577\)](#).

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 \(p. 644\)](#).

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 \(p. 514\)](#).

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 \(p. 616\)](#).

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 \(p. 376\)](#). 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 \(p. 366\)](#), 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 \(p. 84\)](#) 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 \(p. 138\)](#). The event parameter contains information about the bot and the value of each slot in the dialog. The invocationSource parameter indicates whether the Lambda function should validate the inputs (DialogCodeHook) or fulfill the intent (FulfillmentCodeHook).

Example Amazon Lex message event

```
{  
  "messageVersion": "1.0",  
  "invocationSource": "FulfillmentCodeHook",  
  "userId": "ABCD1234",  
  "sessionAttributes": {  
    "key1": "value1",  
    "key2": "value2",  
  },  
  "bot": {  
    "name": "OrderFlowers",  
    "alias": "prod",  
    "version": "1"  
  },  
  "outputDialogMode": "Text",  
  "currentIntent": {  
    "name": "OrderFlowers",  
    "slots": {  
      "FlowerType": "lilies",  
      "PickupDate": "2030-11-08",  
      "PickupTime": "10:00"  
    },  
    "confirmationStatus": "Confirmed"  
  }  
}
```

Amazon Lex expects a response from a Lambda function in the following format. The dialogAction field is required. The sessionAttributes and the recentIntentSummaryView fields are optional.

Example Amazon Lex message event

```
{  
  "sessionAttributes": {  
    "key1": "value1",  
    "key2": "value2"  
  }
```

```

    ...
},
"recentIntentSummaryView": [
{
  "intentName": "Name",
  "checkpointLabel": "Label",
  "slots": {
    "slot name": "value",
    "slot name": "value"
  },
  "confirmationStatus": "None, Confirmed, or Denied (intent confirmation, if
configured)",
  "dialogActionType": "ElicitIntent, ElicitSlot, ConfirmIntent, Delegate, or Close",
  "fulfillmentState": "Fulfilled or Failed",
  "slotToElicit": "Next slot to elicit"
},
],
"dialogAction": {
  "type": "Close",
  "fulfillmentState": "Fulfilled",
  "message": {
    "contentType": "PlainText",
    "content": "Thanks, your pizza has been ordered."
  },
  "responseCard": {
    "version": integer-value,
    "contentType": "application/vnd.amazonaws.card.generic",
    "genericAttachments": [
      {
        "title": "card-title",
        "subTitle": "card-sub-title",
        "imageUrl": "URL of the image to be shown",
        "attachmentLinkUrl": "URL of the attachment to be associated with the card",
        "buttons": [
          {
            "text": "button-text",
            "value": "Value sent to server on button click"
          }
        ]
      }
    ]
  }
}
}
}

```

Note that the additional fields required for dialogAction vary based on the value of the type field. For more information about the event and response fields, see [Lambda event and response format](#) in the *Amazon Lex Developer Guide*. 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 \(p. 47\)](#). 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

Amazon MQ is a managed message broker service for [Apache ActiveMQ](#) and [RabbitMQ](#). Lambda only supports Apache ActiveMQ. A *message broker* allows 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 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. Your function is triggered through an [event source mapping \(p. 148\)](#), a Lambda resource that reads messages from your broker and invokes the function [synchronously \(p. 138\)](#).

The Amazon MQ event source mapping has the following configuration restrictions:

- Cross account – Cross account processing is not supported. You cannot use Lambda to process records from an Amazon MQ message broker that is in a different account.
- Authentication – Only the ActiveMQ [SimpleAuthenticationPlugin](#) is supported. User credentials associated with the broker are the only method of connection. For more information about authentication, see [Messaging Authentication and Authorization for ActiveMQ](#) 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 interact with the records. For more information, see [the section called “Event source mapping API” \(p. 391\)](#) 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.
- Network topology – Only one single-instance or standby broker is supported per event source mapping. Single-instance brokers require a failover endpoint. For more information about these broker deployment modes, see [Amazon MQ Broker Architecture](#) in the *Amazon MQ Developer Guide*.
- Protocols – Lambda consumes messages using the OpenWire/Java Message Service (JMS) protocol. No other protocols are supported. Within the JMS protocol, only [TextMessage](#) and [BytesMessage](#) are supported. For more information about the OpenWire protocol, see [OpenWire](#) on the Apache ActiveMQ website.

Lambda automatically supports the latest versions of ActiveMQ 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 \(p. 390\)](#)
- [Execution role permissions \(p. 391\)](#)
- [Configuring a broker as an event source \(p. 391\)](#)
- [Event source mapping API \(p. 391\)](#)
- [Event source mapping errors \(p. 393\)](#)

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.

Lambda will pull messages until it has processed a maximum of 6 MB, until timeout, or until the batch size is fulfilled. When configured, batch size determines the maximum number of items to retrieve in a single batch. Your batch is converted into a Lambda payload, and your target function is invoked. Messages are neither persisted nor deserialized. Instead, they are retrieved by the consumer group as a BLOB of bytes and are base64-encoded for a JSON payload.

Note

The maximum function invocation time is 14 minutes.

Lambda processes all incoming batches concurrently and automatically scales the concurrency to meet demands. 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 "Concurrency" \(p. 84\)](#).

Example Amazon MQ record event

```
{  
    "eventSource": "aws:amq",  
    "eventSourceArn": "arn:aws:mq:us-  
west-2:112556298976: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:  
            "messageType": "jms/text-message",  
            "data": "QUJDOkFBQUE=",  
            "connectionId": "myJMSSCoID",  
            "redelivered": false,  
            "destination": {  
                "physicalname": "testQueue"  
            },  
            "timestamp": 1598827811958,  
            "brokerInTime": 1598827811958,  
            "brokerOutTime": 1598827811959  
        },  
        {  
            "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-  
west-2.amazonaws.com-37557-1234520418293-4:1:1:1:  
            "messageType": "jms/bytes-message",  
            "data": "3DTOOW7crj51prgVLQaGQ82S48k=",  
            "connectionId": "myJMSSCoID1",  
            "persistent": false,  
            "destination": {  
                "physicalname": "testQueue"  
            },  
            "timestamp": 1598827811958,  
            "brokerInTime": 1598827811958,  
            "brokerOutTime": 1598827811959  
        }  
    ]  
}
```

Execution role permissions

To read records from an Amazon MQ broker, your Lambda function needs the following permissions added to its [execution role](#) (p. 47):

- [mq:DescribeBroker](#)
- [secretsmanager:GetSecretValue](#)
- [ec2>CreateNetworkInterface](#)
- [ec2>DeleteNetworkInterface](#)
- [ec2:DescribeNetworkInterfaces](#)
- [ec2:DescribeSecurityGroups](#)
- [ec2:DescribeSubnets](#)
- [ec2:DescribeVpcs](#)

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. 148) 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](#) on the Lambda console.
2. Choose 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** – Select the AWS 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 CLI](#) or [AWS SDK](#), you can use the following API operations:

- [CreateEventSourceMapping](#) (p. 753)

- [ListEventSourceMappings \(p. 852\)](#)
- [GetEventSourceMapping \(p. 802\)](#)
- [UpdateEventSourceMapping \(p. 918\)](#)
- [DeleteEventSourceMapping \(p. 778\)](#)

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 is given 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. To meet the 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 \(p. 116\)](#).
- Create a connection between your Amazon VPC and Lambda. Your Amazon VPC must also connect to AWS STS and Secrets Manager endpoints. For more information, see [Configuring interface VPC endpoints for Lambda \(p. 118\)](#).

The Amazon Virtual Private Cloud (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 broker named `ExampleMQBroker`. The command also provides a Secrets Manager secret named `ExampleMQBrokerUserPassword` that stores the broker credentials.

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:mq:us-east-1:12345678901:broker:ExampleMQBroker:b-b4d492ef-
bdc3-45e3-a781-cd1a3102ecca \
--function-name MQ-Example-Function \
--source-access-configuration Type=BASIC_AUTH,URI=arn:aws:secretsmanager:us-
east-1:12345678901:secret:ExampleMQBrokerUserPassword-xPBMT \
--queues ExampleQueue
```

You should see the following output:

```
{
    "UUID": "91eaeb7e-c976-1234-9451-8709db01f137",
    "BatchSize": 100,
    "EventSourceArn": "arn:aws:mq:us-east-1:12345678901:broker:ExampleMQBroker:b-b4d492ef-
bdc3-45e3-a781-cd1a3102ecca",
    "FunctionArn": "arn:aws:lambda:us-east-1:12345678901: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-1:12345678901:secret:ExampleMQBrokerUserPassword-xPBMTt"
        }
    ]
}

```

Using the [update-event-source-mapping](#) command, you can configure additional options such as how batches are processed and to specify when to discard records that can't 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-1:12345678901:broker:ExampleMQBroker:b-b4d492ef-
bdc3-45e3-a781-cd1a3102ecca",
    "FunctionArn": "arn:aws:lambda:us-east-1:12345678901:function:MQ-Example-Function",
    "LastModified": 1601928393.531,
    "LastProcessingResult": "No records processed",
    "State": "Updating",
    "StateTransitionReason": "USER_INITIATED"
}
```

Updated settings are applied asynchronously and aren't reflected in the output until the 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-1:12345678901:broker:ExampleMQBroker:b-b4d492ef-
bdc3-45e3-a781-cd1a3102ecca",
    "FunctionArn": "arn:aws:lambda:us-east-1:12345678901: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 they don't

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 they are dropped due to their size. The size limit for Lambda records is 6 MB. To redeliver messages upon function error, you can use a redelivery policy and dead-letter queue (DLQ) handling with Amazon MQ. For more information, see [Message Redelivery and DLQ Handling](#) on the Apache ActiveMQ website.

Using Lambda with Amazon MSK

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). Additionally, 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.

Lambda reads the messages sequentially for each partition. 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.

Lambda allows a function to run for up to 14 minutes before stopping it.

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",
  "records": {
    "mytopic-0": [
      {
        "topic": "mytopic"
        "partition": "0",
        "offset": 15,
        "timestamp": 1545084650987,
        "timestampType": "CREATE_TIME",
        "value": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
      }
    ]
  }
}
```

Topics

- [Managing access and permissions \(p. 395\)](#)
- [Network configuration \(p. 359\)](#)
- [Adding Amazon MSK as an event source \(p. 397\)](#)
- [Auto scaling of the Amazon MSK event source \(p. 398\)](#)
- [Amazon MSK configuration parameters \(p. 362\)](#)

Managing access and permissions

For Lambda to poll your Kafka topic and update other cluster resources, your Lambda function—as well as your IAM users and roles—must have the following permissions.

Required Lambda function permissions

Your Lambda function's [execution role \(p. 47\)](#) must have permission to read records from your Amazon MSK cluster on your behalf. You can either add the AWS managed policy `AWSLambdaMSKExecutionRole` to your execution role, or create a custom policy with permission to perform the following actions:

- [kafka:DescribeCluster](#)
- [kafka:GetBootstrapBrokers](#)
- [ec2>CreateNetworkInterface](#)
- [ec2:DescribeNetworkInterfaces](#)
- [ec2:DescribeVpcs](#)
- [ec2:DeleteNetworkInterface](#)
- [ec2:DescribeSubnets](#)
- [ec2:DescribeSecurityGroups](#)
- [logs>CreateLogGroup](#)
- [logs>CreateLogStream](#)
- [logs:PutLogEvents](#)

Adding a policy 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, IAM users and roles do not have permission to perform Amazon MSK API operations. To grant access to users in your organization or account, you might need 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. 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.

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). If authentication is required, also deploy a VPC endpoint for Secrets Manager.

Alternatively, ensure that the VPC associated with your Amazon MSK cluster includes one NAT gateway per public subnet. For more information, see [Internet and service access for VPC-connected functions \(p. 116\)](#).

You must configure your Amazon VPC security groups with the following rules (at minimum):

- Inbound rules – Allow all traffic on all ports for the security group specified as your event source.
- Outbound rules – Allow all traffic on all ports for all destinations.

Note

Your Amazon VPC configuration is discoverable through the [Amazon MSK API](#), and does not need to be configured 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 \(p. 148\)](#), add Amazon MSK as a Lambda function [trigger \(p. 29\)](#) 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.

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](#).
- A [Lambda execution role \(p. 47\)](#) with permission to access the AWS resources that your Amazon MSK cluster uses.

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 **Topic name**, enter the name of a Kafka topic.
 - e. (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.

- f. (Optional) For **Secret key**, choose the secret key for SASL/SCRAM authentication of the brokers in your Amazon MSK cluster.
 - g. 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: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](#) 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 of the partitions in the Kafka topic. 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.

Every 15 minutes, 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.

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 [Amazon MSK consumer-lag metrics](#). To help you find the metrics for this Lambda function, the value of the **consumer group** field in the logs is set to the event source UUID.

To check how many function invocations occur in parallel, you can also monitor the [concurrency metrics \(p. 692\)](#) for your function.

Amazon MSK configuration parameters

All Lambda event source types share the same [CreateEventSourceMapping \(p. 753\)](#) and [UpdateEventSourceMapping \(p. 918\)](#) API operations. However, only some of the parameters apply to Amazon MSK.

Event source parameters that apply to Amazon MSK

Parameter	Required	Default	Notes
BatchSize	N	100	Maximum: 10,000
DestinationConfig	N	none	
Enabled	N	Enabled	
EventSourceArn	Y		Can set only on Create
FunctionName	Y		
MaximumBatchingWindowInSeconds	N	0	Maximum: 300
MaximumRecordAgeInSeconds	N	604,800 (7 days)	Maximum: no limit
MaximumRetryAttempts	N	10,000	Maximum: no limit
ParallelizationFactor	N	1	
SourceAccessConfiguration	N	No credentials	VPC information or SASL/SCRAM 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.amazonwebservices.com/AmazonRDS/latest/UserGuide/USER_Events.html#RDS-EVENT-0002\",  
\"Event Message\":\"Finished DB Instance backup\"}",  
                "MessageAttributes": {},  
                "Type": "Notification",  
                "UnsubscribeUrl": "https://sns.us-east-2.amazonaws.com/?  
Action=Unsubscribe&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: Configuring a Lambda function to access Amazon RDS in an Amazon VPC \(p. 400\)](#)
- [Configuring the function \(p. 404\)](#)

Tutorial: Configuring a Lambda function to access Amazon RDS in an Amazon VPC

In this tutorial, you do the following:

- Launch an Amazon RDS MySQL database engine instance in your default Amazon VPC. In the MySQL instance, you create a database (ExampleDB) with a sample table (Employee) in it. For more information about Amazon RDS, see [Amazon RDS](#).
- Create a Lambda function to access the ExampleDB database, create a table (Employee), add a few records, and retrieve the records from the table.
- Invoke the Lambda function and verify the query results. This is how you verify that your Lambda function was able to access the RDS MySQL instance in the VPC.

For details on using Lambda with Amazon VPC, see [Configuring a Lambda function to access resources in a VPC \(p. 111\)](#).

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 [Getting started with Lambda \(p. 9\)](#) to create your first Lambda function.

To complete the following steps, you need a command line terminal or shell to run commands. 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. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the execution role

Create the [execution role \(p. 47\)](#) 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 Amazon RDS database instance

In this tutorial, the example Lambda function creates a table (`Employee`), inserts a few records, and then retrieves the records. The table that the Lambda function creates has the following schema:

```
Employee(EmpID, Name)
```

Where `EmpID` is the primary key. Now, you need to add a few records to this table.

First, you launch an RDS MySQL instance in your default VPC with ExampleDB database. If you already have an RDS MySQL instance running in your default VPC, skip this step.

You can launch an RDS MySQL instance using one of the following methods:

- Follow the instructions at [Creating a MySQL DB instance and connecting to a database on a MySQL DB instance](#) in the *Amazon RDS User Guide*.
- Use the following AWS CLI command:

```
aws rds create-db-instance --db-name ExampleDB --engine MySQL \
--db-instance-identifier MySQLForLambdaTest --backup-retention-period 3 \
--db-instance-class db.t2.micro --allocated-storage 5 --no-publicly-accessible \
--master-username username --master-user-password password
```

Write down the database name, user name, and password. You also need the host address (endpoint) of the DB instance, which you can get from the RDS console. You might need to wait until the instance status is available and the Endpoint value appears in the console.

Create a deployment package

The following example Python code runs a SELECT query against the Employee table in the MySQL RDS instance that you created in the VPC. The code creates a table in the ExampleDB database, adds sample records, and retrieves those records.

The following method for handling database credentials is for illustrative purposes only. In a production environment, we recommend using AWS Secrets Manager instead of environment variables to store database credentials. For more information, see [Configuring database access for a Lambda function](#).

Example app.py

```
import sys
import logging
import rds_config
import pymysql
#rds settings
rds_host = "rds-instance-endpoint"
name = rds_config.db_username
password = rds_config.db_password
db_name = rds_config.db_name

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

try:
    conn = pymysql.connect(host=rds_host, 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 handler(event, context):
    """
    This function fetches content from MySQL RDS instance
    """

    item_count = 0

    with conn.cursor() as cur:
        cur.execute("create table Employee ( EmpID  int NOT NULL, Name varchar(255) NOT
NULL, PRIMARY KEY (EmpID))")
        cur.execute('insert into Employee (EmpID, Name) values(1, "Joe")')
        cur.execute('insert into Employee (EmpID, Name) values(2, "Bob")')
        cur.execute('insert into Employee (EmpID, Name) values(3, "Mary")')
```

```
conn.commit()
cur.execute("select * from Employee")
for row in cur:
    item_count += 1
    logger.info(row)
    #print(row)
conn.commit()

return "Added %d items from RDS MySQL table" %(item_count)
```

Executing `pymysql.connect()` outside of the handler allows your function to re-use the database connection for better performance.

A second file contains connection information for the function.

Example rds_config.py

```
#config file containing credentials for RDS MySQL instance
db_username = "username"
db_password = "password"
db_name = "ExampleDB"
```

A deployment package is a .zip file containing your Lambda function code and dependencies. The sample function code has the following dependencies:

Dependencies

- `pymysql` – The Lambda function code uses this library to access your MySQL instance (see [PyMySQL](#)).

To create a deployment package

- Install dependencies with Pip and create a deployment package. For instructions, see [Deploy Python Lambda functions with .zip file archives \(p. 514\)](#).

Create the Lambda function

Create the Lambda function with the `create-function` command. You can find the subnet IDs and security group ID for your default VPC in the [Amazon VPC console](#).

```
aws lambda create-function --function-name CreateTableAddRecordsAndRead --runtime
python3.8 \
--zip-file fileb://app.zip --handler app.handler \
--role arn:aws:iam::123456789012:role/lambda-vpc-role \
--vpc-config SubnetIds=subnet-0532bb6758ce7c71f,subnet-
d6b7fda068036e11f,SecurityGroupIds=sg-0897d5f549934c2fb
```

Test the Lambda function

In this step, you invoke the Lambda function manually using the `invoke` command. When the Lambda function runs, it runs the SELECT query against the Employee table in the RDS MySQL instance and prints the results, which also go to the CloudWatch Logs.

1. Invoke the Lambda function with the `invoke` command.

```
aws lambda invoke --function-name CreateTableAddRecordsAndRead 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 a database 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 \(p. 257\)](#).

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. 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 role**.
4. Choose **Yes, 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**.

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 Lambda with Amazon SNS \(p. 431\)](#).

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 \(p. 140\)](#) 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":
        "v1R7PnpV2Ce81l0PRw6jlUpck7Jo5ZsQjryTjKlc5aLWGVHPZLj5NeC6qMa0emYBDXOo6QBU0Wo="
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "828aa6fc-f7b5-4305-8584-487c791949c1",
        "bucket": {
          "name": "lambda-artifacts-deafc19498e3f2df",
          "ownerIdentity": {
            "principalId": "A3I5XTEXAMA13E"
          },
          "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 \(p. 51\)](#). 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 \(p. 47\)](#).

Topics

- [Tutorial: Using an Amazon S3 trigger to invoke a Lambda function \(p. 407\)](#)
- [Tutorial: Using an Amazon S3 trigger to create thumbnail images \(p. 412\)](#)
- [AWS SAM template for an Amazon S3 application \(p. 425\)](#)

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.

We recommend that you complete this console-based tutorial before you try the [tutorial to create thumbnail images \(p. 412\)](#).

Prerequisites

To use Lambda and other AWS services, you need an AWS account. If you do not have an account, visit aws.amazon.com and choose **Create an AWS Account**. For instructions, see [How do I create and activate a new AWS account?](#)

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 [Getting started with Lambda \(p. 9\)](#) to create your first Lambda function.

Create a bucket and upload a sample object

Create an Amazon S3 bucket and upload a test file to your new bucket. Your Lambda function retrieves information about this file when you test the function from the console.

To create an Amazon S3 bucket using the console

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**.

After creating the bucket, Amazon S3 opens the **Buckets** page, which displays a list of all buckets in your account in the current Region.

To upload a test object using the Amazon S3 console

1. On the [Buckets page](#) of the Amazon S3 console, choose the name of the bucket that you created.

2. On the **Objects** tab, choose **Upload**.
3. Drag a test file from your local machine to the **Upload** page.
4. Choose **Upload**.

Create the Lambda function

Use a [function blueprint \(p. 39\)](#) to create the Lambda function. A blueprint provides a sample function that demonstrates how to use Lambda with other AWS services. Also, a blueprint includes sample code and function configuration presets for a certain runtime. For this tutorial, you can choose the blueprint for the Node.js or Python runtime.

To create a Lambda function from a blueprint in the console

1. Open the [Functions page](#) on the Lambda console.
2. Choose **Create function**.
3. On the **Create function** page, choose **Use a blueprint**.
4. Under **Blueprints**, enter **s3** in the search box.
5. In the search results, do one of the following:
 - For a Node.js function, choose **s3-get-object**.
 - For a Python function, choose **s3-get-object-python**.
6. Choose **Configure**.
7. Under **Basic information**, do the following:
 - a. For **Function name**, enter **my-s3-function**.
 - b. For **Execution role**, choose **Create a new role from AWS policy templates**.
 - c. For **Role name**, enter **my-s3-function-role**.
8. Under **S3 trigger**, choose the S3 bucket that you created previously.

When you configure an S3 trigger using the Lambda console, the console modifies your function's [resource-based policy \(p. 51\)](#) to allow Amazon S3 to invoke the function.

9. Choose **Create function**.

Review the function code

The Lambda function retrieves 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 review 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

Example lambda-function.py

```
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key']),
    encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they exist and your bucket is in the same region as this function.'.format(key, bucket))
        raise e
```

Test in the console

Invoke the Lambda function manually using sample Amazon S3 event data.

To test the Lambda function using the console

1. On the **Code** tab, under **Code source**, choose the arrow next to **Test**, and then choose **Configure test events** from the dropdown list.

2. In the **Configure test event** window, do the following:
 - a. Choose **Create new test event**.
 - b. For **Event template**, choose **Amazon S3 Put (s3-put)**.
 - c. For **Event name**, enter a name for the test event. For example, **mys3testevent**.
 - d. In the test event JSON, replace the S3 bucket name (example-bucket) and object key (test/key) with your bucket name and test file name. Your test event should look similar to the following:

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-west-2",
      "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": "EXAMPLE123/5678abcdefghijklmabdaisawesome/
mnopqrstuvwxyzABCDEFGHI"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "my-s3-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::example-bucket"
        },
        "object": {
          "key": "HappyFace.jpg",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    ]
  }
}
```

- e. Choose **Create**.
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

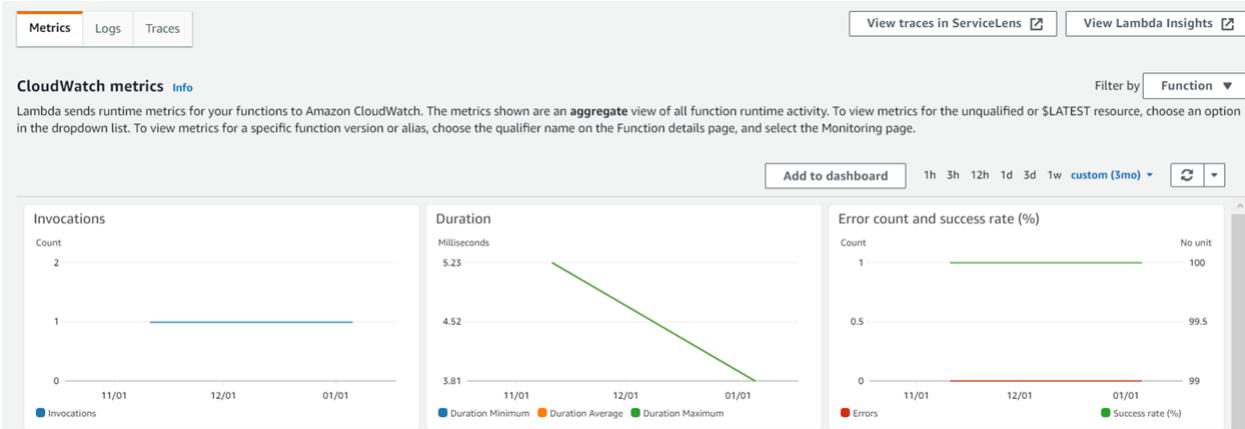
Request ID
12b3cae7-5f4e-415e-93e6-416b8f8b66e6
```

Test with 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 a few .jpg or .png image files to the bucket.
3. Open the [Functions page](#) on 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 in the AWS Lambda console \(p. 683\)](#).

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. Choose **Delete**.

To delete the IAM policy

1. Open the [Policies page](#) of the AWS Identity and Access Management (IAM) console.
2. Select the policy that Lambda created for you. The policy name begins with **AWSLambdaS3ExecutionRole-**.
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 role**.
4. Choose **Yes, 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 box.
5. Choose **Confirm**.

Next steps

Try the more advanced tutorial. In this tutorial, the S3 trigger invokes a function to [create a thumbnail image \(p. 412\)](#) 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 AWS and Lambda domain knowledge. We recommend that you first try [Tutorial: Using an Amazon S3 trigger to invoke a Lambda function \(p. 407\)](#).

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 \(p. 413\)](#)
- [Step 1. Create S3 buckets and upload a sample object \(p. 414\)](#)
- [Step 2. Create the IAM policy \(p. 414\)](#)
- [Step 3. Create the execution role \(p. 415\)](#)
- [Step 4. Create the function code \(p. 415\)](#)
- [Step 5. Create the deployment package \(p. 420\)](#)
- [Step 6. Create the Lambda function \(p. 421\)](#)
- [Step 7. Test the Lambda function \(p. 422\)](#)
- [Step 8. Configure Amazon S3 to publish events \(p. 423\)](#)
- [Step 9. Test using the S3 trigger \(p. 424\)](#)
- [Step 10. Clean up your resources \(p. 424\)](#)

Prerequisites

- AWS account

To use Lambda and other AWS services, you need an AWS account. If you do not have an account, visit aws.amazon.com and choose **Create an AWS Account**. For instructions, see [How do I create and activate a new AWS account?](#)

- Command line

To complete the following steps, you need a command line terminal or shell to run commands. 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. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

- AWS CLI

In this tutorial, you use AWS CLI commands to create and invoke the Lambda function. [Install the AWS CLI](#) and [configure it with your AWS credentials](#).

- Language tools

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 [Code authoring tools \(p. 6\)](#).

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 `mybucket` and a target bucket named `mybucket-resized`.
3. In the source bucket, [upload](#) a .jpg object, for example, `HappyFace.jpg`.

You must create this sample object before you test your Lambda function. When you test the function manually using the Lambda [invoke](#) command, you pass sample event data to the function that specifies the source bucket name and `HappyFace.jpg` as the newly created object.

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 `mybucket` 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:::mybucket/*"
    },
    {
        "Effect": "Allow",
        "Action": [
            "s3:PutObject"
        ],
        "Resource": "arn:aws:s3:::mybucket-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. 47) 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 function code

In the following code examples, the Amazon S3 event contains the source S3 bucket name and the object key name. If the object is a .jpg or a .png image file, it reads the image from the source bucket, generates a thumbnail image, and then saves the thumbnail to the target S3 bucket.

Note the following:

- The code assumes that the target bucket exists and that its name is a concatenation of the source bucket name and `-resized`.
- For each thumbnail file created, the Lambda function code derives the object key name as a concatenation of `resized-` and the source object key name. For example, if the source object key name is `sample.jpg`, the code creates a thumbnail object that has the key `resized-sample.jpg`.

Node.js

Copy the following code example into a file named `index.js`.

Example `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);
};

}
```

Python

Copy the following code example into a file named `lambda_function.py`.

Example `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), key)
```

Java

The Java code implements the `RequestHandler` interface provided in the `aws-lambda-java-core` library. When you create a Lambda function, you specify the class as the handler (in this code example, `example.handler`). For more information about using interfaces to provide a handler, see [Handler interfaces \(p. 575\)](#).

The handler uses `S3Event` as the input type, which provides convenient methods for your function code to read information from the incoming Amazon S3 event. Amazon S3 invokes your Lambda function asynchronously. Because you are implementing an interface that requires you to specify a return type, the handler uses `String` as the return type.

Copy the following code example into a file named `Handler.java`.

Example `Handler.java`

```
package example;
import java.awt.Color;
```

```
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.image.BufferedImage;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.imageio.ImageIO;

import com.amazonaws.AmazonServiceException;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.event.S3EventNotification.S3EventNotificationRecord;
import com.amazonaws.services.s3.model.GetObjectRequest;
import com.amazonaws.services.s3.model.ObjectMetadata;
import com.amazonaws.services.s3.model.S3Object;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;

public class Handler implements
    RequestHandler<S3Event, String> {
    private static final float MAX_WIDTH = 100;
    private static final float MAX_HEIGHT = 100;
    private final String JPG_TYPE = (String) "jpg";
    private final String JPG_MIME = (String) "image/jpeg";
    private final String PNG_TYPE = (String) "png";
    private final String PNG_MIME = (String) "image/png";

    public String handleRequest(S3Event s3event, Context context) {
        try {
            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;

            // Sanity check: validate that source and destination are different
            // buckets.
            if (srcBucket.equals(dstBucket)) {
                System.out
                    .println("Destination bucket must not match source bucket.");
                return "";
            }

            // Infer the image type.
            Matcher matcher = Pattern.compile(".*\\.(\\[^\\.]*)").matcher(srcKey);
            if (!matcher.matches()) {
                System.out.println("Unable to infer image type for key "
                    + srcKey);
                return "";
            }
            String imageType = matcher.group(1);
            if (!(JPG_TYPE.equals(imageType)) && !(PNG_TYPE.equals(imageType))) {
                System.out.println("Skipping non-image " + srcKey);
                return "";
            }

            // Download the image from S3 into a stream
```

```
AmazonS3 s3Client = AmazonS3ClientBuilder.defaultClient();
S3Object s3Object = s3Client.getObject(new GetObjectRequest(
    srcBucket, srcKey));
InputStream objectData = s3Object.getObjectContent();

// Read the source image
BufferedImage srcImage = ImageIO.read(objectData);
int srcHeight = srcImage.getHeight();
int srcWidth = srcImage.getWidth();
// Infer the scaling factor to avoid stretching the image
// unnaturally
float scalingFactor = Math.min(MAX_WIDTH / srcWidth, MAX_HEIGHT
    / srcHeight);
int width = (int) (scalingFactor * srcWidth);
int height = (int) (scalingFactor * srcHeight);

BufferedImage resizedImage = new BufferedImage(width, height,
    BufferedImage.TYPE_INT_RGB);
Graphics2D g = resizedImage.createGraphics();
// Fill with white before applying semi-transparent (alpha) images
g.setPaint(Color.white);
g.fillRect(0, 0, width, height);
// Simple bilinear resize
g.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
    RenderingHints.VALUE_INTERPOLATION_BILINEAR);
g.drawImage(srcImage, 0, 0, width, height, null);
g.dispose();

// Re-encode image to target format
ByteArrayOutputStream os = new ByteArrayOutputStream();
ImageIO.write(resizedImage, imageType, os);
InputStream is = new ByteArrayInputStream(os.toByteArray());
// Set Content-Length and Content-Type
ObjectMetadata meta = new ObjectMetadata();
meta.setContentLength(os.size());
if (JPG_TYPE.equals(imageType)) {
    meta.setContentType(JPG_MIME);
}
if (PNG_TYPE.equals(imageType)) {
    meta.setContentType(PNG_MIME);
}

// Uploading to S3 destination bucket
System.out.println("Writing to: " + dstBucket + "/" + dstKey);
try {
    s3Client.putObject(dstBucket, dstKey, is, meta);
}
catch(AmazonServiceException e)
{
    System.err.println(e.getErrorMessage());
    System.exit(1);
}
System.out.println("Successfully resized " + srcBucket + "/"
    + srcKey + " and uploaded to " + dstBucket + "/" + dstKey);
return "Ok";
} catch (IOException e) {
    throw new RuntimeException(e);
}
}
}
```

Step 5. Create the deployment package

The deployment package is a [.zip file archive \(p. 41\)](#) containing your Lambda function code and its dependencies.

Node.js

The sample function must include the sharp module in the deployment package.

To create a deployment package

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. Save the function code as `index.js` in a directory named `lambda-s3`.
3. Install the sharp library with npm. For Linux, use the following command:

```
npm install sharp
```

After this step, you have the following directory structure:

```
lambda-s3
|- index.js
|- /node_modules/sharp
# /node_modules/...
```

4. 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

Dependencies

- [Pillow](#)

To create a deployment package

- We recommend using the AWS SAM CLI `sam build` command with the `--use-container` option to create deployment packages that contain libraries written in C or C++, such as the [Pillow \(PIL\)](#) library.

Java

Dependencies

- `aws-lambda-java-core`
- `aws-lambda-java-events`
- `aws-java-sdk`

To create a deployment package

- 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 \(p. 577\)](#).

Step 6. Create the Lambda function

To create the function

- Create a Lambda function with the **create-function** command.

Node.js

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs12.x \
--timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/lambda-s3-role
```

The **cli-binary-format** option is required if you are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#)

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 \(p. 486\)](#). The command specifies a runtime of `nodejs12.x`. For more information, see [Lambda runtimes \(p. 195\)](#).

Python

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://function.zip --handler lambda_function.lambda_handler --runtime
python3.8 \
--timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/lambda-s3-role
```

The **cli-binary-format** option is required if you are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#)

The **create-function** command specifies the function handler as `lambda_function.lambda_handler`. This handler name reflects the function name as `lambda_handler`, and the name of the file where the handler code is stored as `lambda_function.py`. For more information, see [Lambda function handler in Python \(p. 511\)](#). The command specifies a runtime of `python3.8`. For more information, see [Lambda runtimes \(p. 195\)](#).

Java

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://function.zip --handler example.handler --runtime java11 \
--timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/lambda-s3-role
```

The **cli-binary-format** option is required if you are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#)

The **create-function** command specifies the function handler as `example.handler`. The function can use the abbreviated handler format of `package.Class` because the function implements a handler interface. For more information, see [AWS Lambda function handler in Java \(p. 573\)](#). The command specifies a runtime of `java11`. For more information, see [Lambda runtimes \(p. 195\)](#).

For the role parameter, replace `123456789012` with your [AWS account ID](#). The preceding example command specifies a 10-second timeout value as the function configuration. 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 7. Test the Lambda function

Invoke the Lambda function manually using sample Amazon S3 event data.

To test the Lambda function

1. Save the following Amazon S3 sample event data in a file named `inputFile.txt`. Be sure to replace `sourcebucket` and `HappyFace.jpg` with your source S3 bucket name and a .jpg object key, respectively.

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-west-2",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "AIDAJDPLRKG7UEXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "C3D13FE58DE4C810",
        "x-amz-id-2": "FMyUVURIY8/IgAtTv8xRjskZQpcIZ9KG4V5Wp6S7S/JRWeUWerMUE5JgHvANOjpD"
      },
      "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 \
  --invocation-type Event \
  --payload file://inputFile.txt outfile.txt
```

The **cli-binary-format** option is required if you are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#).

- Verify that the thumbnail is created in the target S3 bucket.

Step 8. 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

- 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
```

- 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 cause itself to be [invoked continuously in a loop \(p. 406\)](#).

To configure notifications

- 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 Console User Guide*.

Step 9. Test using the 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 10. 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. 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 role**.
4. Choose **Yes, 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 box.
5. Choose **Confirm**.

AWS SAM template for an Amazon S3 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 \(p. 407\)](#). 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:
  CreateThumbnail:
    Type: AWS::Serverless::Function
    Properties:
      Handler: handler
      Runtime: runtime
      Timeout: 60
      Policies: AWSLambdaExecute
      Events:
        CreateThumbnailEvent:
          Type: S3
          Properties:
            Bucket: !Ref SrcBucket
            Events: s3:ObjectCreated:*
```

```
SrcBucket:
  Type: AWS::S3::Bucket
```

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 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 \(p. 138\)](#) 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": "dGFza2lkZ29lc2hlcUK",  
            "s3Key": "customerImage1.jpg",  
            "s3VersionId": "1",  
            "s3BucketArn": "arn:aws:s3:us-east-1:0123456788: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": "dGFza2lkZ29lc2hlcUK",  
            "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 \(p. 51\)](#) 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 \(p. 47\)](#) 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 \(p. 153\)](#) 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](#).

Using AWS Lambda with Secrets Manager

Secrets Manager uses a Lambda function to [rotate the secret](#) for a secured service or database. You can customize the Lambda function to implement the service-specific details of rotating a secret.

Secrets Manager invokes the Lambda rotation function as a synchronous invocation. The event parameter contains the following fields:

```
{  
    "Step" : "request.type",  
    "SecretId" : "string",  
    "ClientRequestToken" : "string"  
}
```

For more information about using Lambda with Secrets Manager, see [Understanding Your Lambda rotation function](#).

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=jX3F0bCAI7sIbkHyy3mLYO28ieDQz2R0P8HwQkk1Fj4=; 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-2:012345678912:function:Example"  
    },  
    "spfVerdict": {  
        "status": "PASS"  
    },  
    "virusVerdict": {  
        "status": "PASS"  
    }  
},  
"eventSource": "aws:ses"  
}  
]  
}
```

For more information, see [Lambda action in the Amazon SES Developer Guide](#).

Using 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 \(p. 140\)](#) with an event that contains a message and metadata.

Example Amazon SNS message event

```
{  
    "Records": [  
        {  
            "EventVersion": "1.0",  
            "EventSubscriptionArn": "arn:aws:sns:us-east-2:123456789012: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-2.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-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": "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 \(p. 432\)](#)
- [Sample function code \(p. 436\)](#)

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 [Getting started with Lambda \(p. 9\)](#) to create your first Lambda function.

To complete the following steps, you need a command line terminal or shell to run commands. 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. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

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

From account A (01234567891A), create the source Amazon SNS topic.

```
aws sns create-topic --name sns-topic-for-lambda --profile accountA
```

Note the topic ARN that is returned by the command. You will need it when you add permissions to the Lambda function to subscribe to the topic.

Create the execution role

From account B (01234567891B), create the [execution role \(p. 47\)](#) 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

From account B (01234567891B), 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 \(p. 436\)](#).

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::01234567891B:role/service-role/lambda-sns-execution-role \
--timeout 60 --profile accountB
```

Note the function ARN that is returned by the command. You will need it when you add permissions to allow Amazon SNS to invoke your function.

Set up cross-account permissions

From account A (01234567891A), grant permission to account B (01234567891B) to subscribe to the topic:

```
aws sns add-permission --label lambda-access --aws-account-id 12345678901B \
--topic-arn arn:aws:sns:us-east-2:12345678901A:sns-topic-for-lambda \
--action-name Subscribe ListSubscriptionsByTopic --profile accountA
```

From account B (01234567891B), 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-2:12345678901A: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-2:12345678901A:sns-topic-for-lambda\"}},\n    \"Action\": [\"lambda:InvokeFunction\"],\n    \"Resource\": \"arn:aws:lambda:us-east-2:01234567891B: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 region, you need to specify the region in the principal. For an example, see [Invoking Lambda functions using Amazon SNS notifications](#) in the *Amazon Simple Notification Service Developer Guide*.

Create a subscription

From account B, subscribe the Lambda function to the topic. When a message is sent to the `sns-topic-for-lambda` topic in account A (01234567891A), Amazon SNS invokes the `Function-With-SNS` function in account B (01234567891B).

```
aws sns subscribe --protocol lambda \
--topic-arn arn:aws:sns:us-east-2:12345678901A:sns-topic-for-lambda \
--notification-endpoint arn:aws:lambda:us-east-2:12345678901B:function:Function-With-SNS \
--profile accountB
```

You should see the following output:

```
{
  "SubscriptionArn": "arn:aws:sns:us-east-2:12345678901A:sns-topic-for-lambda:5d906xxxx-7c8x-45dx-a9dx-0484e31c98xx"
}
```

The output contains the ARN of the topic subscription.

Test subscription

From account A (01234567891A), 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-2:12345678901A: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.

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 box.
5. 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 role**.
4. Choose **Yes, 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. 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 \(p. 436\)](#)
- [Java 11 \(p. 436\)](#)
- [Go \(p. 437\)](#)
- [Python 3 \(p. 437\)](#)

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 \(p. 488\)](#).

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 \(p. 577\)](#).

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 \(p. 616\)](#).

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 \(p. 514\)](#).

Using AWS Lambda with Amazon SQS

You can use an AWS Lambda function to process messages in an Amazon Simple Queue Service (Amazon SQS) queue. Lambda [event source mappings \(p. 148\)](#) 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 \(p. 138\)](#) 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. The following example shows an event for a batch of two messages.

Example Amazon SQS message event (standard queue)

```
{
  "Records": [
    {
      "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
      "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
      "body": "Test message.",
      "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIEQZJOL023YYJ4VO",
        "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": "AIDAIEQZJOL023YYJ4VO",
        "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 invokes your function as soon as records are available in the SQS queue. Lambda will poll up to 10 messages in your queue at once and will send that batch 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 five minutes by configuring a batch window. Before invoking the function, Lambda continues to poll messages from the SQS standard queue until batch window expires, the [payload limit \(p. 44\)](#) is reached or full batch size is reached.

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": "AQEBBX8nesZEXmkhsrnZeyIE8iQAMig7qw...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1573251510774",
                "SequenceNumber": "18849496460467696128",
                "MessageGroupId": "1",
                "SenderId": "AIDAIO23YVJENQZJOL4VO",
                "MessageDuplicationId": "1",
                "ApproximateFirstReceiveTimestamp": "1573251510774"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:fifo fifo",
            "awsRegion": "us-east-2"
        }
    ]
}
```

When Lambda reads a batch, the messages stay in the queue but become hidden for the length of the queue's [visibility timeout](#). If your function successfully processes the batch, Lambda deletes the messages from the queue. If your function is [throttled](#) (p. 153), returns an error, or doesn't respond, the message becomes visible again. All messages in a failed batch return to the queue, so your function code must be able to process the same message multiple times without side effects.

Sections

- [Scaling and processing \(p. 439\)](#)
- [Configuring a queue to use with Lambda \(p. 440\)](#)
- [Execution role permissions \(p. 440\)](#)
- [Configuring a queue as an event source \(p. 440\)](#)
- [Event source mapping APIs \(p. 316\)](#)
- [Tutorial: Using AWS Lambda with Amazon Simple Queue Service \(p. 442\)](#)
- [Sample Amazon SQS function code \(p. 446\)](#)
- [AWS SAM template for an Amazon SQS application \(p. 448\)](#)

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 5 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 can be processed simultaneously by an event source mapping is 1000.

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 the function returns an error, all retries are attempted 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 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 6 times the [timeout \(p. 70\)](#) that you configure on your function. The extra time allows for Lambda to retry if your function execution is throttled while your function is processing a previous batch.

If a message fails to be processed multiple times, Amazon SQS can send it to a [dead-letter queue](#). When your function returns an error, Lambda leaves it in 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 \(p. 140\)](#), 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.

- [sq:ReceiveMessage](#)
- [sq:DeleteMessage](#)
- [sq:GetQueueAttributes](#)

For more information, see [AWS Lambda execution role \(p. 47\)](#).

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](#) on the Lambda console.
2. Choose 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 SQS event sources.

Event source options

- **SQS queue** – The Amazon SQS queue to read records from.
- **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 `MaximumBatchingWindowInSeconds` parameter to at least 1 second. 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 \(p. 44\)](#) for synchronous invocation (6 MB).

Metadata is generated by both Lambda and Amazon SQS for each record. This additional metadata is counted towards the total payload size and may cause the total number of records sent in a batch to be lower than your configured batch size. The metadata fields sent by Amazon SQS can be variable in length. For more information about the Amazon SQS metadata fields, see the [ReceiveMessage](#) documentation in the *Amazon Simple Queue Service API Reference*.

- **Batch window** – Specify the maximum amount of time to gather records before invoking the function, in seconds. Only applicable to standard queues.

If you are using a batch window greater than 0 seconds, you must account for the increased processing time in your queue visibility timeout. Set your queue visibility timeout to 6 times your function timeout, 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.

Lambda processes up to 5 batches at a time. This means there are a maximum of 5 workers available to batch and process messages in parallel at any one time. Each worker will show a distinct Lambda invocation for its current batch of messages.

- **Enabled** – Set to true to enable the event source mapping. Set to false to stop processing records.

Note

Amazon SQS has a perpetual free tier for requests. Beyond the free tier, Amazon SQS charges per million requests. While your event source mapping is active, Lambda makes requests to the queue to get items. For pricing details, see [Amazon Simple Queue Service pricing](#).

To manage the event source configuration later, choose the trigger in the designer.

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. However, if your function returns an error, all items in the batch return to the queue. If you configure [reserved concurrency \(p. 84\)](#) on your function, set a minimum of 5 concurrent executions to reduce the chance of throttling errors when Lambda invokes your function. To eliminate the chance of throttling errors, set the [reserved concurrency \(p. 84\)](#) value to 1000, which is the maximum number of concurrent executions for an Amazon SQS event source.

Event source mapping APIs

To manage an event source with the [AWS CLI](#) or [AWS SDK](#), you can use the following API operations:

- [CreateEventSourceMapping \(p. 753\)](#)
- [ListEventSourceMappings \(p. 852\)](#)
- [GetEventSourceMapping \(p. 802\)](#)
- [UpdateEventSourceMapping \(p. 918\)](#)
- [DeleteEventSourceMapping \(p. 778\)](#)

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"  
}
```

Tutorial: Using AWS Lambda with Amazon Simple Queue Service

In this tutorial, you create a Lambda function to consume messages from an [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 [Getting started with Lambda \(p. 9\)](#) to create your first Lambda function.

To complete the following steps, you need a command line terminal or shell to run commands. 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. On Windows 10, you can [install the Windows Subsystem for Linux](#) to get a Windows-integrated version of Ubuntu and Bash.

Create the execution role

Create the [execution role \(p. 47\)](#) 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 – **AWSLambdaSQSQueueExecutionRole**.
- Role name – **lambda-sqs-role**.

The **AWSLambdaSQSQueueExecutionRole** policy has the permissions that the function needs to read items from Amazon SQS and write logs to CloudWatch Logs.

Create the function

The following example shows how to process each Amazon SQS message in the event input. See [Using AWS Lambda with Amazon SQS](#) for an example of an event with multiple messages. In the example, the code writes each message to a log in CloudWatch Logs.

Note

For sample code in other languages, see [Sample Amazon SQS function code \(p. 446\)](#).

Example index.js

```
exports.handler = async function(event, context) {
  event.Records.forEach(record => {
    const { body } = record;
    console.log(body);
  });
  return {};
}
```

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 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 Simple Queue Service event.

If the handler returns normally without exceptions, Lambda considers the message processed successfully and begins reading new messages in the queue. Once a message is processed successfully, it is automatically deleted from the queue. If the handler throws an exception, Lambda considers the input of messages as not processed and invokes the function with the same batch of messages.

1. Copy the following JSON into a file and save it as `input.txt`.

```
{
  "Records": [
```

```
{  
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",  
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",  
    "body": "test",  
    "attributes": {  
        "ApproximateReceiveCount": "1",  
        "SentTimestamp": "1545082649183",  
        "SenderId": "AIDAENQZJOL023YVJ4VO",  
        "ApproximateFirstReceiveTimestamp": "1545082649185"  
    },  
    "messageAttributes": {},  
    "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",  
    "eventSource": "aws:sqs",  
    "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",  
    "awsRegion": "us-east-2"  
}  
]  
}
```

- Run the following `invoke` command.

```
aws lambda invoke --function-name ProcessSQSRecord \  
--payload file://input.txt outfile.txt
```

The **cli-binary-format** option is required if you are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#).

- Verify the output in the `outfile.txt` file.

Create an Amazon SQS queue

Create an Amazon SQS queue that the Lambda function can use as an event source.

To create a queue

- Sign in to the AWS Management Console and open the Amazon SQS console at <https://console.aws.amazon.com/sqs/>.
- In the Amazon SQS console, create a queue.
- Write down or otherwise record the identifying queue ARN (Amazon Resource Name). You need this in the next step when you associate the queue with your Lambda function.

Create an event source mapping in AWS Lambda. This event source mapping associates the Amazon SQS queue with your Lambda function. After you create this event source mapping, AWS Lambda starts polling the queue.

Test the end-to-end experience. As you perform queue updates, Amazon Simple Queue Service writes messages to the queue. AWS Lambda polls the queue, detects new records and invokes your Lambda function on your behalf by passing events, in this case Amazon SQS messages, to the function.

Configure the event source

To create a mapping between the specified Amazon SQS queue and the Lambda function, run the following AWS CLI `create-event-source-mapping` command. After the command runs, write down or otherwise record the UUID. You'll need this UUID to refer to the event source mapping in any other commands, for example, if you choose to delete the event source mapping.

```
aws lambda create-event-source-mapping --function-name ProcessSQSRecord --batch-size 10 \
```

```
--event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue
```

You can get the list of event source mappings by running the following command.

```
aws lambda list-event-source-mappings --function-name ProcessSQSRecord \  
--event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue
```

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 queue) and `OK` (indicates AWS Lambda successfully read records from the queue and invoked your Lambda function) indicate that there are no issues. If there are issues, you receive an error message.

Test the setup

Now you can test the setup as follows:

1. In the Amazon SQS console, send messages to the queue. Amazon SQS writes records of these actions to the queue.
2. AWS Lambda polls the queue and when it detects updates, it invokes your Lambda function by passing in the event data it finds in the queue.
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 execution role

1. Open the [Roles page](#) of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete role**.
4. Choose **Yes, 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. 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 box.
5. Choose **Delete**.

Sample Amazon SQS function code

Sample code is available for the following languages.

Topics

- [Node.js \(p. 446\)](#)
- [Java \(p. 446\)](#)
- [C# \(p. 447\)](#)
- [Go \(p. 447\)](#)
- [Python \(p. 448\)](#)

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 \(p. 488\)](#).

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 \(p. 577\)](#).

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 \(p. 644\)](#).

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 \(p. 616\)](#).

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 \(p. 514\)](#).

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 \(p. 442\)](#). 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 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.

To trace requests that don't have a tracing header, enable active tracing in your function's configuration.

To enable active tracing

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring tools**.
4. Choose **Edit**.
5. Under **X-Ray**, enable **Active tracing**.
6. Choose **Save**.

Pricing

X-Ray has a perpetual free tier. Beyond the free tier threshold, X-Ray charges for trace storage and retrieval. For details, see [AWS X-Ray pricing](#).

Your function needs permission to upload trace data to X-Ray. When you enable active tracing in the Lambda console, Lambda adds the required permissions to your function's [execution role \(p. 47\)](#). Otherwise, add the [AWSXRayDaemonWriteAccess](#) policy to the execution role.

X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of the requests that your application serves. The default sampling rule is 1 request per second and 5 percent of additional requests. This sampling rate cannot be configured for Lambda 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 \(p. 200\)](#).

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.

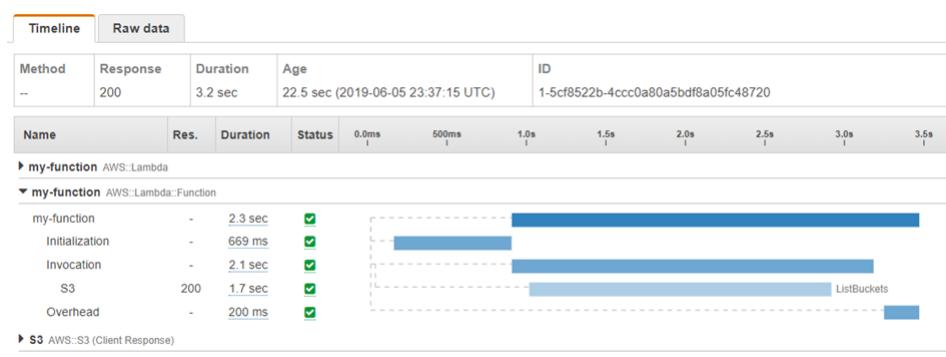
The Overhead subsegment represents the phase that occurs between the time when the runtime sends the response and the signal for the next invoke. During this time, the runtime finishes all tasks related to an invoke and prepares to freeze the sandbox.

Note

If your Lambda function uses [provisioned concurrency \(p. 87\)](#), your X-Ray trace might display a function initialization with a very long duration.

Provisioned concurrency initializes function instances in advance, to reduce lag at the time of invocation. Over time, provisioned concurrency refreshes these instances by creating new instances to replace the old ones. For workloads with steady traffic, the new instances are initialized well in advance of their first invocation. The time gap gets recorded in the X-Ray trace as the initialization duration.

The following example shows a trace with 2 segments. Both are named **my-function**, but one is type AWS::Lambda and the other is AWS::Lambda::Function. The function segment is expanded to show its subsegments.



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 \(p. 504\)](#)
- [Instrumenting Python code in AWS Lambda \(p. 542\)](#)
- [Instrumenting Ruby code in AWS Lambda \(p. 565\)](#)
- [Instrumenting Java code in AWS Lambda \(p. 601\)](#)
- [Instrumenting Go code in AWS Lambda \(p. 632\)](#)

- [Instrumenting C# code in AWS Lambda \(p. 662\)](#)

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 \(p. 452\)](#)
- [The AWS X-Ray daemon \(p. 452\)](#)
- [Enabling active tracing with the Lambda API \(p. 452\)](#)
- [Enabling active tracing with AWS CloudFormation \(p. 453\)](#)

Execution role permissions

Lambda needs the following permissions to send trace data to X-Ray. Add them to your function's [execution role \(p. 47\)](#).

- `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 \(p. 935\)](#)
- [GetFunctionConfiguration \(p. 816\)](#)
- [CreateFunction \(p. 762\)](#)

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 that is locked 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 enable active 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  
      ...
```

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 \(p. 454\)](#)
- [Managing state machines in the Lambda console \(p. 457\)](#)
- [Orchestration examples with Step Functions \(p. 459\)](#)

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 \(p. 454\)](#)
- [State machine application patterns \(p. 454\)](#)
- [Applying patterns to state machines \(p. 455\)](#)
- [Example branching application pattern \(p. 455\)](#)

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.

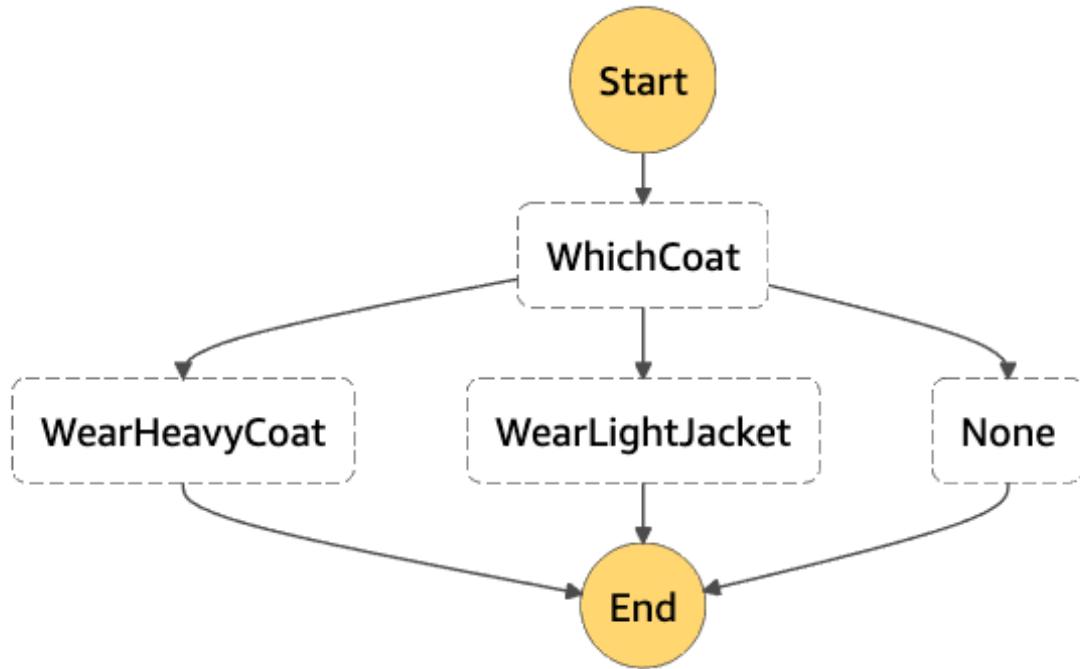
In addition to application patterns, Step Functions supports various [service integration patterns](#), including the ability to pause a workflow for human approval, or to call a legacy system or other third party.

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 view details about your Step Functions state machines and the Lambda functions that they use.

Sections

- [Viewing state machine details \(p. 458\)](#)

- [Editing a state machine \(p. 458\)](#)
- [Running a state machine \(p. 458\)](#)

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 \(p. 459\)](#)
- [Configuring a state machine as an event source \(p. 459\)](#)
- [Handling function and service errors \(p. 460\)](#)
- [AWS CloudFormation and AWS SAM \(p. 461\)](#)

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 \(p. 472\)](#)

```
{
    "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 \(p. 839\)](#) 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:

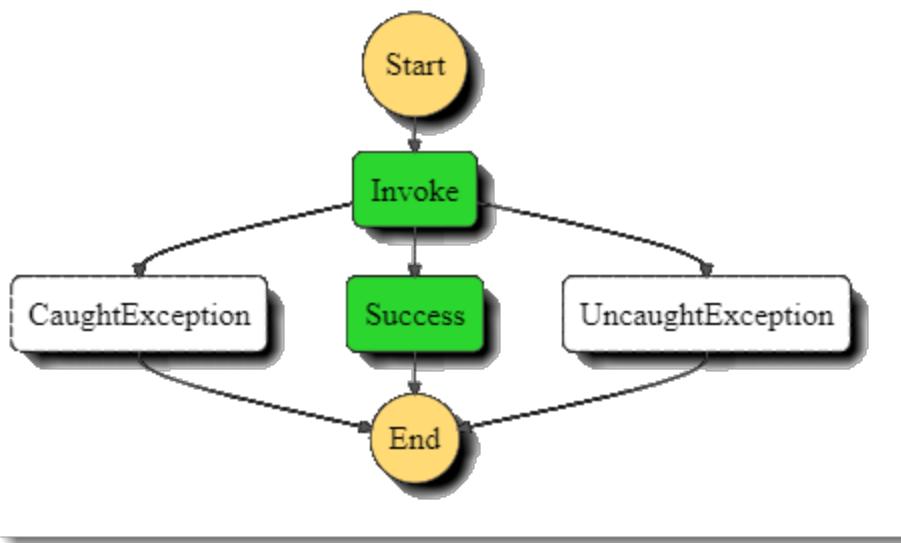
```

```

- 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*.

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

- [blank-java](#) – A Java function that shows the use of Lambda's Java libraries, logging, environment variables, layers, AWS X-Ray tracing, unit tests, and the AWS SDK.
- [java-basic](#) – A minimal Java function with unit tests and variable logging configuration.

- [java-events](#) – A minimal Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that don't require the AWS SDK as a dependency, such as Amazon API Gateway.
- [java-events-v1sdk](#) – A Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that require the AWS SDK as a dependency (Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, and Amazon Kinesis).
- [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.

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:

Sample Lambda applications in PowerShell

- [blank-powershell](#) – A PowerShell function that shows the use of logging, environment variables, and the AWS SDK.

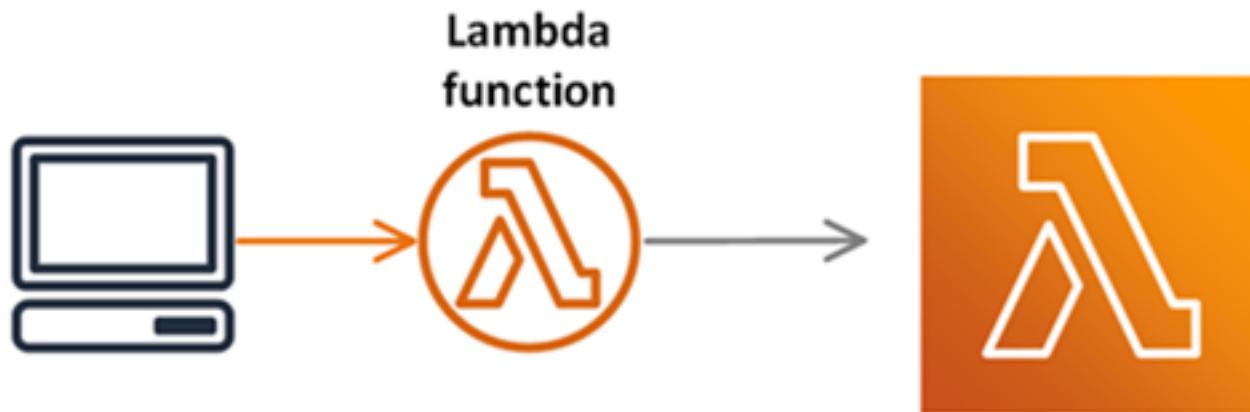
To deploy a sample application, follow the instructions in its README file. To learn more about the architecture and use cases of an application, read the topics in this chapter.

Topics

- [Blank function sample application for AWS Lambda \(p. 465\)](#)
- [Error processor sample application for AWS Lambda \(p. 472\)](#)
- [List manager sample application for AWS Lambda \(p. 477\)](#)

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 \(p. 466\)](#)
- [Deployment automation with AWS CloudFormation and the AWS CLI \(p. 467\)](#)
- [Instrumentation with the AWS X-Ray \(p. 469\)](#)
- [Dependency management with layers \(p. 470\)](#)

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()
}

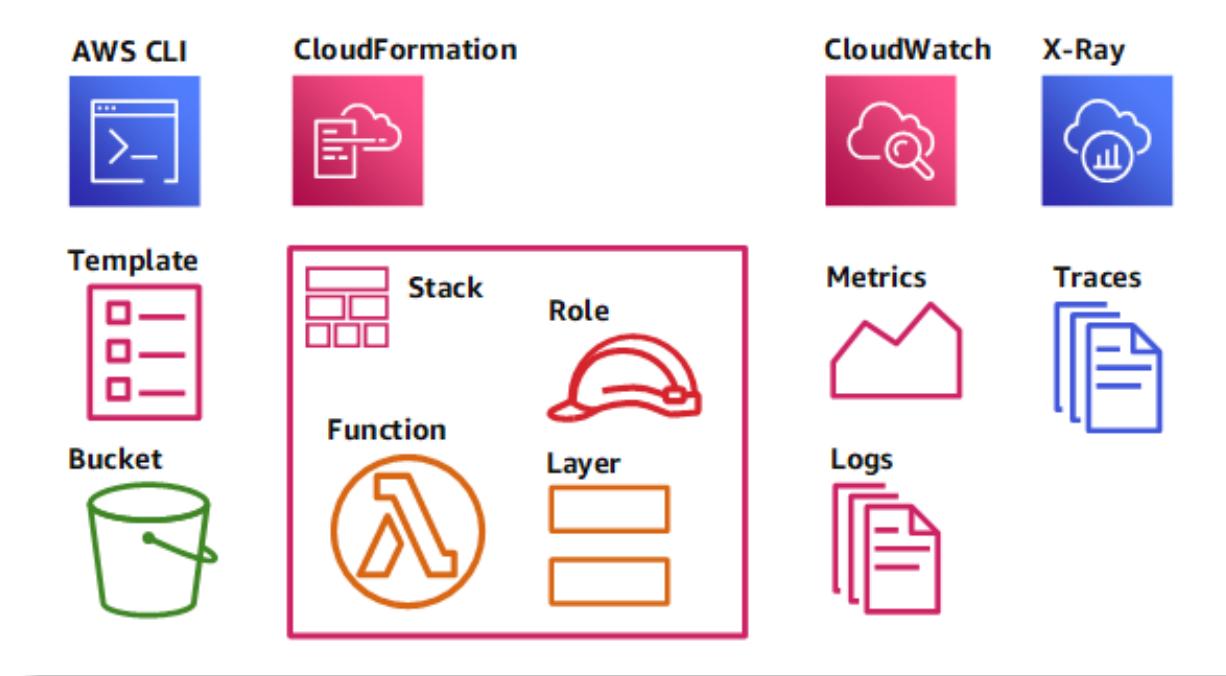
var serialize = function(object) {
    return JSON.stringify(object, null, 2)
}
```

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 AWS Lambda with Amazon SQS \(p. 438\)](#).

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
```

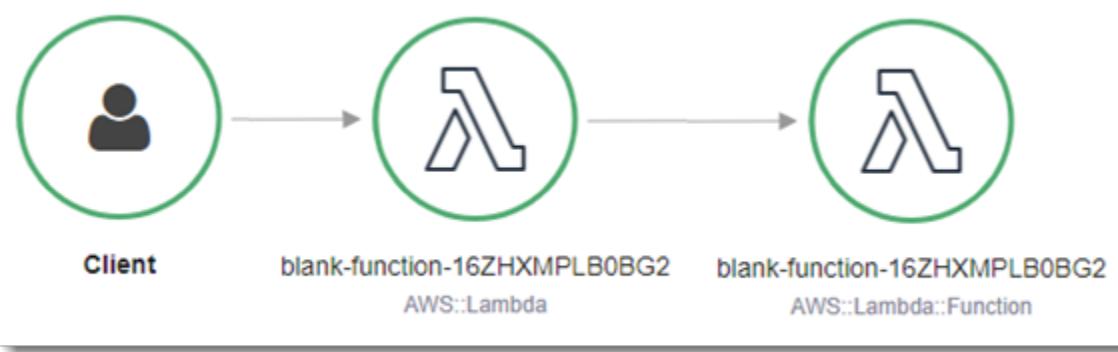
```
set -eo pipefail
ARTIFACT_BUCKET=$(cat bucket-name.txt)
aws cloudformation package --template-file template.yml --s3-bucket $ARTIFACT_BUCKET --
output-template-file out.yml
aws cloudformation deploy --template-file out.yml --stack-name blank-nodejs --capabilities
CAPABILITY_NAMED_IAM
```

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.

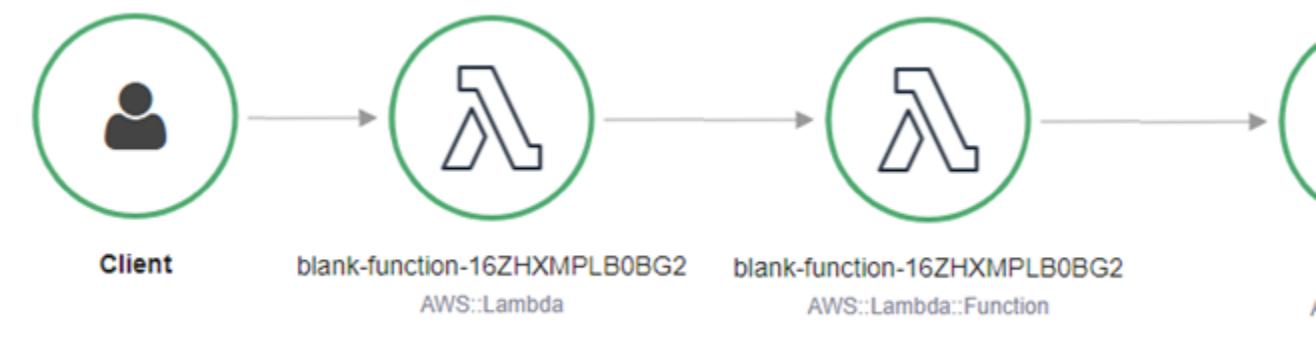
To record additional detail, the sample function uses the X-Ray SDK. With minimal changes to the function code, the X-Ray SDK records details about calls made with the AWS SDK to AWS services.

Example [blank-nodejs/function/index.js](#) – Instrumentation

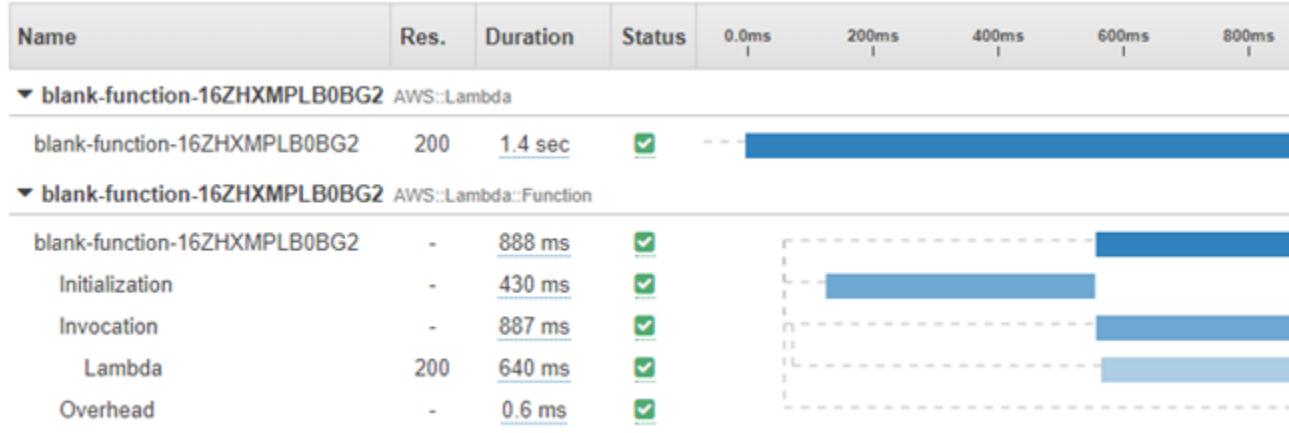
```
const AWSXRay = require('aws-xray-sdk-core')
const AWS = AWSXRay.captureAWS(require('aws-sdk'))

// 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 \(p. 101\)](#) 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 \(p. 102\)](#).

Example 2-build-layer.sh – Preparing the layer

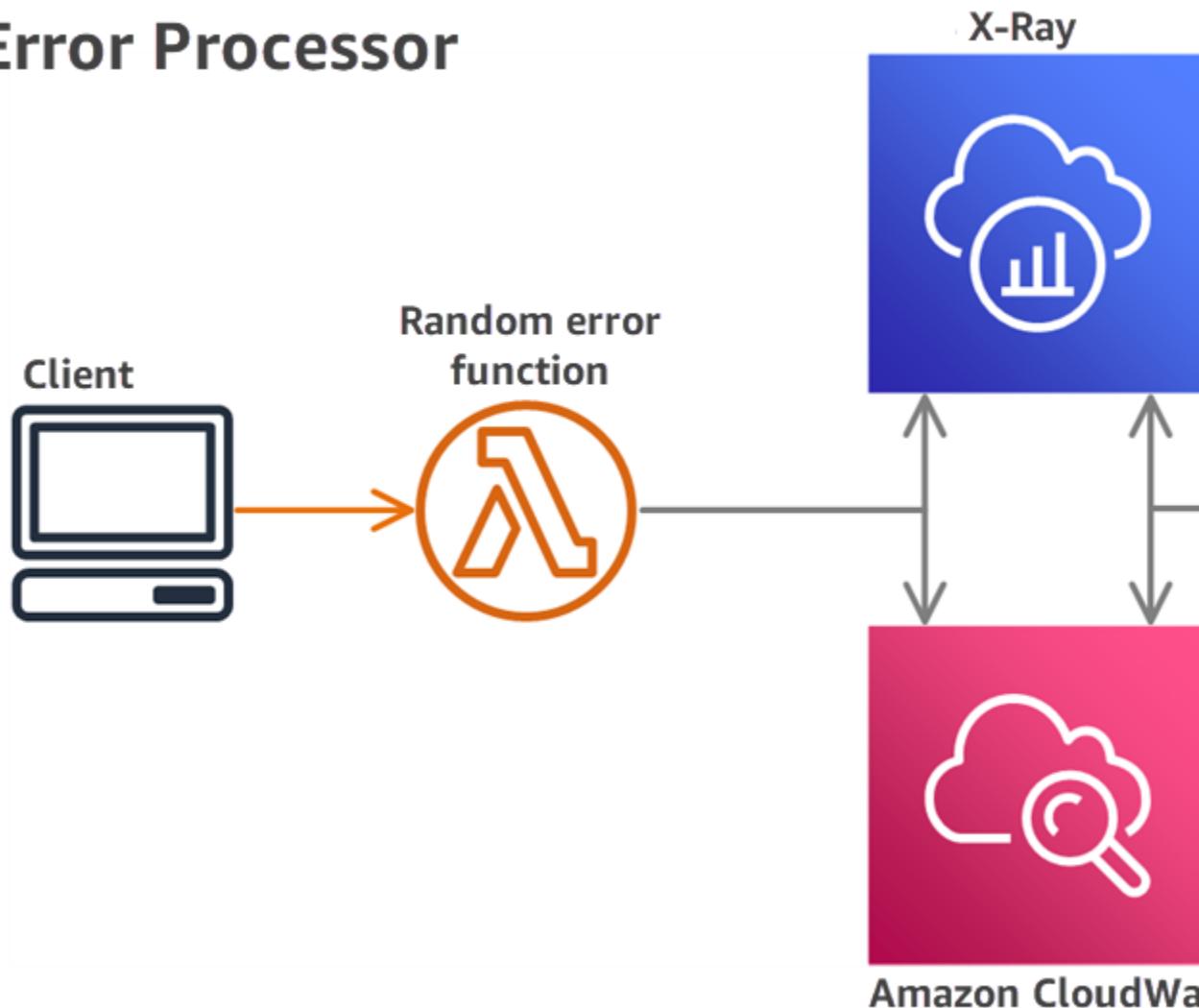
```
#!/bin/bash
set -eo pipefail
mkdir -p lib/nodejs
rm -rf node_modules lib/nodejs/node_modules
npm install --production
mv node_modules lib/nodejs/
```

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 \(p. 302\)](#). 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.

Error Processor



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 \(p. 473\)](#)
- [Instrumentation with AWS X-Ray \(p. 474\)](#)
- [AWS CloudFormation template and additional resources \(p. 474\)](#)

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": "H4sIAAAAAAAHWQT0/DMAzFv0vEkbLYcdJkt4qVXmCDteIAm1DbZKjS
+kdpB0Jo350MhsQFyVLsZ+unl/fJWjeO5asrPgbH5 . . ."
  }
}
```

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-15OPDVQ59CG07"
  ],
  "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

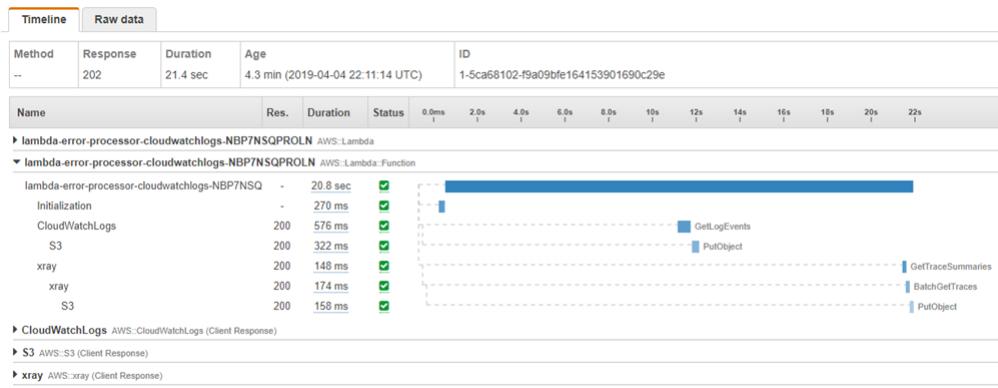
allow the log stream and trace time to finalize, the function waits for a short period of time before accessing the data.

Instrumentation with AWS X-Ray

The application uses [AWS X-Ray \(p. 450\)](#) 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 following service map shows the random error function generating errors for some requests. It also shows the processor function calling X-Ray, CloudWatch Logs, and Amazon S3.



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 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.



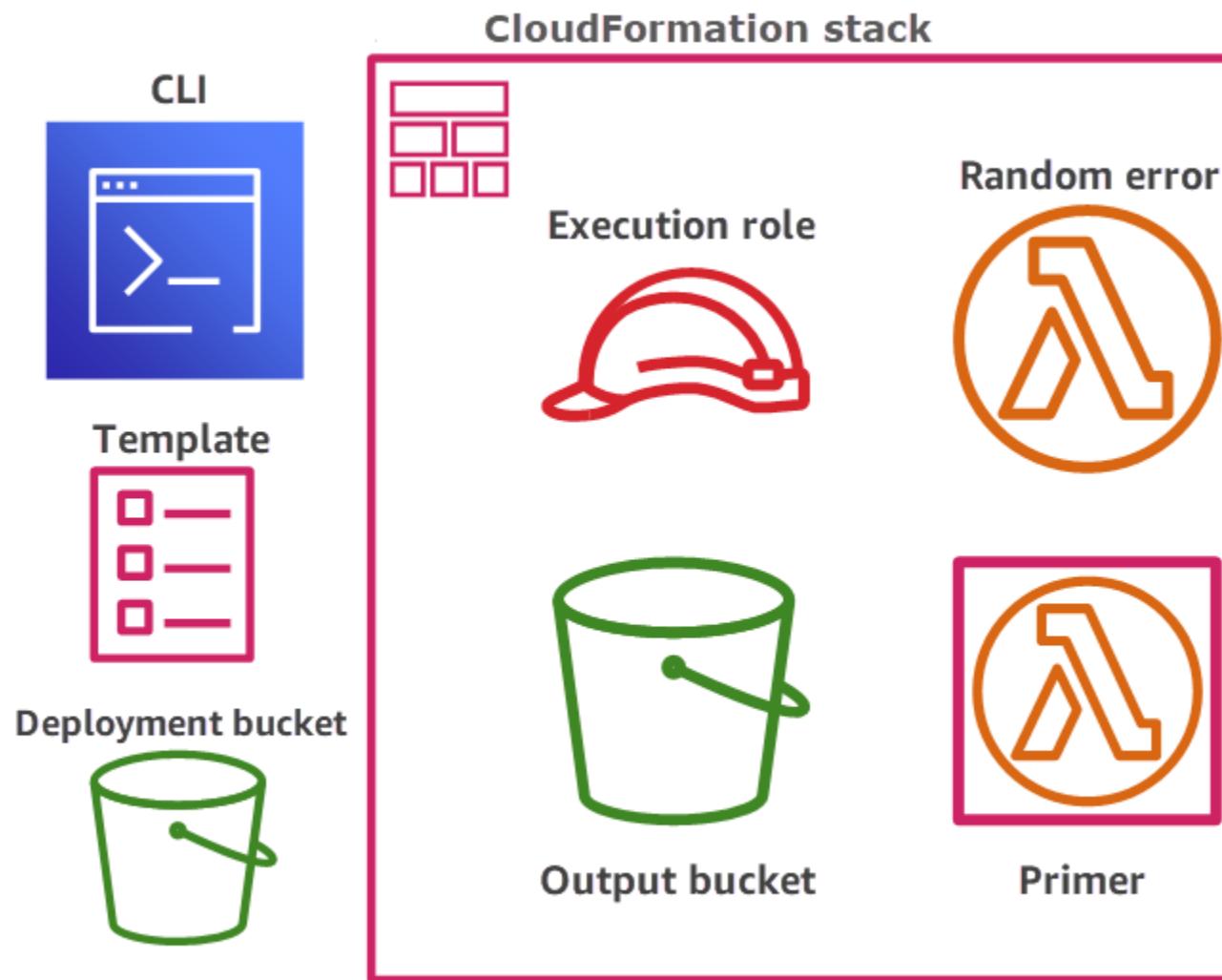
The processor function gets the request ID from the CloudWatch Logs event, and uses the AWS SDK for JavaScript to search X-Ray for that request. It uses AWS SDK clients, which are instrumented with the X-Ray SDK, to download the trace and log stream. Then it stores them in the output bucket. The X-Ray 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.



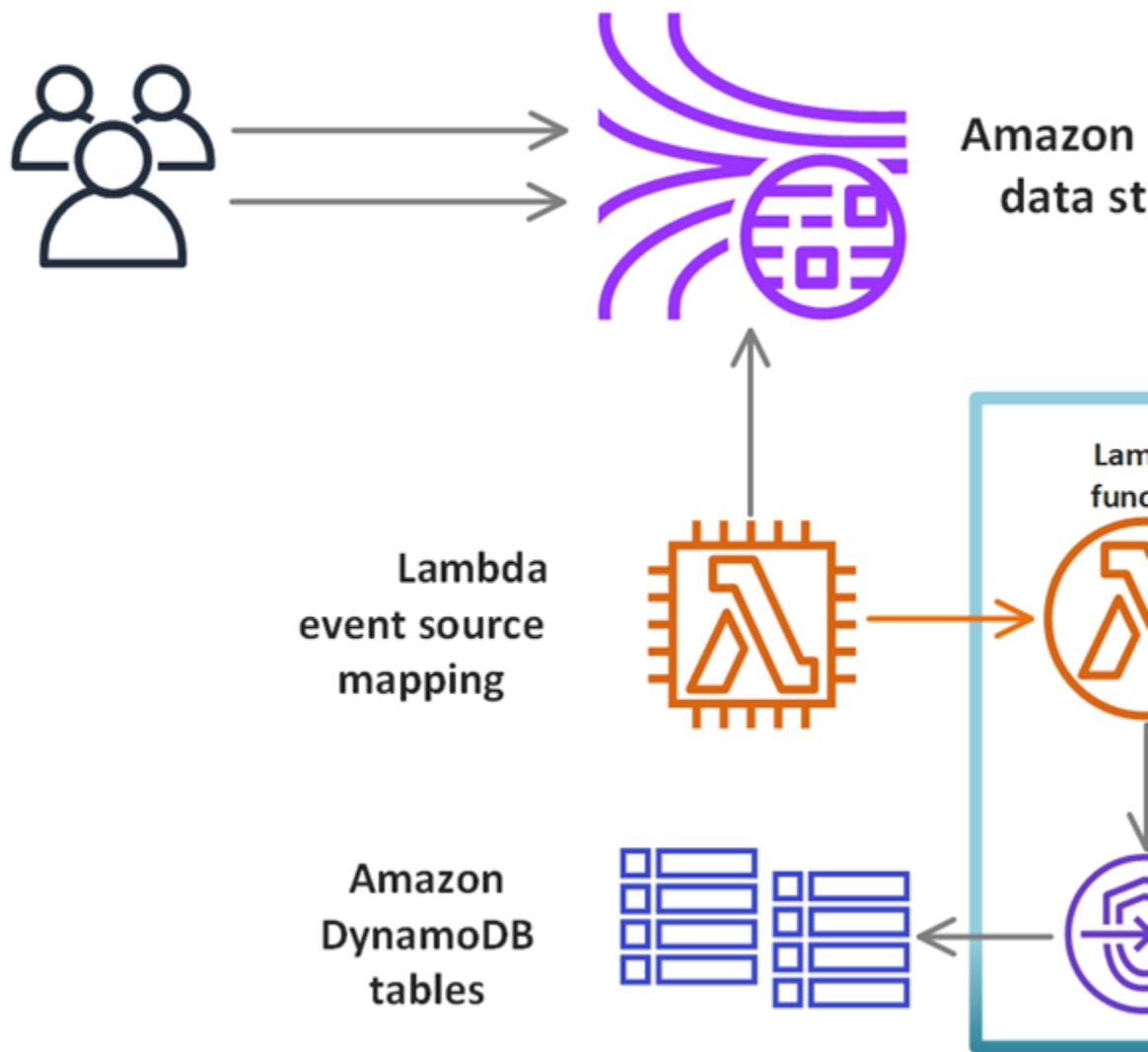
To work around a limitation of Lambda's integration with AWS CloudFormation, the template creates an additional function that runs during deployments. All Lambda functions come with a CloudWatch Logs log group that stores output from function executions. However, the log group isn't created until the 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).

List manager application



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 \(p. 478\)](#)
- [Instrumentation with AWS X-Ray \(p. 479\)](#)
- [AWS CloudFormation templates and additional resources \(p. 482\)](#)

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 \(p. 148\)](#) 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.

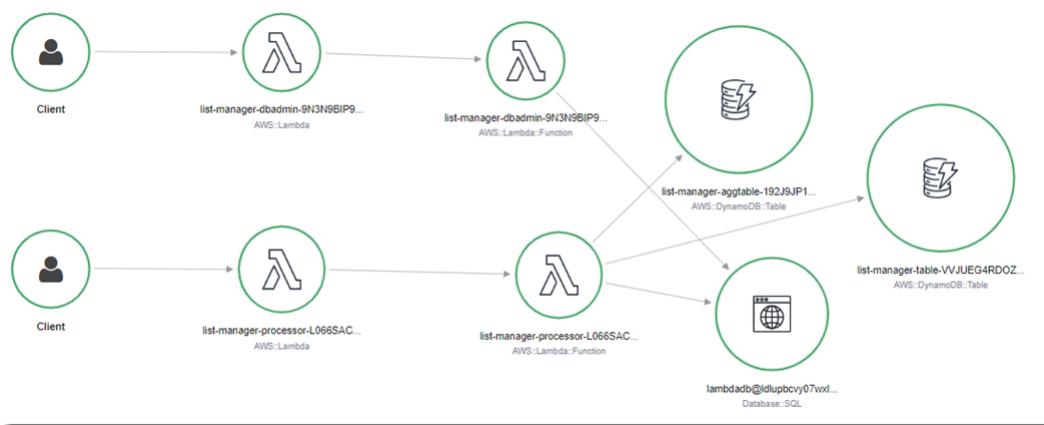
Example `events/kinesis.json` – Record

```
{
  "Records": [
    {
      "kinesis": {
        "kinesisSchemaVersion": "1.0",
        "partitionKey": "0",
        "sequenceNumber": "49598630142999655949899443842509554952738656579378741250",
        "data": "eyJ0aXRsi6ICJmYXZvcml0ZSBtb3ZpZXMiLCI6ICJy5c2N0IiwgInR5cGUoAicmFuayIsICJlbnRyaWVzIjog
          "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-87B3XMPLF1AZ"
      },
      ...
    ]
}
```

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.

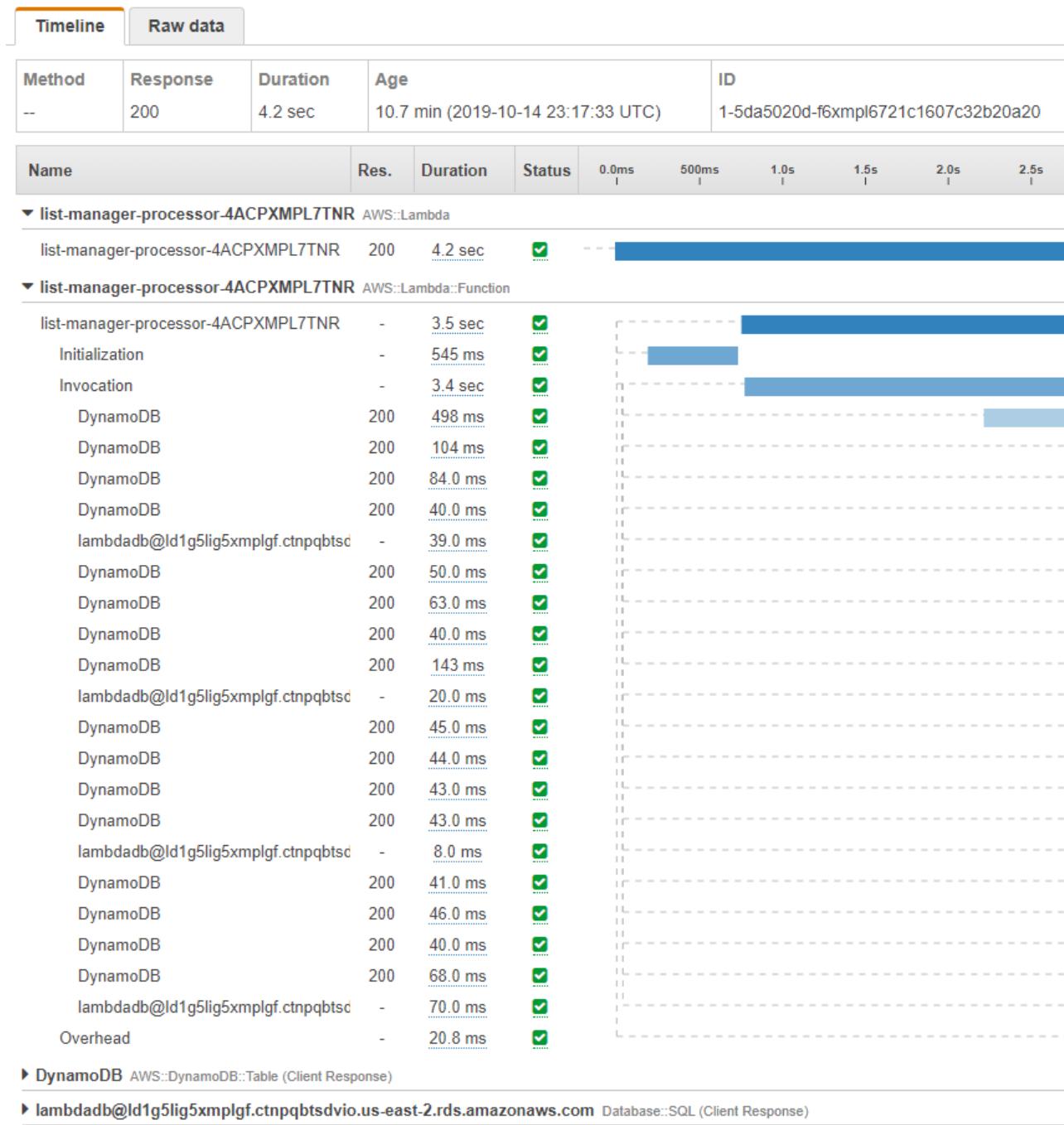
Instrumentation with AWS X-Ray

The application uses [AWS X-Ray \(p. 450\)](#) 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 following service map shows the function communicating with two DynamoDB tables and a MySQL database.



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.

Traces > Details



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.

Building Lambda functions with Node.js

You can run JavaScript code with Node.js in AWS Lambda. Lambda provides [runtimes \(p. 195\)](#) 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 runtimes

Name	Identifier	SDK for JavaScript	Operating system
Node.js 14	nodejs14.x	2.880.0	Amazon Linux 2
Node.js 12	nodejs12.x	2.880.0	Amazon Linux 2
Node.js 10	nodejs10.x	2.880.0	Amazon Linux 2

Note

For end of support information about Node.js 10, see [the section called "Runtime support policy" \(p. 198\)](#).

Lambda functions use an [execution role \(p. 47\)](#) 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 14.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 **Create**.
8. To invoke the function, choose **Test**.

The console creates a Lambda function with a single source file named `index.js`. You can edit this file and add more files in the built-in [code editor \(p. 17\)](#). 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` file exports a function named `handler` that takes an event object and a context object. This is the [handler function \(p. 486\)](#) 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`.

Each time you save your function code, the Lambda console creates 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 \(p. 488\)](#) and updating your code at the command line.

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 \(p. 493\)](#) 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 \(p. 495\)](#) during invocation. If your function [returns an error \(p. 500\)](#), Lambda formats the error and returns it to the invoker.

Topics

- [AWS Lambda function handler in Node.js \(p. 486\)](#)
- [Deploy Node.js Lambda functions with .zip file archives \(p. 488\)](#)
- [Deploy Node.js Lambda functions with container images \(p. 491\)](#)
- [AWS Lambda context object in Node.js \(p. 493\)](#)
- [AWS Lambda function logging in Node.js \(p. 495\)](#)
- [AWS Lambda function errors in Node.js \(p. 500\)](#)
- [Instrumenting Node.js code in AWS Lambda \(p. 504\)](#)

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.

Example index.js

```
exports.handler = async function(event, context) {
  console.log("EVENT: \n" + JSON.stringify(event, null, 2))
  return context.logStreamName
}
```

When you [configure a function \(p. 70\)](#), 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 three 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 \(p. 839\)](#), and the runtime converts it to an object. When an AWS service invokes your function, the event structure [varies by service \(p. 257\)](#).

The second argument is the [context object \(p. 493\)](#), which contains information about the invocation, function, and execution environment. In the preceding example, the function gets the name of the [log stream \(p. 495\)](#) from the context object and returns it to the invoker.

The third argument, `callback`, is a function that you can call in [non-async handlers \(p. 487\)](#) to send a response. The callback function takes two arguments: an `Error` and a response. When you call it, Lambda waits for the event loop to be empty and then returns the response or error to the invoker. The response object must be compatible with `JSON.stringify`.

For async handlers, you return a response, error, or promise to the runtime instead of using `callback`.

Async handlers

For async handlers, you can use `return` and `throw` to send a response or error, respectively. Functions must use the `async` keyword to use these methods to return a response or error.

If your code performs an asynchronous task, return a promise to make sure that it finishes running. When you resolve or reject the promise, Lambda sends the response or error to the invoker.

Example index.js file – HTTP request with async handler and promises

```
const https = require('https')
let url = "https://docs.aws.amazon.com/lambda/latest/dg/welcome.html"

exports.handler = async function(event) {
  const promise = new Promise(function(resolve, reject) {
    https.get(url, (res) => {
      resolve(res.statusCode)
    }).on('error', (e) => {
      reject(Error(e))
    })
  })
  return promise
}
```

```
}
```

For libraries that return a promise, you can return that promise directly to the runtime.

Example index.js file – AWS SDK with async handler and promises

```
const AWS = require('aws-sdk')
const s3 = new AWS.S3()

exports.handler = async function(event) {
    return s3.listBuckets().promise()
}
```

Non-async handlers

The following example function checks a URL and returns the status code to the invoker.

Example index.js file – HTTP request with callback

```
const https = require('https')
let url = "https://docs.aws.amazon.com/lambda/latest/dg/welcome.html"

exports.handler = function(event, context, callback) {
    https.get(url, (res) => {
        callback(null, res.statusCode)
    }).on('error', (e) => {
        callback(Error(e))
    })
}
```

For non-async handlers, function execution continues 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 \(p. 493\)](#) to false.

In the following 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.

Example index.js file – callbackWaitsForEmptyEventLoop

```
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)
}
```

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 \(p. 195\)](#) 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*.

- Lambda uses POSIX file permissions, so you may need to [set permissions for the deployment package folder](#) before you create the .zip file archive.

This section describes how to create a .zip file as your deployment package, and then use the .zip file to deploy your function code to Lambda using the AWS Command Line Interface (AWS CLI).

Sections

- [Prerequisites \(p. 488\)](#)
- [Updating a function with no dependencies \(p. 488\)](#)
- [Updating a function with additional dependencies \(p. 489\)](#)

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 CLI – Install 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 \(p. 926\)](#) 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": "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 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 \(p. 483\)](#), 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 file:///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 \(p. 248\)](#). AWS provides the following resources to help you build a container image for your Node.js 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, add a runtime interface client to the base image to make it compatible with Lambda.

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 ECR container registry. See steps 7-9 in [Create image \(p. 249\)](#).
3. [Create \(p. 72\)](#) the Lambda function and deploy the image.

AWS base images for Node.js

AWS provides the following base images for Node.js:

Tags	Runtime	Operating system	Dockerfile
14	NodeJS 14.x	Amazon Linux 2	Dockerfile for Node.js 14.x on GitHub
12	NodeJS 12.x	Amazon Linux 2	Dockerfile for Node.js 12.x on GitHub
10	NodeJS 10.x	Amazon Linux 2	Dockerfile for Node.js 10.x on GitHub

Docker Hub repository: [amazon/aws-lambda-nodejs](#)

Amazon ECR repository: [gallery.ecr.aws/lambda/nodejs](#)

Using a Node.js base image

For instructions on how to use a Node.js base image, choose the **usage** tab on [AWS Lambda base images for Node.js](#) in the *Amazon ECR repository*.

The instructions are also available on [AWS Lambda base images for Node.js](#) in the *Docker Hub repository*.

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.

AWS Lambda context object in Node.js

When Lambda runs your function, it passes a context object to the [handler \(p. 486\)](#). 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 \(p. 95\)](#) 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 \(p. 487\)](#) 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 function metrics 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.

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 \(p. 495\)](#)
- [Using the Lambda console \(p. 496\)](#)
- [Using the CloudWatch console \(p. 496\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 496\)](#)
- [Deleting logs \(p. 499\)](#)

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: 3d752xmp11bbe37e 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 \(p. 450\)](#).
- **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 \(p. 694\)](#).

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 \(p. 200\)](#). 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 \(p. 472\)](#).

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 CLI – Install 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":  
        "U1RBULQgUmVxdWVzdElkOiaA4N2QwNDRiOC1mMTU0LTEzZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",  
    "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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#)

```
#!/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
```

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.

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 \(p. 500\)](#)
- [How it works \(p. 500\)](#)
- [Using the Lambda console \(p. 501\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 501\)](#)
- [Error handling in other AWS services \(p. 502\)](#)
- [What's next? \(p. 503\)](#)

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 \(p. 195\)](#) 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 [Invoke API Errors \(p. 841\)](#).

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 \(p. 450\)](#) is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Test**.
4. Select **New event** and then choose an **Event template** from the dropdown list.
5. Enter a name for the test event.
6. Enter the JSON for the test event.
7. Choose **Create event**.
8. Choose **Invoke**.

The Lambda console invokes your function [synchronously \(p. 138\)](#) 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 CLI – Install 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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#).

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)"]}
```

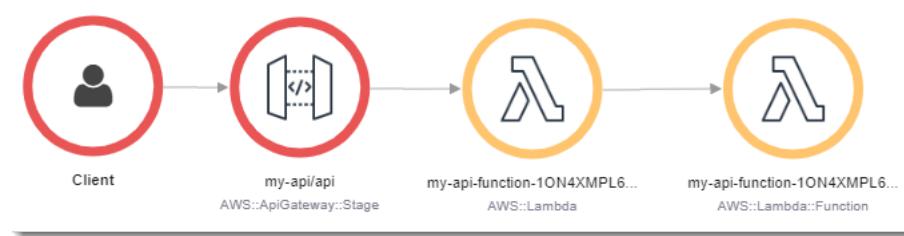
Lambda also records up to 256 KB of the error object in the function's logs. For more information, see [AWS Lambda function logging in Node.js \(p. 495\)](#).

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 \(p. 504\)](#).

What's next?

- Learn how to show logging events for your Lambda function on the [the section called "Logging" \(p. 495\)](#) page.

Instrumenting Node.js 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.

The X-Ray *service map* shows the flow of requests through your application. The following example from the [error processor \(p. 472\)](#) sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon S3 and Amazon CloudWatch Logs.



To trace requests that don't have a tracing header, enable active tracing in your function's configuration.

To enable active tracing

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring tools**.
4. Choose **Edit**.
5. Under **X-Ray**, enable **Active tracing**.
6. Choose **Save**.

Pricing

X-Ray has a perpetual free tier. Beyond the free tier threshold, X-Ray charges for trace storage and retrieval. For details, see [AWS X-Ray pricing](#).

Your function needs permission to upload trace data to X-Ray. When you enable active tracing in the Lambda console, Lambda adds the required permissions to your function's [execution role \(p. 47\)](#). Otherwise, add the [AWSXRayDaemonWriteAccess](#) policy to the execution role.

X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of the requests that your application serves. The default sampling rule is 1 request

per second and 5 percent of additional requests. This sampling rate cannot be configured for Lambda functions.

When active tracing is enabled, Lambda records a trace for a subset of invocations. Lambda records two *segments*, which creates two nodes on the service map. The first node represents the Lambda service that receives the invocation request. The second node is recorded by the function's [runtime \(p. 30\)](#).



Configuration

The Lambda runtime sets some environment variables to configure the X-Ray SDK, including `AWS_XRAY_CONTEXT_MISSING`. 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. For more information, see [Runtime environment variables \(p. 80\)](#).

Example Example initialization code

```
const AWSXRay = require('aws-xray-sdk-core');

// Configure the context missing strategy to do nothing
AWSXRay.setContextMissingStrategy(() => {});
```

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 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.

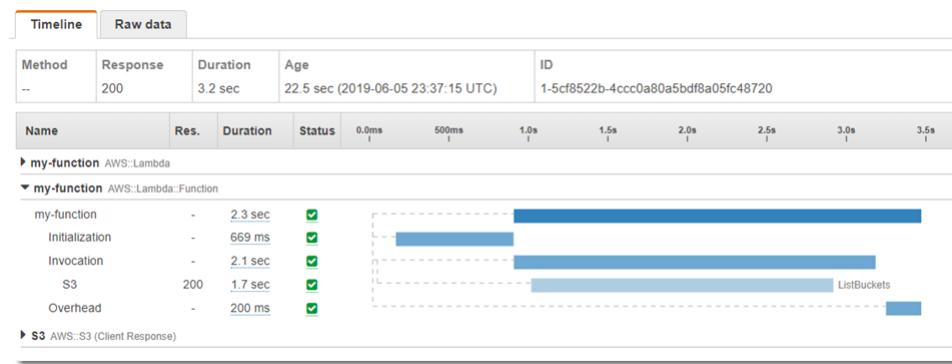
Example [blank-nodejs/function/index.js](#) – Tracing an AWS SDK client

```
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 following example shows a trace with 2 segments. Both are named **my-function**, but one is type `AWS::Lambda` and the other is `AWS::Lambda::Function`. The function segment is expanded to show its subsegments.



The first segment represents the invocation request processed by the Lambda service. The second segment records the work done by your function. The function segment has 3 subsegments.

- **Initialization** – Represents time spent loading your function and running [initialization code \(p. 32\)](#). This subsegment only appears for the first event processed by each instance of your function.
- **Invocation** – Represents the work done by your handler code. By instrumenting your code, you can extend this subsegment with additional subsegments.
- **Overhead** – Represents the work done by the Lambda runtime to prepare 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 Node.js](#) in the AWS X-Ray Developer Guide.

Sections

- [Enabling active tracing with the Lambda API \(p. 506\)](#)
- [Enabling active tracing with AWS CloudFormation \(p. 507\)](#)
- [Storing runtime dependencies in a layer \(p. 507\)](#)

Enabling active tracing with the Lambda API

To manage tracing configuration with the AWS CLI or AWS SDK, use the following API operations:

- [UpdateFunctionConfiguration \(p. 935\)](#)
- [GetFunctionConfiguration \(p. 816\)](#)

- [CreateFunction \(p. 762\)](#)

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 that is locked 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 enable active 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 functions code, package them in a [Lambda layer \(p. 101\)](#).

The following example shows an `AWS::Serverless::LayerVersion` resource that stores 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 only update library layer if you change your runtime dependencies. The function deployment package only contains your code. When you update your function code, upload time is much faster than if you include dependencies in the deployment package.

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 Python

You can run Python code in AWS Lambda. Lambda provides [runtimes \(p. 195\)](#) 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.

Note

End of support for the Python 2.7 runtime starts on July 15, 2021. For more information, see [the section called "Runtime support policy" \(p. 198\)](#).

Python runtimes

Name	Identifier	AWS SDK for Python	Operating system	
Python 3.8	python3.8	boto3-1.17.42 botocore-1.20.42	Amazon Linux 2	
Python 3.7	python3.7	boto3-1.17.42 botocore-1.20.42	Amazon Linux	
Python 3.6	python3.6	boto3-1.17.42 botocore-1.20.42	Amazon Linux	
Python 2.7	python2.7	boto3-1.17.42 botocore-1.20.42	Amazon Linux	

To create a Python function

1. Open the [Lambda console](#).
2. Choose **Create function**.
3. Configure the following settings:
 - **Name – my-function**.
 - **Runtime – Python 3.8**.
 - **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 **Create**.
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 \(p. 17\)](#). 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` file exports a function named `lambda_handler` that takes an event object and a context object. This is the [handler function \(p. 511\)](#) that Lambda calls when the function is invoked. The Python 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`.

Each time you save your function code, the Lambda console creates 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 \(p. 514\)](#) and updating your code at the command line.

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.

The function runtime passes a context object to the handler, in addition to the invocation event. The [context object \(p. 531\)](#) 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 \(p. 533\)](#) during invocation. If your function [returns an error \(p. 538\)](#), Lambda formats the error and returns it to the invoker.

Topics

- [Lambda function handler in Python \(p. 511\)](#)
- [Deploy Python Lambda functions with .zip file archives \(p. 514\)](#)
- [Deploy Python Lambda functions with container images \(p. 528\)](#)
- [AWS Lambda context object in Python \(p. 531\)](#)
- [AWS Lambda function logging in Python \(p. 533\)](#)
- [AWS Lambda function errors in Python \(p. 538\)](#)
- [Instrumenting Python code in AWS Lambda \(p. 542\)](#)

Lambda function handler in Python

Note

End of support for the Python 2.7 runtime starts on July 15, 2021. For more information, see the section called "Runtime support policy" (p. 198).

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
```

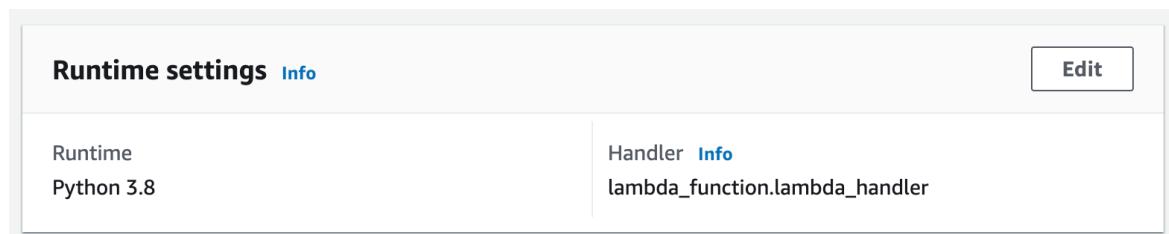
Naming

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`).

To change the function handler name in the Lambda console, on the **Runtime settings** pane, choose **Edit**.



How it works

When Lambda invokes your function handler, the [Lambda runtime \(p. 195\)](#) 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 \(p. 195\)](#) 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 \(p. 257\)](#).

- The second argument is the [context object \(p. 531\)](#). 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 \(p. 137\)](#) and the [Using AWS Lambda with other services \(p. 257\)](#). For example:

- If you use the `RequestResponse` invocation type, such as [Synchronous invocation \(p. 138\)](#), 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 an `Event` an [Asynchronous invocation \(p. 140\)](#) invocation type, the value is discarded.

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 \(p. 514\)](#) 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 \(p. 41\)](#).

Returning a message

The following example shows a function called `lambda_handler` that uses the `python3.8 Lambda runtime (p. 195)`. 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` that uses the `python3.8 Lambda runtime (p. 195)`. The function uses event data passed by Lambda at runtime. It parses the `environment variable (p. 77)` 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 \(p. 77\)](#).

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 API Response Syntax](#).

Returning a calculation

The following example [Lambda Python function code on GitHub](#) shows a function called `lambda_handler` that uses the `python3.6 Lambda runtime (p. 195)`. The function accepts user input and returns a calculation to the user.

You can use the following event data to [invoke the function](#):

```
{
    "action": "increment",
    "number": 3
}
```

Deploy Python Lambda functions with .zip file archives

Note

End of support for the Python 2.7 runtime starts on July 15, 2021. For more information, see [the section called "Runtime support policy" \(p. 198\)](#).

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 \(p. 195\)](#) 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*.

- Lambda uses POSIX file permissions, so you may need to [set permissions for the deployment package folder](#) before you create the .zip file archive.

This section describes how to create a .zip file as your deployment package, and then use the .zip file to deploy your function code to Lambda using the AWS Command Line Interface (AWS CLI).

Sections

- [Prerequisites \(p. 514\)](#)
- [What is a runtime dependency? \(p. 515\)](#)
- [Tutorial: Creating a Lambda function in Python 3.8 \(p. 515\)](#)
- [Updating a Lambda function in Python 3.8 \(p. 524\)](#)

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 CLI – Install version 2](#)
- [AWS CLI – Quick configuration with `aws configure`](#)

What is a runtime dependency?

A [deployment package](#) (p. 41) 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. 195) environment for your function's code.

The following example describes a Lambda function without runtime dependencies:

- If your function's code is in Python 3.8, 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 `python3.8` runtime.
- If your function's code depends only 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 the `python3.8` runtime.

For a complete list of AWS SDKs, see [Tools to Build on AWS](#).

Tutorial: Creating a Lambda function in Python 3.8

This tutorial guides you through building the code and assets to create a Lambda function in Python 3.8 using the AWS Command Line Interface (AWS CLI).

Sections

- [Prerequisites](#) (p. 515)
- [Creating a function without runtime dependencies](#) (p. 516)
- [Creating a function with runtime dependencies](#) (p. 520)

Prerequisites

This section describes the tools and resources required to complete the steps in the tutorial.

Install the 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 CLI – Install version 2](#)
- [AWS CLI – Quick configuration with `aws configure`](#)

Create an execution role

Your Lambda function's [execution role](#) (p. 47) is an AWS Identity and Access Management (IAM) role that grants your function permission to access AWS services and resources. Create an execution role in IAM with `Invoke` permission.

To create the execution role

1. Open a command prompt and use the `create-role` command to create an execution role named `lambda-ex`.

macOS/Linux

```
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"}]}'
```

Windows

```
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"}]}'
```

This command produces the following output. Save the value returned in Arn.

```
{
    "Role": {
        "Path": "/",
        "RoleName": "lambda-ex",
        "RoleId": "AROAWNZPPVHULXRJXQJD5",
        "Arn": "arn:aws:iam::your-account-id:role/lambda-ex",
        "CreateDate": "2021-01-05T18:00:30Z",
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {
                        "Service": "lambda.amazonaws.com"
                    },
                    "Action": "sts:AssumeRole"
                }
            ]
        }
    }
}
```

2. Use the [attach-role-policy](#) command to add `AWSLambdaBasicExecutionRole` permissions to the role.

```
$ aws iam attach-role-policy --role-name lambda-ex --policy-arn
arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

This command produces no output.

Creating a function without runtime dependencies

A dependency can be any package, module or other assembly dependency that is not included with the [Lambda runtime \(p. 195\)](#) environment for your function's code.

For more information, see [What is a runtime dependency? \(p. 515\)](#)

This section describes how to create a Lambda function without runtime dependencies.

Sections

- [Overview \(p. 517\)](#)
- [Create the deployment package \(p. 517\)](#)

- [Create the Lambda function \(p. 518\)](#)
- [Invoke the Lambda function \(p. 518\)](#)
- [What's next? \(p. 519\)](#)
- [Clean up your resources \(p. 519\)](#)

Overview

In this tutorial, you use the [sample code from the AWS SDK for Python \(Boto3\) project on GitHub](#) to create a Lambda function using the AWS CLI. You'll learn how to:

- Set up the directory structure of a deployment package (.zip file).
- Create a deployment package for a Lambda function without any runtime dependencies.
- Use the AWS CLI to upload the deployment package and create the Lambda function.
- Invoke the Lambda function to return a mathematical calculation.

The sample code contains standard math and logging Python libraries, which are used to return a calculation based on user input. Standard Python libraries are included with the [python3.8 runtime \(p. 195\)](#). Although the function's code doesn't depend on any other Python libraries and has no additional application dependencies, Lambda still requires a deployment package to create a function.

Create the deployment package

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.

Create the .zip file that Lambda uses as 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%)
```

Create the Lambda function

Lambda needs to know the [runtime \(p. 195\)](#) environment to use for your function's code, the [handler \(p. 511\)](#) in your function code, and the [execution role \(p. 47\)](#) that it can use to invoke your function.

Create the Lambda function using the execution role and deployment package that you created in the previous steps.

To create the function

1. Navigate to the my-math-function project directory.

```
cd my-math-function
```

2. Create a function named my-math-function. Substitute the value for role with the Arn you copied in previous steps.

```
aws lambda create-function --function-name my-math-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
```

This command produces the following output:

```
{  
    "FunctionName": "my-math-function",  
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-math-function",  
    "Runtime": "python3.8",  
    "Role": "arn:aws:iam::123456789012:role/lambda-ex",  
    "Handler": "lambda_function.lambda_handler",  
    "CodeSize": 753,  
    "Description": "",  
    "Timeout": 3,  
    "MemorySize": 128,  
    "LastModified": "2021-01-05T18:39:44.847+0000",  
    "CodeSha256": "82RtIE7p1ET5Od6bk4xSleJbUybUnZX52m92x/fEH84=",  
    "Version": "$LATEST",  
    "TracingConfig": {  
        "Mode": "PassThrough"  
    },  
    "RevisionId": "589e5115-f3c0-446c-bc62-4e05cf0a3c85",  
    "State": "Active",  
    "LastUpdateStatus": "Successful"  
}
```

The Lambda function in this step uses a function handler of `lambda_function.lambda_handler`. For more information about function handler naming conventions, see [Naming \(p. 511\)](#) in **AWS Lambda function handler in Python**.

Invoke the Lambda function

Invoke the Lambda function [synchronously \(p. 138\)](#) using the event input defined for the sample code. For more information, see [How it works \(p. 511\)](#) in **AWS Lambda function handler in Python**.

To invoke the function

- Use the `invoke` command.

```
aws lambda invoke \
```

```
--function-name my-math-function \
--cli-binary-format raw-in-base64-out \
--payload '{"action": "square","number": 3}' output.txt
```

The **cli-binary-format** option is required if you are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#).

This command produces the following output:

```
{
  "StatusCode": 200,
  "ExecutedVersion": "$LATEST"
}
```

For the `RequestResponse` invocation type, the status code is 200. For more information, see the [Invoke API reference](#).

You should see the following mathematical calculation in `output.txt`:

```
{"result": 9}
```

What's next?

- Learn how to update your Lambda function, see [Updating a Lambda function in Python 3.8 \(p. 524\)](#).
- Learn how to show logging events for your Lambda function, see [AWS Lambda function logging in Python \(p. 533\)](#).
- Explore other [AWS SDK examples in Python on GitHub](#).

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

- Use the [delete-function](#) command.

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

This command produces no output.

To delete the execution role policy

- Use the [delete-policy](#) command.

```
aws iam delete-role-policy --role-name lambda-ex --policy-name
AWSLambdaBasicExecutionRole
```

To delete the execution role

- Use the [delete-role](#) command.

```
aws iam delete-role --role-name lambda-ex
```

Creating a function with runtime dependencies

A dependency can be any package, module or other assembly dependency that is not included with the [Lambda runtime \(p. 195\)](#) environment for your function's code.

For more information, see [What is a runtime dependency? \(p. 515\)](#)

This section describes how to create a Lambda function with runtime dependencies.

Sections

- [Overview \(p. 520\)](#)
- [Create the deployment package \(p. 520\)](#)
- [Create the Lambda function \(p. 521\)](#)
- [Invoke the Lambda function \(p. 522\)](#)
- [What's next? \(p. 523\)](#)
- [Clean up your resources \(p. 523\)](#)

Overview

In this tutorial, you use sample code to create a Lambda function using the AWS CLI. The sample code uses the `requests` library to get the source code for <https://www.test.com/>. The `requests` library is not included with the `python3.8` [runtime \(p. 195\)](#), so you install it to a package directory.

You'll learn how to:

- Set up the directory structure of a deployment package (.zip file).
- Create a deployment package for a Lambda function with runtime dependencies.
- Use the AWS CLI to upload the deployment package and create the Lambda function.
- Invoke the Lambda function to return the source code.

Create the deployment package

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.

Create the .zip file that Lambda uses as 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 main(event, context):
    response = requests.get("https://www.test.com/")
    print(response.text)
    return response.text
if __name__ == "__main__":
    main('', '')
```

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 -g my-deployment-package.zip lambda_function.py
```

Create the Lambda function

Lambda needs to know the [runtime \(p. 195\)](#) environment to use for your function's code, the [handler \(p. 511\)](#) in your function code, and the [execution role \(p. 47\)](#) that it can use to invoke your function.

Create the Lambda function using the execution role and deployment package that you created in the previous steps.

To create the function

1. Navigate to the `my-sourcecode-function` project directory.

```
cd my-sourcecode-function
```

2. Create a function named `my-sourcecode-function`. Substitute the value for `role` with the Arn you copied in previous steps.

```
aws lambda create-function --function-name my-sourcecode-function --zip-file fileb://my-deployment-package.zip --handler lambda_function.main --runtime python3.8 --role arn:aws:iam::your-account-id:role/lambda-ex
```

This command produces the following output:

```
{  
    "FunctionName": "my-sourcecode-function",  
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-sourcecode-function",  
    "Runtime": "python3.8",  
    "Role": "arn:aws:iam::123456789012:role/lambda-ex",  
    "Handler": "lambda_function.main",  
    "CodeSize": 753,  
    "Description": "",  
    "Timeout": 3,  
    "MemorySize": 128,  
    "LastModified": "2021-01-11T18:39:44.847+0000",  
    "CodeSha256": "82RtIE7p1ET5Od6bk4xS1eJbUybUnZX52m92x/fEH84=",  
    "Version": "$LATEST",  
    "TracingConfig": {  
        "Mode": "PassThrough"  
    },  
    "RevisionId": "589e5115-f3c0-446c-bc62-4e05cf0a3c85",  
    "State": "Active",  
    "LastUpdateStatus": "Successful"  
}
```

The Lambda function in this step uses a function handler of `lambda_function.main`. For more information about function handler naming conventions, see [the section called “Naming” \(p. 511\)](#) in [AWS Lambda function handler in Python](#).

Invoke the Lambda function

Invoke the Lambda function [synchronously \(p. 138\)](#) using the event input defined for the sample code. For more information, see [How it works \(p. 511\)](#) in [AWS Lambda function handler in Python](#).

To invoke the function

- Use the `invoke` command.

```
aws lambda invoke \  
    --function-name my-sourcecode-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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#)

This command produces the following output:

```
{  
    "StatusCode": 200,  
    "ExecutedVersion": "$LATEST"
```

```
}
```

For the `RequestResponse` invocation type, the status code is 200. For more information, see the [Invoke API reference](#).

You should see the source code in `output.txt`.

What's next?

- Learn how to update your Lambda function, see [Updating a Lambda function in Python 3.8 \(p. 524\)](#).
- Learn how to show logging events for your Lambda function, see [AWS Lambda function logging in Python \(p. 533\)](#).
- Explore other [AWS SDK examples in Python on GitHub](#).

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

- Use the `delete-function` command.

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

This command produces no output.

To delete the execution role policy

- Use the `delete-policy` command.

```
aws iam delete-role-policy --role-name lambda-ex --policy-name
AWSLambdaBasicExecutionRole
```

To delete the execution role

- Use the `delete-role` command.

```
aws iam delete-role --role-name lambda-ex
```

Updating a Lambda function in Python 3.8

This tutorial guides you through the process of installing a Python library, creating a deployment package, and updating a Lambda function in Python 3.8 using the AWS Command Line Interface (AWS CLI).

The following steps assume that you have created a Lambda function and are updating the .zip file used as your deployment package. If you haven't created a function yet, see [the section called "Tutorial: Create function" \(p. 515\)](#).

Sections

- [Prerequisites \(p. 524\)](#)
- [Updating a function without runtime dependencies \(p. 524\)](#)
- [Updating a function with runtime dependencies \(p. 525\)](#)
- [Using a virtual environment \(p. 526\)](#)

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 CLI – Install version 2](#)
- [AWS CLI – Quick configuration with aws configure](#)

Updating a function without runtime dependencies

A dependency can be any package, module or other assembly dependency that is not included with the [Lambda runtime \(p. 195\)](#) environment for your function's code.

For more information, see [What is a runtime dependency? \(p. 515\)](#).

The following steps show how to create a deployment package that contains only your function code, and upload it to Lambda using the AWS CLI.

To update a Python function without runtime dependencies

1. Add function code files to the root of your deployment package.

```
~/my-function$ zip my-deployment-package.zip lambda_function.py
```

2. Use the [fileb://](#) prefix to upload the binary .zip file to Lambda and update the function code.

```
~/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": "python3.8",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Handler": "lambda_function.lambda_handler",
    "CodeSize": 815,
    "CodeSha256": "GcZ05oeHoJi61VpQj7vCLPs8DwCXmX5sE/fE2IHsizc=",
    "Version": "$LATEST",
    "RevisionId": "d1e983e3-ca8e-434b-8dc1-7add83d72ebd",
    ...
}
```

```
}
```

The Lambda function in this step uses a function handler of `lambda_function.lambda_handler`. For more information about function handler naming conventions, see [Naming \(p. 511\)](#) in [AWS Lambda function handler in Python](#).

Updating a function with runtime dependencies

A dependency can be any package, module or other assembly dependency that is not included with the [Lambda runtime \(p. 195\)](#) environment for your function's code.

For more information, see [What is a runtime dependency? \(p. 515\)](#).

The following steps show how to install the `requests` library, create a deployment package, and upload it to Lambda using the AWS CLI. The steps assume that you are not using a virtual environment. It also assumes that your function code uses Python 3.8 and the [python3.8 Lambda runtime \(p. 195\)](#).

Note

If you are creating a deployment package used in a layer, see [Including library dependencies in a layer \(p. 102\)](#).

To update a Python function with dependencies

1. Install libraries in a package directory with pip's `--target` option.

```
~/my-function$ pip install --target ./package requests
```

Note

To prevent `distutils` errors on [Debian-based systems](#) such as Ubuntu, you may need to pass the `--system` option.

2. Navigate to the package directory.

```
cd package
```

3. Create a deployment package with the installed libraries at the root.

```
~/my-function$ zip -r ../my-deployment-package.zip .
```

The last command saves the deployment package to the root of the `my-function` directory.

4. Navigate back to the `my-function` directory.

```
cd ..
```

5. Add function code files to the root of your deployment package.

```
~/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/  
...
```

6. Use the [update-function-code](#) command with the `fileb://` prefix to upload the binary .zip file to Lambda and update the function code.

```
~/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": "python3.8",  
    "Role": "arn:aws:iam::123456789012:role/lambda-role",  
    "Handler": "lambda_function.lambda_handler",  
    "CodeSize": 2269409,  
    "CodeSha256": "Gcz05oeHoJi61VpQj7vCLPs8DwCXmX5sE/fE2IHsizc=",  
    "Version": "$LATEST",  
    "RevisionId": "a9c05ffd-8ad6-4d22-b6cd-d34a00c1702c",  
    ...  
}
```

The Lambda function in this step uses a function handler of `lambda_function.lambda_handler`. For more information about function handler naming conventions, see [Naming \(p. 511\)](#) in [AWS Lambda function handler in Python](#).

Using a virtual environment

A dependency can be any package, module or other assembly dependency that is not included with the [Lambda runtime \(p. 195\)](#) environment for your function's code.

For more information, see [What is a runtime dependency? \(p. 515\)](#).

The following steps show how to install the [requests](#) library, create a deployment package, and upload it to Lambda using the AWS CLI. It also assumes that your function code uses Python 3.8 and the [python3.8 Lambda runtime \(p. 195\)](#).

Note

If you are creating a deployment package used in a layer, see [Including library dependencies in a layer \(p. 102\)](#).

To update a Python function with 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 myvenv/lib/python3.8/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/python3.8/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/  
...
```

6. Use the `fileb://` prefix to upload the binary .zip file to Lambda and update the function code.

```
~/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": "python3.8",  
    "Role": "arn:aws:iam::123456789012:role/lambda-role",  
    "Handler": "lambda_function.lambda_handler",  
    "CodeSize": 5912988,  
    "CodeSha256": "A2P0NUWq1J+LtSbkuP8tm9uNYqs1TAa3M76ptmZCw5g=",  
    "Version": "$LATEST",  
    "RevisionId": "5afdc7dc-2fcb-4ca8-8f24-947939ca707f",  
    ...  
}
```

The Lambda function in this step uses a function handler of `lambda_function.lambda_handler`. For more information about function handler naming conventions, see [Naming \(p. 511\)](#) in **AWS Lambda function handler in Python**.

Deploy Python Lambda functions with container images

Note

End of support for the Python 2.7 runtime starts on July 15, 2021. For more information, see [the section called "Runtime support policy" \(p. 198\)](#).

You can deploy your Lambda function code as a [container image \(p. 248\)](#). 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, add a runtime interface client to the base image to make it compatible with Lambda.

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 ECR container registry. See steps 7-9 in [Create image \(p. 249\)](#).
3. [Create \(p. 72\)](#) the Lambda function and deploy the image.

Topics

- [AWS base images for Python \(p. 528\)](#)
- [Python runtime interface clients \(p. 529\)](#)
- [Deploying Python with an AWS base image \(p. 529\)](#)
- [Create a Python image from an alternative base image \(p. 530\)](#)

AWS base images for Python

AWS provides the following base images for Python:

Tags	Runtime	Operating system	Dockerfile
3, 3.8	Python 3.8	Amazon Linux 2	Dockerfile for Python 3.8 on GitHub
3.7	Python 3.7	Amazon Linux 2018.03	Dockerfile for Python 3.7 on GitHub
3.6	Python 3.6	Amazon Linux 2018.03	Dockerfile for Python 3.6 on GitHub
2, 2.7	Python 2.7	Amazon Linux 2018.03	Dockerfile for Python 2.7 on GitHub

Docker Hub repository: [amazon/aws-lambda-python](#)

Amazon ECR repository: gallery.ecr.aws/lambda/python

Python runtime interface clients

Install the runtime interface client 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.

Deploying Python with 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.

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. 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.

```
FROM public.ecr.aws/lambda/python:3.8

COPY app.py  .
CMD [ "app.handler" ]
```

4. To create the container image, follow steps 4 through 7 in [Create an image from an AWS base image for Lambda \(p.\)](#).

Adding dependencies when you create a Python image

If your Lambda function depends on external Python libraries, modify the previous procedure as follows:

1. In your project directory, add a file named `requirements.txt`. List each required library as a separate line in this file.
2. Modify your Dockerfile to add the required libraries to the container image. The following example copies the requirements file and installs the required libraries into the `app` directory. Do not install the dependencies globally or in user space.

```
FROM public.ecr.aws/lambda/python:3.8

# Create function directory
WORKDIR /app

# Install the function's dependencies
# Copy file requirements.txt from your project folder and install
# the requirements in the app directory.
```

```
COPY requirements.txt .
RUN pip3 install -r requirements.txt

# Copy handler function (from the local app directory)
COPY app.py .

# Overwrite the command by providing a different command directly in the template.
CMD ["/app/app.handler"]
```

Create a Python image from an alternative base image

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.

AWS Lambda context object in Python

Note

End of support for the Python 2.7 runtime starts on July 15, 2021. For more information, see the section called “[Runtime support policy](#)” (p. 198).

When Lambda runs your function, it passes a context object to the [handler](#) (p. 511). 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. 511).

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. 95) 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_timeInMillis.
time.sleep(1)
print("Lambda time remaining in MS:", context.get_remaining_timeInMillis())
```

In addition to the options listed above, you can also use the AWS X-Ray SDK for [Instrumenting Python code in AWS Lambda \(p. 542\)](#) to identify critical code paths, trace their performance and capture the data for analysis.

AWS Lambda function logging in Python

Note

End of support for the Python 2.7 runtime starts on July 15, 2021. For more information, see the section called “[Runtime support policy](#)” (p. 198).

AWS Lambda automatically monitors Lambda functions on your behalf and sends function metrics 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.

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 \(p. 533\)](#)
- [Using the Lambda console \(p. 534\)](#)
- [Using the CloudWatch console \(p. 534\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 534\)](#)
- [Deleting logs \(p. 536\)](#)
- [Logging library \(p. 536\)](#)

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 environment variables and the event object.

Example `lambda_function.py`

```
import os

def lambda_handler(event, context):
    print('## ENVIRONMENT VARIABLES')
    print(os.environ)
    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',
         '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 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 Traceld** – For traced requests, the [AWS X-Ray trace ID \(p. 450\)](#).
- **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 \(p. 694\)](#).

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 \(p. 200\)](#). 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 \(p. 472\)](#).

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 CLI – Install 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":  
        "U1RBULQgUmVxdWVzdElkOia4N2QwNDRiOC1mMTU0LTEXZTgtOGNkYS0yOTc0YzVlNGZimjEgVmVyc2lvb...",  
    "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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#)

.

```
#!/bin/bash  
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --  
payload '{"key": "value"}' out  
sed -i'' -e 's/"/\'' out  
sleep 15  
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-name $(cat  
out) --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.

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)
    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',
    '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 Python

Note

End of support for the Python 2.7 runtime starts on July 15, 2021. For more information, see the section called "Runtime support policy" (p. 198).

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 \(p. 538\)](#)
- [Using the Lambda console \(p. 539\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 539\)](#)
- [Error handling in other AWS services \(p. 540\)](#)
- [Error examples \(p. 540\)](#)
- [Sample applications \(p. 541\)](#)
- [What's next? \(p. 503\)](#)

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 \(p. 195\)](#) 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 [Invoke API Errors \(p. 841\)](#).

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 \(p. 450\)](#) is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Test**.
4. Select **New event** and then choose an **Event template** from the dropdown list.
5. Enter a name for the test event.
6. Enter the JSON for the test event.
7. Choose **Create event**.
8. Choose **Invoke**.

The Lambda console invokes your function [synchronously \(p. 138\)](#) 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 CLI – Install 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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#).

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"]}
```

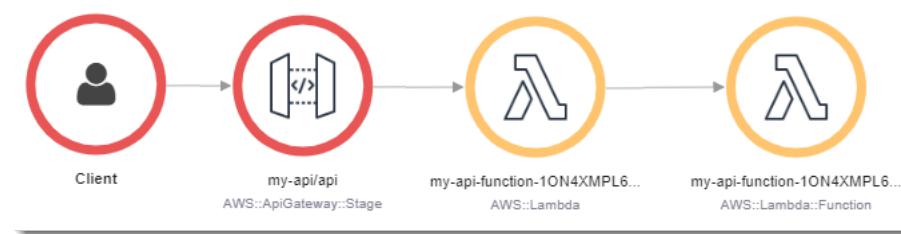
Lambda also records up to 256 KB of the error object in the function's logs. For more information, see [AWS Lambda function logging in Python \(p. 533\)](#).

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 \(p. 542\)](#).

Error examples

The following section shows common errors you may receive when creating, updating, or invoking your function using the Python [Lambda runtimes \(p. 195\)](#).

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 [file://](#) 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" \(p. 533\)](#) page.

Instrumenting Python code in AWS Lambda

Note

End of support for the Python 2.7 runtime starts on July 15, 2021. For more information, see the section called “[Runtime support policy](#)” (p. 198).

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.

The X-Ray *service map* shows the flow of requests through your application. The following example from the [error processor](#) (p. 472) sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon S3 and Amazon CloudWatch Logs.



To trace requests that don't have a tracing header, enable active tracing in your function's configuration.

To enable active tracing

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring tools**.
4. Choose **Edit**.
5. Under **X-Ray**, enable **Active tracing**.
6. Choose **Save**.

Pricing

X-Ray has a perpetual free tier. Beyond the free tier threshold, X-Ray charges for trace storage and retrieval. For details, see [AWS X-Ray pricing](#).

Your function needs permission to upload trace data to X-Ray. When you enable active tracing in the Lambda console, Lambda adds the required permissions to your function's [execution role](#) (p. 47). Otherwise, add the [AWSXRayDaemonWriteAccess](#) policy to the execution role.

X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of the requests that your application serves. The default sampling rule is 1 request per second and 5 percent of additional requests. This sampling rate cannot be configured for Lambda functions.

When active tracing is enabled, Lambda records a trace for a subset of invocations. Lambda records two *segments*, which creates two nodes on the service map. The first node represents the Lambda service that receives the invocation request. The second node is recorded by the function's [runtime \(p. 30\)](#).



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 Python. To get the SDK, add the `aws-xray-sdk` package to your application's dependencies.

Example [blank-python/function/requirements.txt](#)

```
jsonpickle==1.3
aws-xray-sdk==2.4.3
```

To instrument AWS SDK clients, patch the `boto3` library with the `aws_xray_sdk.core` module.

Example [blank-python/function/lambda_function.py](#) – Tracing an AWS SDK client

```
import boto3
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

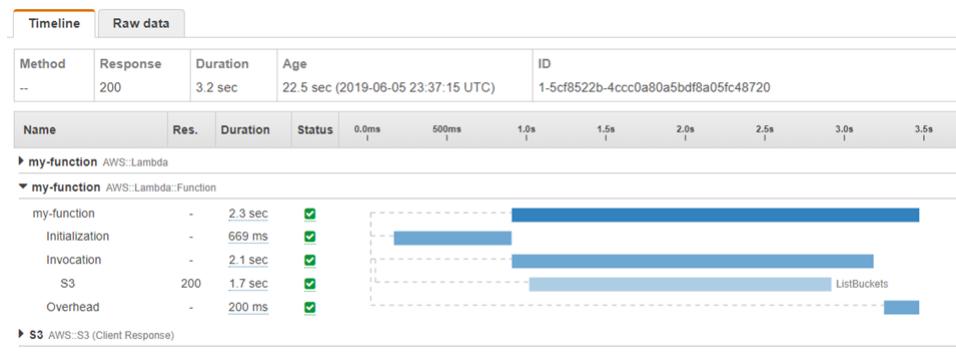
logger = logging.getLogger()
logger.setLevel(logging.INFO)
patch_all()

client = boto3.client('lambda')
client.get_account_settings()

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES\r' + jsonpickle.encode(dict(**os.environ)))
    ...

```

The following example shows a trace with 2 segments. Both are named **my-function**, but one is type `AWS::Lambda` and the other is `AWS::Lambda::Function`. The function segment is expanded to show its subsegments.



The first segment represents the invocation request processed by the Lambda service. The second segment records the work done by your function. The function segment has 3 subsegments.

- **Initialization** – Represents time spent loading your function and running [initialization code \(p. 32\)](#). This subsegment only appears for the first event processed by each instance of your function.
- **Invocation** – Represents the work done by your handler code. By instrumenting your code, you can extend this subsegment with additional subsegments.
- **Overhead** – Represents the work done by the Lambda runtime to prepare 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 Python](#) in the AWS X-Ray Developer Guide.

Sections

- [Enabling active tracing with the Lambda API \(p. 544\)](#)
- [Enabling active tracing with AWS CloudFormation \(p. 544\)](#)
- [Storing runtime dependencies in a layer \(p. 545\)](#)

Enabling active tracing with the Lambda API

To manage tracing configuration with the AWS CLI or AWS SDK, use the following API operations:

- [UpdateFunctionConfiguration \(p. 935\)](#)
- [GetFunctionConfiguration \(p. 816\)](#)
- [CreateFunction \(p. 762\)](#)

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 that is locked 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 enable active 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 functions code, package them in a [Lambda layer \(p. 101\)](#).

The following example shows an `AWS::Serverless::LayerVersion` resource that stores 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 only update library layer if you change your runtime dependencies. The function deployment package only contains your code. When you update your function code, upload time is much faster than if you include dependencies in the deployment package.

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 \(p. 195\)](#) 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 runtimes

Name	Identifier	SDK for Ruby	Operating system	
Ruby 2.7	<code>ruby2.7</code>	3.0.1	Amazon Linux 2	
Ruby 2.5	<code>ruby2.5</code>	3.0.1	Amazon Linux	

Note

For end of support information about Ruby 2.5, see [the section called “Runtime support policy” \(p. 198\)](#).

Lambda functions use an [execution role \(p. 47\)](#) 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 **Create**.
 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 \(p. 17\)](#). 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 \(p. 549\)](#) 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`.

Each time you save your function code, the Lambda console creates 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 \(p. 550\)](#) and updating your code at the command line.

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 \(p. 555\)](#) 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 \(p. 556\)](#) during invocation. If your function [returns an error \(p. 561\)](#), Lambda formats the error and returns it to the invoker.

Topics

- [AWS Lambda function handler in Ruby \(p. 549\)](#)
- [Deploy Ruby Lambda functions with .zip file archives \(p. 550\)](#)
- [Deploy Ruby Lambda functions with container images \(p. 553\)](#)
- [AWS Lambda context object in Ruby \(p. 555\)](#)
- [AWS Lambda function logging in Ruby \(p. 556\)](#)

- [AWS Lambda function errors in Ruby \(p. 561\)](#)
- [Instrumenting Ruby code in AWS Lambda \(p. 565\)](#)

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 \(p. 555\)](#) 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 [AWS Lambda execution environment \(p. 200\)](#).

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 \(p. 195\)](#) 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*.

- Lambda uses POSIX file permissions, so you may need to [set permissions for the deployment package folder](#) before you create the .zip file archive.

This section describes how to create a .zip file as your deployment package, and then use the .zip file to deploy your function code to Lambda using the AWS Command Line Interface (AWS CLI).

Sections

- [Prerequisites \(p. 550\)](#)
- [Tools and libraries \(p. 550\)](#)
- [Updating a function with no dependencies \(p. 551\)](#)
- [Updating a function with additional dependencies \(p. 551\)](#)

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 CLI – Install 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 \(p. 926\)](#) 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 install --path vendor/bundle
```

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  
...
```

The `--path` installs the gems in the project directory instead of the system location, and sets this as the default path for future installations. To later install gems globally, use the `--system` option.

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 \(p. 248\)](#). 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, add a runtime interface client to the base image to make it compatible with Lambda.

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 ECR container registry. See steps 7-9 in [Create image \(p. 249\)](#).
3. [Create \(p. 72\)](#) the Lambda function and deploy the image.

AWS base images for Ruby

AWS provides the following base images for Ruby:

Tags	Runtime	Operating system	Dockerfile
2, 2.7	Ruby 2.7	Amazon Linux 2	Dockerfile for Ruby 2.7 on GitHub
2.5	Ruby 2.5	Amazon Linux 2018.03	Dockerfile for Ruby 2.5 on GitHub

Docker Hub repository: [amazon/aws-lambda-ruby](#)

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*.

The instructions are also available on [AWS Lambda base images for Ruby](#) in the *Docker Hub 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.

AWS Lambda context object in Ruby

When Lambda runs your function, it passes a context object to the [handler \(p. 549\)](#). 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 \(p. 95\)](#) 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 function metrics 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.

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 \(p. 556\)](#)
- [Using the Lambda console \(p. 557\)](#)
- [Using the CloudWatch console \(p. 557\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 558\)](#)
- [Deleting logs \(p. 560\)](#)

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
```

For more detailed logs, use the [logger library](#).

```
# 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 timestamp, process ID, log level, and request ID.

```
I, [2019-10-26T10:04:01.689856 #8]  INFO 6573a3a0-2fb1-4e78-a582-2c769282e0bd -- : ## EVENT
I, [2019-10-26T10:04:01.689874 #8]  INFO 6573a3a0-2fb1-4e78-a582-2c769282e0bd -- :
 {"key1"=>"value1", "key2"=>"value2", "key3"=>"value3"}
```

Example log format

```
START RequestId: 50aba555-99c8-4b21-8358-644ee996a05f Version: $LATEST
## ENVIRONMENT VARIABLES
AWS_LAMBDA_FUNCTION_VERSION
$LATEST
AWS_LAMBDA_LOG_GROUP_NAME
/aws/lambda/my-function
AWS_LAMBDA_LOG_STREAM_NAME
2020/01/31/[ $LATEST]3f34xmpl069f4018b4a773bcfe8ed3f9
AWS_EXECUTION_ENV
AWS_Lambda_ruby2.5
...
## EVENT
key
value
END RequestId: 50aba555-xmpl-4b21-8358-644ee996a05f
REPORT RequestId: 50aba555-xmpl-4b21-8358-644ee996a05f Duration: 12.96 ms Billed Duration:
13 ms Memory Size: 128 MB Max Memory Used: 48 MB Init Duration: 117.86 ms
XRAY TraceId: 1-5e34a246-2a04xmpl0fa44eb60ea08c5f SegmentId: 454xmpl46ca1c7d3 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 TraceId** – For traced requests, the [AWS X-Ray trace ID \(p. 450\)](#).
- **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 \(p. 694\)](#).

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 \(p. 200\)](#). 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 \(p. 472\)](#).

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 CLI – Install 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":  
        "U1RBULQgUmVxdWVzdElkOiaA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",  
    "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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#).

```
#!/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
```

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 +x 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\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.

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 \(p. 561\)](#)
- [How it works \(p. 561\)](#)
- [Using the Lambda console \(p. 562\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 562\)](#)
- [Error handling in other AWS services \(p. 563\)](#)
- [Sample applications \(p. 564\)](#)
- [What's next? \(p. 564\)](#)

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 \(p. 195\)](#) 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 [Invoke API Errors \(p. 841\)](#).

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 \(p. 450\)](#) is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Test**.
4. Select **New event** and then choose an **Event template** from the dropdown list.
5. Enter a name for the test event.
6. Enter the JSON for the test event.
7. Choose **Create event**.
8. Choose **Invoke**.

The Lambda console invokes your function [synchronously \(p. 138\)](#) 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 CLI – Install 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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#)

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.

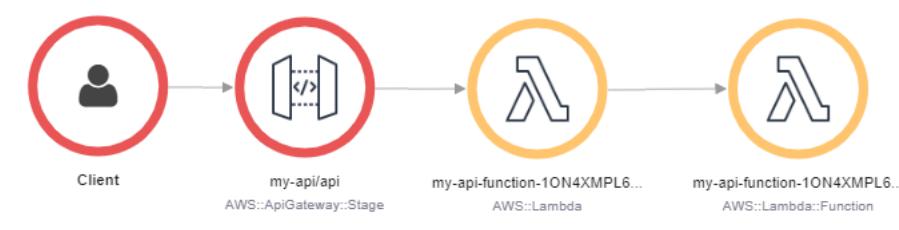
```
{"errorMessage": "no implicit conversion of String into
Integer", "errorType": "Function<TypeError>", "stackTrace": ["/var/task/function.rb:3:in
`first'", "/var/task/function.rb:3:in `handler'"]}
```

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 \(p. 565\)](#).

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" \(p. 556\)](#) 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.

The X-Ray *service map* shows the flow of requests through your application. The following example from the [error processor \(p. 472\)](#) sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon S3 and Amazon CloudWatch Logs.



To trace requests that don't have a tracing header, enable active tracing in your function's configuration.

To enable active tracing

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring tools**.
4. Choose **Edit**.
5. Under **X-Ray**, enable **Active tracing**.
6. Choose **Save**.

Pricing

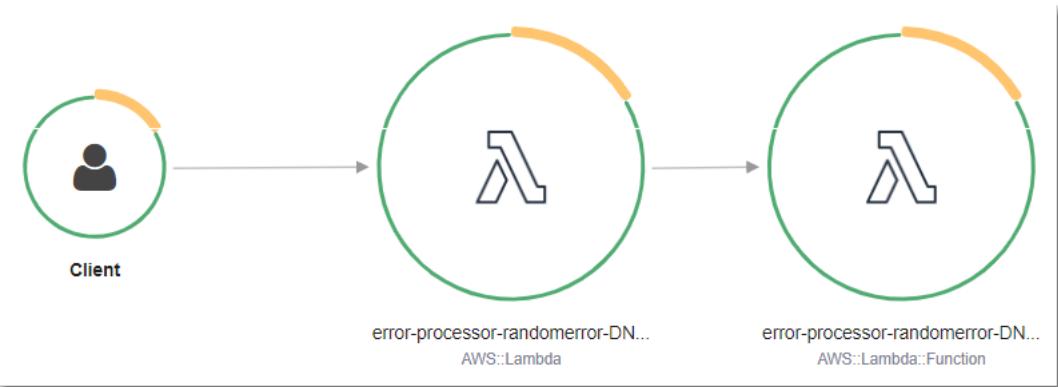
X-Ray has a perpetual free tier. Beyond the free tier threshold, X-Ray charges for trace storage and retrieval. For details, see [AWS X-Ray pricing](#).

Your function needs permission to upload trace data to X-Ray. When you enable active tracing in the Lambda console, Lambda adds the required permissions to your function's [execution role \(p. 47\)](#). Otherwise, add the [AWSXRayDaemonWriteAccess](#) policy to the execution role.

X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of the requests that your application serves. The default sampling rule is 1 request

per second and 5 percent of additional requests. This sampling rate cannot be configured for Lambda functions.

When active tracing is enabled, Lambda records a trace for a subset of invocations. Lambda records two *segments*, which creates two nodes on the service map. The first node represents the Lambda service that receives the invocation request. The second node is recorded by the function's [runtime \(p. 30\)](#).



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.

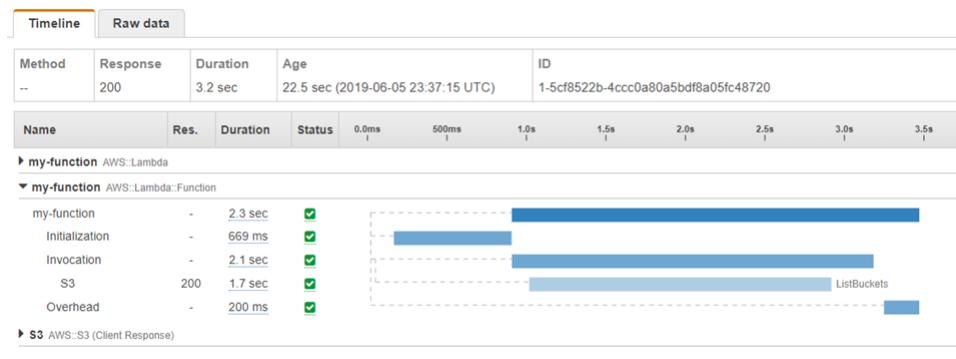
Example [blank-ruby/function/lambda_function.rb – Tracing an AWS SDK client](#)

```
# 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)
  ...
end
```

The following example shows a trace with 2 segments. Both are named **my-function**, but one is type `AWS::Lambda` and the other is `AWS::Lambda::Function`. The function segment is expanded to show its subsegments.



The first segment represents the invocation request processed by the Lambda service. The second segment records the work done by your function. The function segment has 3 subsegments.

- **Initialization** – Represents time spent loading your function and running [initialization code \(p. 32\)](#). This subsegment only appears for the first event processed by each instance of your function.
- **Invocation** – Represents the work done by your handler code. By instrumenting your code, you can extend this subsegment with additional subsegments.
- **Overhead** – Represents the work done by the Lambda runtime to prepare 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 \(p. 567\)](#)
- [Enabling active tracing with AWS CloudFormation \(p. 567\)](#)
- [Storing runtime dependencies in a layer \(p. 568\)](#)

Enabling active tracing with the Lambda API

To manage tracing configuration with the AWS CLI or AWS SDK, use the following API operations:

- [UpdateFunctionConfiguration \(p. 935\)](#)
- [GetFunctionConfiguration \(p. 816\)](#)
- [CreateFunction \(p. 762\)](#)

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 that is locked 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 enable active 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 functions code, package them in a [Lambda layer \(p. 101\)](#).

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 only update library layer if you change your runtime dependencies. The function deployment package only contains your code. When you update your function code, upload time is much faster than if you include dependencies in the deployment package.

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 \(p. 195\)](#) 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 runtimes

Name	Identifier	JDK	Operating system
Java 11	java11	amazon-corretto-11	Amazon Linux 2
Java 8	java8.al2	amazon-corretto-8	Amazon Linux 2
Java 8	java8	java-1.8.0-openjdk	Amazon Linux

Lambda functions use an [execution role \(p. 47\)](#) 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 11.**
 - **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 **Create**.
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 \(p. 571\)](#) 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 \(p. 573\)](#) 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 \(p. 577\)](#) 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 \(p. 586\)](#) 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 \(p. 589\)](#) during invocation. If your function [returns an error \(p. 596\)](#), Lambda formats the error and returns it to the invoker.

Topics

- [Java sample applications for AWS Lambda \(p. 571\)](#)
- [AWS Lambda function handler in Java \(p. 573\)](#)
- [Deploy Java Lambda functions with .zip or JAR file archives \(p. 577\)](#)
- [Deploy Java Lambda functions with container images \(p. 584\)](#)
- [AWS Lambda context object in Java \(p. 586\)](#)
- [AWS Lambda function logging in Java \(p. 589\)](#)
- [AWS Lambda function errors in Java \(p. 596\)](#)
- [Instrumenting Java code in AWS Lambda \(p. 601\)](#)
- [Creating a deployment package using Eclipse \(p. 606\)](#)

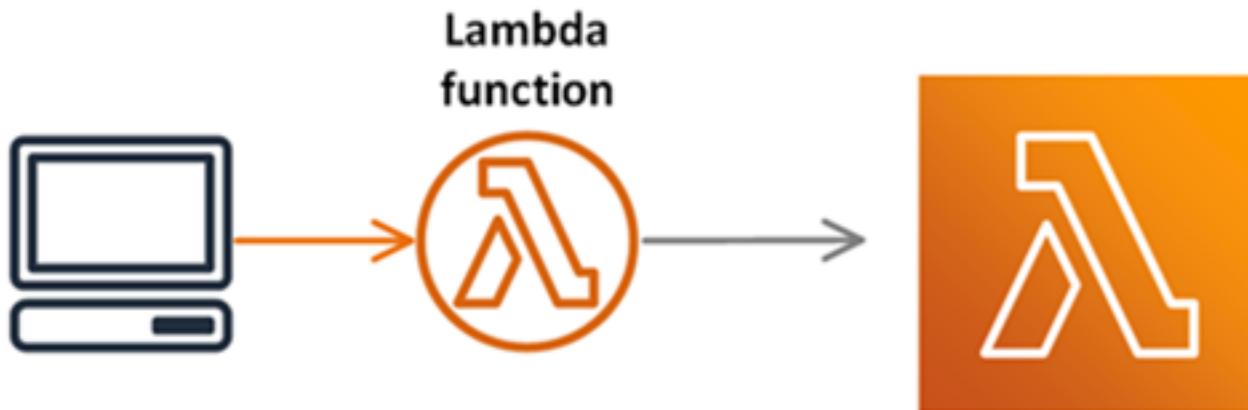
Java sample applications for AWS Lambda

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

- [blank-java](#) – A Java function that shows the use of Lambda's Java libraries, logging, environment variables, layers, AWS X-Ray tracing, unit tests, and the AWS SDK.
- [java-basic](#) – A minimal Java function with unit tests and variable logging configuration.
- [java-events](#) – A minimal Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that don't require the AWS SDK as a dependency, such as Amazon API Gateway.
- [java-events-v1sdk](#) – A Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that require the AWS SDK as a dependency (Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, and Amazon Kinesis).
- [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 the `blank-java` sample app to learn the basics, or as a starting point for your own application. It shows the use of Lambda's Java libraries, environment variables, the AWS SDK, and the AWS X-Ray SDK. It uses a Lambda layer to package its dependencies separately from the function code, which speeds up deployment times when you are iterating on your function code. The project requires minimal setup and can be deployed from the command line in less than a minute.



The other sample applications show other build configurations, handler interfaces, and use cases for services that integrate with Lambda. The `java-basic` sample shows a function with minimal dependencies. You can use this sample for cases where you don't need additional libraries like the AWS SDK, and can represent your function's input and output with standard Java types. To try a different handler type, you can simply change the handler setting on the function.

Example [java-basic/src/main/java/example/HandlerStream.java](#) – Stream handler

```
// 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,
        Charset.forName("US-ASCII")));
        PrintWriter writer = new PrintWriter(new BufferedWriter(new
        OutputStreamWriter(outputStream, Charset.forName("US-ASCII"))));
        try
        {
            HashMap<String, String> 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));
            ...
        }
    }
}
```

The `java-events` and `java-events-v1sdk` samples show the use of the event types provided by the `aws-lambda-java-events` library. These types represent the event documents that [AWS services \(p. 257\)](#) send to your function. `java-events` includes handlers for types that don't require additional dependencies. For event types like `DynamodbEvent` that require types from the AWS SDK for Java, `java-events-v1sdk` includes the SDK in its build configuration.

Example [java-events-v1sdk/src/main/java/example/HandlerDynamoDB.java](#) – DynamoDB records

```
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.amazonaws.services.dynamodbv2.model.Record;
...
// Handler value: example.HandlerDynamoDB
public class HandlerDynamoDB implements RequestHandler<DynamodbEvent, String>{
    private static final Logger logger = LoggerFactory.getLogger(HandlerDynamoDB.class);
    Gson gson = new GsonBuilder().setPrettyPrinting().create();
    @Override
    public String handleRequest(DynamodbEvent event, Context context)
    {
        String response = new String("200 OK");
        for (DynamodbStreamRecord record : event.getRecords()){
            logger.info(record.getEventID());
            logger.info(record.geteventName());
            logger.info(record.getDynamodb().toString());
        }
        ...
    }
}
```

For more highlights, see the other topics in this chapter.

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 \(p. 195\)](#) 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 \(p. 575\)](#). For example: `example.Handler`.

You can add [initialization code \(p. 32\)](#) 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, serializer, and AWS SDK client 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 Handler.java – Initialization code

```
// Handler value: example.Handler
```

```

public class Handler implements RequestHandler<SQSEvent, String>{
    private static final Logger logger = LoggerFactory.getLogger(Handler.class);
    private static final Gson gson = new GsonBuilder().setPrettyPrinting().create();
    private static final LambdaAsyncClient lambdaClient = LambdaAsyncClient.create();
    ...
    @Override
    public String handleRequest(SQSEvent event, Context context)
    {
        String response = new String();
        // call Lambda API
        logger.info("Getting account settings");
        CompletableFuture<GetAccountSettingsResponse> accountSettings =
            lambdaClient.getAccountSettings(GetAccountSettingsRequest.builder().build());
        // log execution details
        logger.info("ENVIRONMENT VARIABLES: " + gson.toJson(System.getenv()));
        ...
    }
}

```

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 \(p. 576\)](#).

Sections

- [Choosing input and output types \(p. 574\)](#)
- [Handler interfaces \(p. 575\)](#)
- [Sample handler code \(p. 576\)](#)

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 \(p. 577\)](#) 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,
        Charset.forName("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

- [blank-java](#) – A Java function that shows the use of Lambda's Java libraries, logging, environment variables, layers, AWS X-Ray tracing, unit tests, and the AWS SDK.
- [java-basic](#) – A minimal Java function with unit tests and variable logging configuration.
- [java-events](#) – A minimal Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that don't require the AWS SDK as a dependency, such as Amazon API Gateway.
- [java-events-v1sdk](#) – A Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that require the AWS SDK as a dependency (Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, and Amazon Kinesis).
- [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.

The `blank-java` 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 \(p. 577\)](#)
- [Tools and libraries \(p. 577\)](#)

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 CLI – Install 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 \(p. 589\)](#).

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:1.2.1'
    implementation 'com.amazonaws:aws-lambda-java-events:3.1.0'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.2.0'
}
```

Maven

```
<dependencies>
```

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-core</artifactId>
    <version>1.2.1</version>
</dependency>
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-events</artifactId>
    <version>3.1.0</version>
</dependency>
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-log4j2</artifactId>
    <version>1.2.0</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 \(p. 578\)](#). For Apache Maven, [use the Maven Shade plugin \(p. 579\)](#).

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 \(p. 101\)](#).

You can upload your deployment package by using the Lambda console, the Lambda API, or AWS Serverless Application Model (AWS SAM).

To upload a deployment package with the Lambda console

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Under **Code source**, choose **Upload from**.
4. Upload the deployment package.
5. Choose **Save**.

Sections

- [Building a deployment package with Gradle \(p. 578\)](#)
- [Building a deployment package with Maven \(p. 579\)](#)
- [Uploading a deployment package with the Lambda API \(p. 581\)](#)
- [Uploading a deployment package with AWS SAM \(p. 582\)](#)

Building a deployment package with Gradle

To create a deployment package with your function's code and dependencies, use the **Zip** build type.

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 platform('software.amazon.awssdk:bom:2.10.73')
    implementation 'software.amazon.awssdk:lambda'
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.1'
    implementation 'com.amazonaws:aws-lambda-java-events:3.1.0'
    implementation 'com.google.code.gson:gson:2.8.6'
    implementation 'org.apache.logging.log4j:log4j-api:2.13.0'
    implementation 'org.apache.logging.log4j:log4j-core:2.13.0'
    runtimeOnly 'org.apache.logging.log4j:log4j-slf4j18-impl:2.13.0'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.2.0'
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.6.0'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.6.0'
}
```

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:1.2.1 in the shaded jar.
[INFO] Including com.amazonaws:aws-lambda-java-events:jar:3.1.0 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.

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 API

To update a function's code with the AWS Command Line Interface (AWS CLI) or AWS SDK, use the [UpdateFunctionCode \(p. 926\)](#) 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 URI, 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

- [blank-java](#) – A Java function that shows the use of Lambda's Java libraries, logging, environment variables, layers, AWS X-Ray tracing, unit tests, and the AWS SDK.
- [java-basic](#) – A minimal Java function with unit tests and variable logging configuration.
- [java-events](#) – A minimal Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that don't require the AWS SDK as a dependency, such as Amazon API Gateway.

- **java-events-v1sdk** – A Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that require the AWS SDK as a dependency (Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, and Amazon Kinesis).
- **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.

Deploy Java Lambda functions with container images

You can deploy your Lambda function code as a [container image \(p. 248\)](#). AWS provides the following resources to help you build a container image for your Java 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, add a runtime interface client to the base image to make it compatible with Lambda.

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 ECR container registry. See steps 7-9 in [Create image \(p. 249\)](#).
3. [Create \(p. 72\)](#) the Lambda function and deploy the image.

AWS base images for Java

AWS provides the following base images for Java:

Tags	Runtime	Operating system	Dockerfile
11	Java 11 (Corretto)	Amazon Linux 2	Dockerfile for Java 11 on GitHub
8.al2	Java 8 (Corretto)	Amazon Linux 2	Dockerfile for Java 8.al2 on GitHub
8	Java 8 (OpenJDK)	Amazon Linux 2018.03	Dockerfile for Java 8 on GitHub

Docker Hub repository: [amazon/aws-lambda-java](#)

Amazon ECR repository: [gallery.ecr.aws/lambda/java](#)

Using a Java base image

For instructions on how to use a Java base image, choose the **usage** tab on [Lambda base images for Java](#) in the *Amazon ECR repository*.

The instructions are also available on [Lambda base images for Java](#) in the *Docker Hub repository*.

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>1.0.0</version>
</dependency>
```

For package details, see [Lambda RIC in Maven Central Repository](#).

You can also view the Java client source code in the [AWS Lambda Java Support Libraries](#) repository on GitHub.

After your container image resides in the Amazon ECR container registry, you can [create and run \(p. 72\)](#) the Lambda function.

AWS Lambda context object in Java

When Lambda runs your function, it passes a context object to the [handler \(p. 573\)](#). 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 \(p. 95\)](#) 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 \(p. 589\)](#) 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 \(p. 589\)](#).

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

- [blank-java](#) – A Java function that shows the use of Lambda's Java libraries, logging, environment variables, layers, AWS X-Ray tracing, unit tests, and the AWS SDK.
- [java-basic](#) – A minimal Java function with unit tests and variable logging configuration.
- [java-events](#) – A minimal Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that don't require the AWS SDK as a dependency, such as Amazon API Gateway.

- [java-events-v1sdk](#) – A Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that require the AWS SDK as a dependency (Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, and Amazon Kinesis).
- [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.

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 function metrics 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.

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 \(p. 589\)](#)
- [Using the Lambda console \(p. 590\)](#)
- [Using the CloudWatch console \(p. 590\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 591\)](#)
- [Deleting logs \(p. 593\)](#)
- [Advanced logging with Log4j 2 and SLF4J \(p. 593\)](#)
- [Sample logging code \(p. 595\)](#)

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 \(p. 577\)](#) 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 TracId** – For traced requests, the [AWS X-Ray trace ID \(p. 450\)](#).
- **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 \(p. 694\)](#).

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 \(p. 200\)](#). 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 \(p. 472\)](#).

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 CLI – Install 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":  
        "U1RBULQgUmVxdWVzdElkOia4N2QwNDRiOC1mMTU0LTEzTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",  
    "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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#).

```
#!/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
```

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 +x 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\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

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 \(p. 577\)](#) 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/Handler.java](#) – Logging with SLF4J

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

// Handler value: example.Handler
public class Handler implements RequestHandler<SQSEvent, String>{
    private static final Logger logger = LoggerFactory.getLogger(Handler.class);
    Gson gson = new GsonBuilder().setPrettyPrinting().create();
    LambdaAsyncClient lambdaClient = LambdaAsyncClient.create();
    @Override
    public String handleRequest(SQSEvent event, Context context)
    {
        String response = new String();
        // call Lambda API
        logger.info("Getting account settings");
        CompletableFuture<GetAccountSettingsResponse> accountSettings =
            lambdaClient.getAccountSettings(GetAccountSettingsRequest.builder().build());
        // log execution details
        logger.info("ENVIRONMENT VARIABLES: {}", gson.toJson(System.getenv()));
        ...
    }
}
```

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 platform('software.amazon.awssdk:bom:2.10.73')
    implementation platform('com.amazonaws:aws-xray-recorder-sdk-bom:2.4.0')
    implementation 'software.amazon.awssdk:lambda'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-core'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk-core'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk-v2'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk-v2-instrumentor'
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.1'
    implementation 'com.amazonaws:aws-lambda-java-events:3.1.0'
    implementation 'com.google.code.gson:gson:2.8.6'
    implementation 'org.apache.logging.log4j:log4j-api:2.13.0'
    implementation 'org.apache.logging.log4j:log4j-core:2.13.0'
```

```
    runtimeOnly 'org.apache.logging.log4j:log4j-slf4j18-impl:2.13.0'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.2.0'
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.6.0'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.6.0'
}
```

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

- [blank-java](#) – A Java function that shows the use of Lambda's Java libraries, logging, environment variables, layers, AWS X-Ray tracing, unit tests, and the AWS SDK.
- [java-basic](#) – A minimal Java function with unit tests and variable logging configuration.
- [java-events](#) – A minimal Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that don't require the AWS SDK as a dependency, such as Amazon API Gateway.
- [java-events-v1sdk](#) – A Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that require the AWS SDK as a dependency (Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, and Amazon Kinesis).
- [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.

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.

The `blank-java` sample application builds on the basic configuration with AWS SDK logging and the Lambda Log4j 2 appender. It uses Log4j 2 in Lambda with custom appender that adds the invocation request ID to each line.

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 \(p. 596\)](#)
- [How it works \(p. 597\)](#)
- [Creating a function that returns exceptions \(p. 597\)](#)
- [Using the Lambda console \(p. 598\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 599\)](#)
- [Error handling in other AWS services \(p. 599\)](#)
- [Sample applications \(p. 600\)](#)
- [What's next? \(p. 600\)](#)

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.java:  
          ...  
        ]  
      }  
    }  
  }  
}
```

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 \(p. 195\)](#) 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 [Invoke API Errors \(p. 841\)](#).

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 an array 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 contain 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. 575) 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(Hand
```

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. 450) is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Test**.
4. Select **New event** and then choose an **Event template** from the dropdown list.
5. Enter a name for the test event.
6. Enter the JSON for the test event.
7. Choose **Create event**.
8. Choose **Invoke**.

The Lambda console invokes your function [synchronously \(p. 138\)](#) 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 CLI – Install 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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#).

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 \(p. 589\)](#).

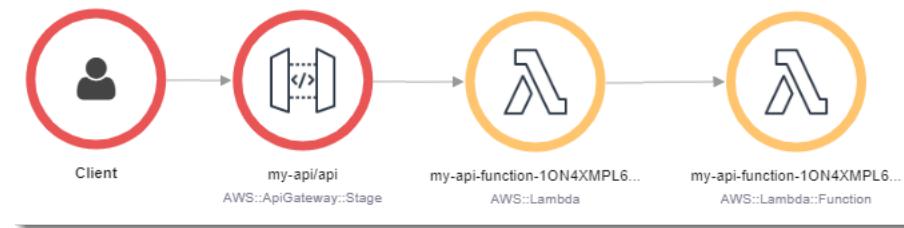
Error handling in other AWS services

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 \(p. 601\)](#).

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

- **blank-java** – A Java function that shows the use of Lambda's Java libraries, logging, environment variables, layers, AWS X-Ray tracing, unit tests, and the AWS SDK.
- **java-basic** – A minimal Java function with unit tests and variable logging configuration.
- **java-events** – A minimal Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that don't require the AWS SDK as a dependency, such as Amazon API Gateway.
- **java-events-v1sdk** – A Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that require the AWS SDK as a dependency (Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, and Amazon Kinesis).
- **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.

The `java-basic` function includes a handler (`HandlerDivide`) that returns a custom runtime exception. The `HandlerStream` handler implements the `RequestStreamHandler` and can throw an `IOException` checked exception.

What's next?

- Learn how to show logging events for your Lambda function on the [the section called "Logging" \(p. 589\)](#) page.

Instrumenting Java 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.

The X-Ray *service map* shows the flow of requests through your application. The following example from the [error processor \(p. 472\)](#) sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon S3 and Amazon CloudWatch Logs.



To trace requests that don't have a tracing header, enable active tracing in your function's configuration.

To enable active tracing

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring tools**.
4. Choose **Edit**.
5. Under **X-Ray**, enable **Active tracing**.
6. Choose **Save**.

Pricing

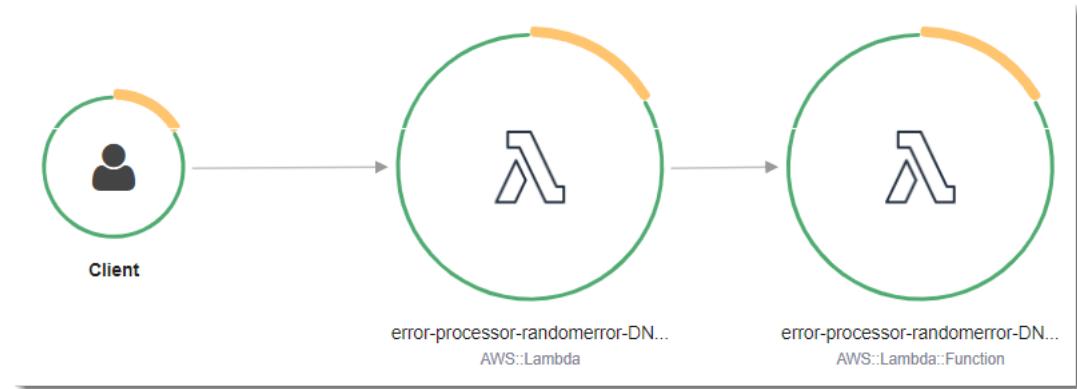
X-Ray has a perpetual free tier. Beyond the free tier threshold, X-Ray charges for trace storage and retrieval. For details, see [AWS X-Ray pricing](#).

Your function needs permission to upload trace data to X-Ray. When you enable active tracing in the Lambda console, Lambda adds the required permissions to your function's [execution role \(p. 47\)](#). Otherwise, add the [AWSXRayDaemonWriteAccess](#) policy to the execution role.

X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of the requests that your application serves. The default sampling rule is 1 request

per second and 5 percent of additional requests. This sampling rate cannot be configured for Lambda functions.

When active tracing is enabled, Lambda records a trace for a subset of invocations. Lambda records two *segments*, which creates two nodes on the service map. The first node represents the Lambda service that receives the invocation request. The second node is recorded by the function's [runtime \(p. 30\)](#).

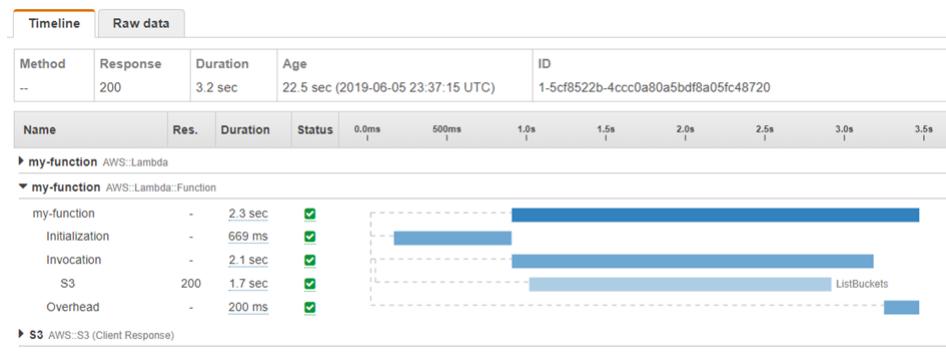


To record detail about calls that your function makes to other resources and services, add the X-Ray SDK for Java to your build configuration. The following example shows a Gradle build configuration that includes the libraries that enable automatic instrumentation of AWS SDK for Java 2.x clients.

Example `build.gradle` – Tracing dependencies

```
dependencies {
    implementation platform('software.amazon.awssdk:bom:2.10.73')
    implementation platform('com.amazonaws:aws-xray-recorder-sdk-bom:2.4.0')
    implementation 'software.amazon.awssdk:lambda'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-core'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk-core'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk-v2'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk-v2-instrumentor'
    ...
}
```

The following example shows a trace with 2 segments. Both are named **my-function**, but one is type `AWS::Lambda` and the other is `AWS::Lambda::Function`. The function segment is expanded to show its subsegments.



The first segment represents the invocation request processed by the Lambda service. The second segment records the work done by your function. The function segment has 3 subsegments.

- **Initialization** – Represents time spent loading your function and running [initialization code \(p. 32\)](#). This subsegment only appears for the first event processed by each instance of your function.
- **Invocation** – Represents the work done by your handler code. By instrumenting your code, you can extend this subsegment with additional subsegments.
- **Overhead** – Represents the work done by the Lambda runtime to prepare 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.

Sections

- [Enabling active tracing with the Lambda API \(p. 603\)](#)
- [Enabling active tracing with AWS CloudFormation \(p. 603\)](#)
- [Storing runtime dependencies in a layer \(p. 604\)](#)
- [Tracing in sample applications \(p. 604\)](#)

Enabling active tracing with the Lambda API

To manage tracing configuration with the AWS CLI or AWS SDK, use the following API operations:

- [UpdateFunctionConfiguration \(p. 935\)](#)
- [GetFunctionConfiguration \(p. 816\)](#)
- [CreateFunction \(p. 762\)](#)

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 that is locked 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 enable active 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 functions code, package them in a [Lambda layer \(p. 101\)](#).

The following example shows an `AWS::Serverless::LayerVersion` resource that stores the 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 only update library layer if you change your runtime dependencies. The function deployment package only contains your code. When you update your function code, upload time is much faster than if you include dependencies in the deployment package.

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.

Tracing in sample applications

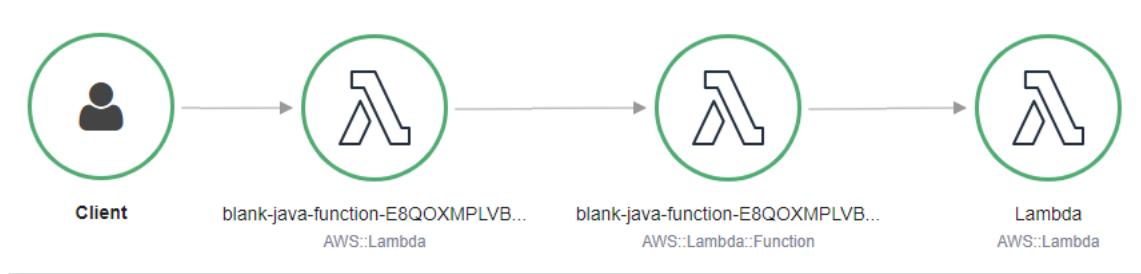
The GitHub repository for this guide includes sample applications that demonstrate the use of tracing. Each sample application includes scripts for easy deployment and cleanup, an AWS SAM template, and supporting resources.

Sample Lambda applications in Java

- [blank-java](#) – A Java function that shows the use of Lambda's Java libraries, logging, environment variables, layers, AWS X-Ray tracing, unit tests, and the AWS SDK.
- [java-basic](#) – A minimal Java function with unit tests and variable logging configuration.
- [java-events](#) – A minimal Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that don't require the AWS SDK as a dependency, such as Amazon API Gateway.
- [java-events-v1sdk](#) – A Java function that uses the [aws-lambda-java-events \(p. 577\)](#) library with event types that require the AWS SDK as a dependency (Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, and Amazon Kinesis).

- [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.

All of the sample applications have active tracing enabled for Lambda functions. The [blank-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.



This example from the [blank-java](#) sample application shows nodes for the Lambda service, a function, and the Lambda API. The function calls the Lambda API to monitor storage use in Lambda.

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 \(p. 606\)](#)
- [Create and build a project \(p. 606\)](#)

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: 1.2.1

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`.

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 runtimes

Name	Identifier	Operating system
Go 1.x	go1.x	Amazon Linux

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 \(p. 610\)](#).
- [github.com/aws/aws-lambda-go/lambdacontext](#): Helpers for accessing execution context information from the context object (p. 614).
- [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 \(p. 610\)](#)
- [AWS Lambda context object in Go \(p. 614\)](#)
- [Deploy Go Lambda functions with .zip file archives \(p. 616\)](#)
- [Deploy Go Lambda functions with container images \(p. 619\)](#)
- [AWS Lambda function logging in Go \(p. 623\)](#)
- [AWS Lambda function errors in Go \(p. 628\)](#)
- [Instrumenting Go code in AWS Lambda \(p. 632\)](#)
- [Using environment variables \(p. 636\)](#)

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 \(p. 614\)](#).
 - **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 \(p. 614\)](#).
 - **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 \(p. 628\)](#).
 - **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.

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 \(p. 628\)](#).

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 \(p. 610\)](#). 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 \(p. 95\)](#) 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 \(p. 616\)](#)
- [Tools and libraries \(p. 616\)](#)
- [Sample applications \(p. 616\)](#)
- [Creating a .zip file on macOS and Linux \(p. 617\)](#)
- [Creating a .zip file on Windows \(p. 617\)](#)

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 CLI – Install 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 \(p. 610\)](#).
- [github.com/aws/aws-lambda-go/lambdacontext](#): Helpers for accessing execution context information from the context object (p. 614).
- [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 go build main.go
```

Setting `GOOS` to `linux` ensures that the compiled executable is compatible with the [Go runtime \(p. 195\)](#), 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 go build main
```

4. (Optional) You may need to compile packages with `CGO_ENABLED=0` set on Linux:

```
GOOS=linux CGO_ENABLED=0 go build 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 function.zip main
```

Creating a .zip file on Windows

The following steps demonstrate how to download the [build-lambda-zip](#) tool for Windows from GitHub with `go get`, 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 **build-lambda-zip** tool from GitHub:

```
go.exe get -u github.com/aws/aws-lambda-go/cmd/build-lambda-zip
```

2. 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
go build -o main main.go
%USERPROFILE%\Go\bin\build-lambda-zip.exe -output main.zip main
```

PowerShell

In PowerShell, run the following:

```
$env:GOOS = "linux"
$env:CGO_ENABLED = "0"
$env:GOARCH = "amd64"
go build -o main main.go
~\Go\Bin\build-lambda-zip.exe -output main.zip main
```

Deploy Go Lambda functions with container images

You can deploy your Lambda function code as a [container image \(p. 248\)](#). AWS provides the following resources to help you build a container image for your Go 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, add a runtime interface client to the base image to make it compatible with Lambda.

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 ECR container registry. See steps 7-9 in [Create image \(p. 249\)](#).
3. [Create \(p. 72\)](#) the Lambda function and deploy the image.

Topics

- [AWS base images for Go \(p. 619\)](#)
- [Go runtime interface clients \(p. 619\)](#)
- [Using the Go:1.x base image \(p. 620\)](#)
- [Deploying Go with the provided.al2 base image \(p. 620\)](#)
- [Deploying Go with an alternative base image \(p. 621\)](#)

AWS base images for Go

AWS provides the following base image for Go:

Tags	Runtime	Operating system	Dockerfile
1	Go 1.x	Amazon Linux 2018.03	Dockerfile for Go 1.x on GitHub

Docker Hub repository: [amazon/aws-lambda-go](#)

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

For instructions on how to use the base image for Go:1.x, choose the [usage](#) tab on [Lambda base images for Go](#) in the *Amazon ECR repository*.

The instructions are also available on [Lambda base images for Go](#) in the *Docker Hub repository*.

Deploying Go with the provided.al2 base image

To build a container image for Go that runs on Amazon Linux 2, use the `provided.al2` 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.al2` base image also includes the runtime interface emulator.

To build and deploy a Go function with the `provided.al2` base image.

Note that the first three steps are identical whether you deploy your function as a .zip file archive or as a container image.

1. On your local machine, create a project directory for your new function.
2. From your project folder, run the following command to install the required Lambda Go libraries.

```
go get github.com/aws/aws-lambda-go
```

For a description of the Lambda Go libraries libraries, see [Building Lambda functions with Go \(p. 609\)](#).

3. Create your [Go handler code \(p. 610\)](#) and include the `aws-lambda-go/lambda` package.
4. Use a text editor to create a Dockerfile in your project directory. The following example Dockerfile uses the AWS `provided.al2` base image.

```
FROM public.ecr.aws/lambda/provided:al2 as build
# install compiler
RUN yum install -y golang
RUN go env -w GOPROXY=direct
# cache dependencies
ADD go.mod go.sum ./
RUN go mod download
# build
ADD ..
RUN go build -o /main
# copy artifacts to a clean image
FROM public.ecr.aws/lambda/provided:al2
COPY --from=build /main /main
ENTRYPOINT [ "/main" ]
```

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. Authenticate the Docker CLI to your Amazon ECR registry.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-
stdin 123456789012.dkr.ecr.us-east-1.amazonaws.com
```

7. Tag your image to match your repository name, and deploy the image to Amazon ECR using the `docker push` command.

```
docker tag hello-world:latest 123456789012.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
docker push 123456789012.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

Now that your container image resides in the Amazon ECR container registry, you can [create \(p. 72\)](#) the Lambda function and deploy the image.

Deploying Go with 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.

```
FROM alpine as build
# install build tools
RUN apk add go git
RUN go env -w GOPROXY=direct
# cache dependencies
ADD go.mod go.sum ./
RUN go mod download
# build
ADD .
RUN go build -o /main
# copy artifacts to a clean image
FROM alpine
COPY --from=build /main /main
ENTRYPOINT [ "/main" ]
```

The steps are the same as described for a `provided.al2` base image, 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" \(p. 254\)](#).

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 '{}'
```

This command invokes the function running in the container image and returns a response.

Now that your container image resides in the Amazon ECR container registry, you can you can [create \(p. 72\)](#) the Lambda function and deploy the image.

AWS Lambda function logging in Go

AWS Lambda automatically monitors Lambda functions on your behalf and sends function metrics 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.

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 \(p. 623\)](#)
- [Using the Lambda console \(p. 624\)](#)
- [Using the CloudWatch console \(p. 624\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 624\)](#)
- [Deleting logs \(p. 627\)](#)

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-9DV3XMP16XBC
```

```
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: 1-5e7d7595-212fxmpl9ee07c488419132 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 TracId** – For traced requests, the [AWS X-Ray trace ID \(p. 450\)](#).
- **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 \(p. 694\)](#).

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 \(p. 200\)](#). 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 \(p. 472\)](#).

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 CLI – Install 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":  
        "U1RBULQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTexZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",  
    "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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#)

.

```
#!/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
```

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\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.

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 \(p. 628\)](#)
- [How it works \(p. 628\)](#)
- [Using the Lambda console \(p. 629\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 629\)](#)
- [Error handling in other AWS services \(p. 630\)](#)
- [What's next? \(p. 631\)](#)

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 \(p. 195\)](#) 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 [Invoke API Errors \(p. 841\)](#).

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 \(p. 450\)](#) is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Test**.
4. Select **New event** and then choose an **Event template** from the dropdown list.
5. Enter a name for the test event.
6. Enter the JSON for the test event.
7. Choose **Create event**.
8. Choose **Invoke**.

The Lambda console invokes your function [synchronously \(p. 138\)](#) 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 CLI – Install 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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#).

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
```

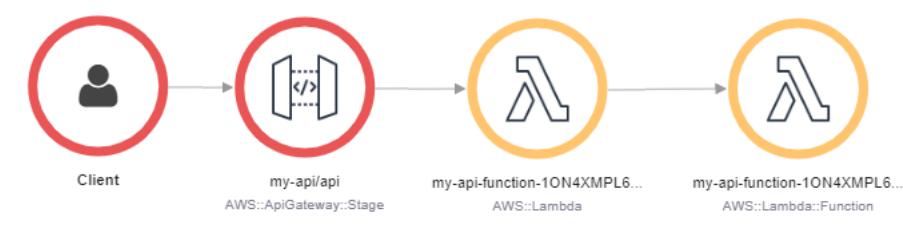
You should see the invocation response in your command prompt.

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 \(p. 632\)](#).

What's next?

- Learn how to show logging events for your Lambda function on the [the section called "Logging" \(p. 623\)](#) page.

Instrumenting Go 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.

The X-Ray *service map* shows the flow of requests through your application. The following example from the [error processor \(p. 472\)](#) sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon S3 and Amazon CloudWatch Logs.



To trace requests that don't have a tracing header, enable active tracing in your function's configuration.

To enable active tracing

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring tools**.
4. Choose **Edit**.
5. Under **X-Ray**, enable **Active tracing**.
6. Choose **Save**.

Pricing

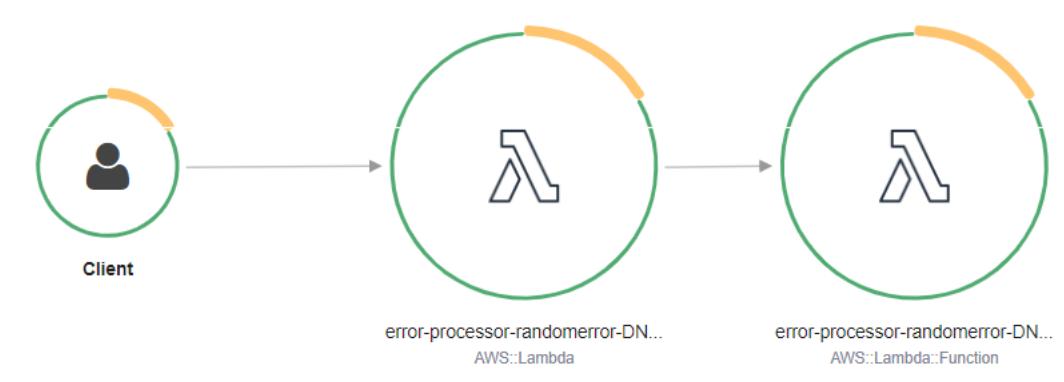
X-Ray has a perpetual free tier. Beyond the free tier threshold, X-Ray charges for trace storage and retrieval. For details, see [AWS X-Ray pricing](#).

Your function needs permission to upload trace data to X-Ray. When you enable active tracing in the Lambda console, Lambda adds the required permissions to your function's [execution role \(p. 47\)](#). Otherwise, add the [AWSXRayDaemonWriteAccess](#) policy to the execution role.

X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of the requests that your application serves. The default sampling rule is 1 request

per second and 5 percent of additional requests. This sampling rate cannot be configured for Lambda functions.

When active tracing is enabled, Lambda records a trace for a subset of invocations. Lambda records two *segments*, which creates two nodes on the service map. The first node represents the Lambda service that receives the invocation request. The second node is recorded by the function's [runtime \(p. 30\)](#).



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 Go. Download the SDK from its [GitHub repository](#) with go get:

```
go get github.com/aws/aws-xray-sdk-go
```

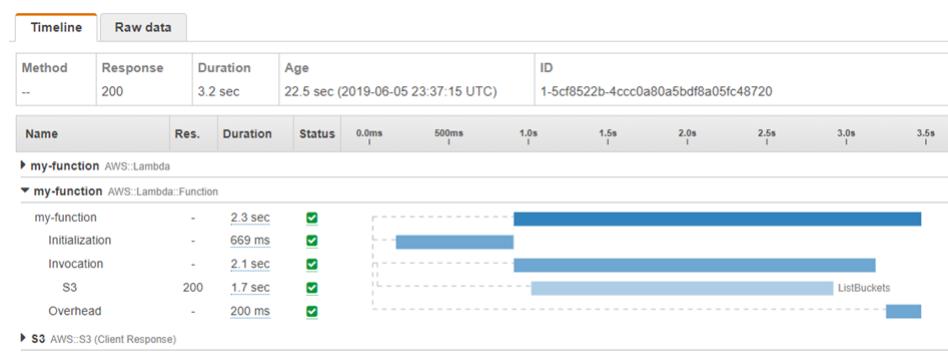
To instrument AWS SDK clients, pass the client to the `xray.AWS()` method.

```
xray.AWS(s3.Client)
```

Then you can trace your calls by using the `WithContext` version of the method.

```
svc.ListBucketsWithContext(ctx aws.Context, input *ListBucketsInput)
```

The following example shows a trace with 2 segments. Both are named **my-function**, but one is type `AWS::Lambda` and the other is `AWS::Lambda::Function`. The function segment is expanded to show its subsegments.



The first segment represents the invocation request processed by the Lambda service. The second segment records the work done by your function. The function segment has 3 subsegments.

- **Initialization** – Represents time spent loading your function and running [initialization code \(p. 32\)](#). This subsegment only appears for the first event processed by each instance of your function.

- **Invocation** – Represents the work done by your handler code. By instrumenting your code, you can extend this subsegment with additional subsegments.
- **Overhead** – Represents the work done by the Lambda runtime to prepare 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 Go](#) in the AWS X-Ray Developer Guide.

Sections

- [Enabling active tracing with the Lambda API \(p. 634\)](#)
- [Enabling active tracing with AWS CloudFormation \(p. 634\)](#)

Enabling active tracing with the Lambda API

To manage tracing configuration with the AWS CLI or AWS SDK, use the following API operations:

- [UpdateFunctionConfiguration \(p. 935\)](#)
- [GetFunctionConfiguration \(p. 816\)](#)
- [CreateFunction \(p. 762\)](#)

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 that is locked 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 enable active 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

```



Using environment variables

To access [environment variables \(p. 77\)](#) 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 %. 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 \(p. 80\)](#).

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# \(p. 651\)](#)) 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 runtimes

Name	Identifier	Operating system	
.NET Core 3.1	dotnetcore3.1	Amazon Linux 2	
.NET Core 2.1	dotnetcore2.1	Amazon Linux	

Note

For end of support information about .NET Core 2.1, see [the section called “Runtime support policy” \(p. 198\)](#).

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

- [AWS Lambda function handler in C# \(p. 639\)](#)
- [Deploy C# Lambda functions with .zip file archives \(p. 644\)](#)

- [Deploy .NET Lambda functions with container images \(p. 650\)](#)
- [AWS Lambda context object in C# \(p. 651\)](#)
- [AWS Lambda function logging in C# \(p. 652\)](#)
- [AWS Lambda function errors in C# \(p. 657\)](#)
- [Instrumenting C# code in AWS Lambda \(p. 662\)](#)

AWS 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. If you want access to the Lambda context object, it is available by defining a method parameter of type *ILambdaContext*, an interface 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.

```
returnType handler-name(inputType input, ILambdaContext context) {  
    ...  
}
```

In the syntax, note the following:

- ***inputType*** – The first handler parameter is the input to the handler, which can be event data (published by an event source) 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 will be 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 AWS Lambda with event sources such as Amazon S3 or 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 \(p. 651\)](#) which has information about the function and request.

Handling streams

Only the `System.IO.Stream` type is supported as an input parameter by default.

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`), which can be event data (published by an event source such as Amazon S3) or custom input 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 other types, as listed below, 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 the return-type will be ignored by Lambda. The return type may be set to void in such cases.
- 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 AWS Lambda \(p. 642\)](#).

Unless your function input and output parameters are of type `System.IO.Stream`, you will need to serialize them. AWS Lambda provides a default serializer 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 \(p. 644\)](#).

To add the default serializer attribute to a method, first add a dependency on `Amazon.Lambda.Serialization.Json` in your `.csproj` file.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AssemblyName>AssemblyName</AssemblyName>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="1.2.0" />
    <PackageReference Include="Amazon.Lambda.APIGatewayEvents" Version="2.3.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.Json" Version="1.8.0" />
    <PackageReference Include="Newtonsoft.Json" Version="12.0.1" />
  </ItemGroup>
</Project>
```

The example below illustrates the flexibility you can leverage by specifying the default Json.NET serializer on one method and another of your choosing on a different method:

```
public class ProductService{
  [LambdaSerializer(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]
  public Product DescribeProduct(DescribeProductRequest request)
  {
    return catalogService.DescribeProduct(request.Id);
  }

  [LambdaSerializer(typeof(MyJsonSerializer))]
  public Customer DescribeCustomer(DescribeCustomerRequest request)
  {
    return customerService.DescribeCustomer(request.Id);
  }
}
```

Note

If you are using .NET Core 3.1, we recommend that you use the [Amazon.Lambda.Serialization.SystemTextJson](#) serializer. This package provides a performance improvement over `Amazon.Lambda.Serialization.Json`.

Handler signatures

When creating Lambda functions, you have to provide a handler string that tells AWS 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 `.csproj`, the **ASSEMBLY** name will be the `.csproj` file name. For more information, see [.NET Core CLI \(p. 644\)](#). In this case, let's assume 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 will be of this format: **Assembly::Namespace.ClassName::MethodName**

Again, 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 Lambda should invoke. AWS Lambda will reject an otherwise valid signature if the resolution would require selecting among multiple (overloaded) signatures.

Serializing Lambda functions

For any Lambda functions that use input or output types other than a `Stream` object, you will need to add a serialization library to your application. You can do this in the following ways:

- Use the `Amazon.Lambda.Serialization.Json` NuGet package. This library uses JSON.NET to handle serialization.

Note

If you are using .NET Core 3.1, we recommend that you use the [Amazon.Lambda.Serialization.SystemTextJson](#) serializer. This package provides a performance improvement over `Amazon.Lambda.Serialization.Json`.

- 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 is returned by the `Invoke` API.

You use whichever serializer you wish by adding it as a dependency to your `MyProject.csproj` file.

```
...
<ItemGroup>
  <PackageReference Include="Amazon.Lambda.Core" Version="1.0.0" />
  <PackageReference Include="Amazon.Lambda.Serialization.Json" Version="1.3.0" />
</ItemGroup>
```

You then add it to your `AssemblyInfo.cs` file. For example, if you are using the default `Json.NET` serializer, this is what you would add:

```
[assembly:LambdaSerializer(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]
```

Note

You can define a custom serialization attribute at the method level, which will override the default serializer specified at the assembly level. For more information, see [Handling standard data types \(p. 640\)](#).

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 `unsafe` context can be used inside the handler method and its dependencies. For more information, see [unsafe \(C# reference\)](#).
- It may not pass a variable number of parameters using the `params` keyword, or use `ArgIterator` as an input or return parameter which is used to support variable number of parameters.
- The handler may not be a generic method (e.g. `IList<T> Sort<T>(IList<T> input)`).
- Async handlers with signature `async void` are not supported.

Using `async` in C# functions with AWS Lambda

If you know 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 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 \(p. 70\)](#).

```
public async Task<Response> ProcessS3ImageResizeAsync(SimpleS3Event input)
{
    var response = await client.DoAsyncWork(input);
    return response;
}
```

If you use this pattern, there are some considerations you must take into account:

- AWS 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 example below, 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 only waiting for 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 \(p. 644\)](#) or the [AWS Toolkit for Visual Studio \(p. 647\)](#). 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 \(p. 644\)](#)
- [AWS Toolkit for Visual Studio \(p. 647\)](#)

.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 and navigate to where you installed the .NET Core runtime and 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                            mstest         [C#], F#, VB
Common/Library                           xunit          [C#], F#, VB
Unit Test Project                        ...
Test/MSTest                             ...
xUnit Test Project                       ...
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",
"framework" : "netcoreapp2.1",
"function-runtime": "dotnetcore3.1",
"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 \(p. 641\)](#). 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.Json.JsonSerializer))]

namespace MyFunction
{
    public class Function
    {

        public string FunctionHandler1(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>netcoreapp2.1</TargetFramework>
</PropertyGroup>

<ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="1.0.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.Json" Version="1.3.0" />
</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, allowing you to seamlessly integrate any Lambda templates required to test your function.

```
<Project Sdk="Microsoft.NET.Sdk">
    ...
    <PackageReference Include="Amazon.Lambda.Core" Version="1.0.0" />
    ...
```

- **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:

The **cli-binary-format** option is required if you are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#).

```
dotnet lambda invoke-function MyFunction --cli-binary-format raw-in-base64-out --payload "Just Checking If Everything is OK"
```

If everything is successful, you see the following:

```
dotnet lambda invoke-function MyFunction --cli-binary-format raw-in-base64-out --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

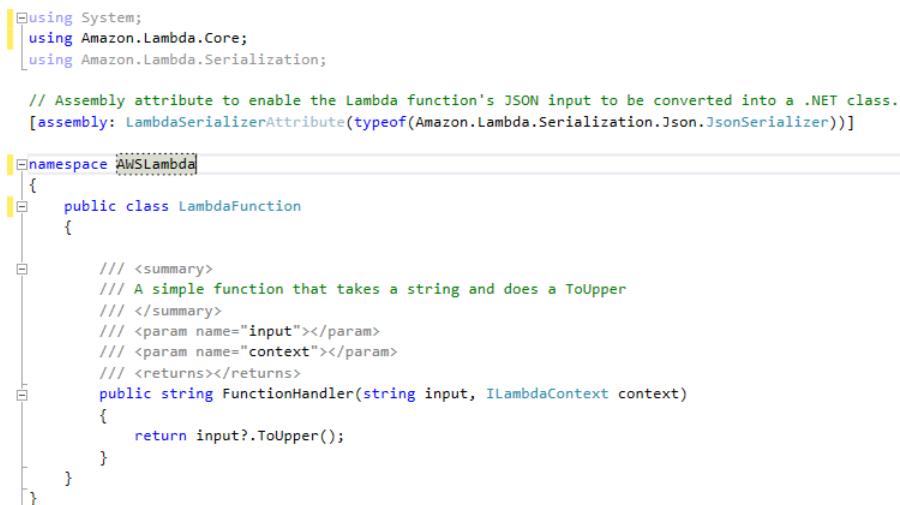
You can build .NET-based Lambda applications using the Lambda plugin to the [AWS Toolkit for Visual Studio](#). The toolkit is available as a [Visual Studio extension](#).

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, you can select from a list of sample applications that provide you with sample code to 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** – This is where the `function_handler` is specified, which 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",
    "framework" : "netcoreapp2.1",
    "function-runtime": "dotnetcore3.1",
    "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.



```
using System;
using Amazon.Lambda.Core;
using Amazon.Lambda.Serialization;

// Assembly attribute to enable the Lambda function's JSON input to be converted into a .NET class.
[Assembly: LambdaSerializerAttribute(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]

namespace AWSLambda
{
    public class LambdaFunction
    {
        /// <summary>
        /// A simple function that takes a string and does a ToUpper
        /// </summary>
        /// <param name="input"></param>
        /// <param name="context"></param>
        /// <returns></returns>
        public string FunctionHandler(string input, ILambdaContext context)
        {
            return input?.ToUpper();
        }
    }
}
```

4. Once you have written 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. 47) 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. 77).
 - **Memory** – The amount of memory available to the function at runtime. Choose an amount [between 128 MB and 10,240 MB](#) (p. 44) 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. 111).
 - **DLQ** – If your function is invoked asynchronously, [choose a dead-letter queue](#) (p. 145) to receive failed invocations.

- **Enable active tracing** – Sample incoming requests and [trace sampled requests with AWS X-Ray \(p. 450\)](#).
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 \(p. 248\)](#). AWS provides the following resources to help you build a container image for your .NET 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, add a runtime interface client to the base image to make it compatible with Lambda.

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 ECR container registry. See steps 7-9 in [Create image \(p. 249\)](#).
3. [Create \(p. 72\)](#) the Lambda function and deploy the image.

AWS base images for .NET

AWS provides the following base images for .NET:

Tags	Runtime	Operating system	Dockerfile
5.0	.NET 5.0	Amazon Linux 2	Dockerfile for .NET 5.0 on GitHub
core3.1	.NET Core 3.1	Amazon Linux 2	Dockerfile for .NET 3.1 on GitHub
core2.1	.NET Core 2.1	Amazon Linux 2018.03	Dockerfile for .NET 2.1 on GitHub

Docker Hub repository: `amazon/aws-lambda-dotnet`

Amazon ECR repository: `gallery.ecr.aws/lambda/dotnet`

Using a .NET base image

For instructions on how to use a .NET base image, choose the **usage** tab on [AWS Lambda base images for .NET](#) in the *Amazon ECR repository*.

The instructions are also available on [Lambda base images for .NET](#) in the *Docker Hub repository*.

.NET runtime interface clients

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 \(p. 639\)](#). 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 \(p. 95\)](#) 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 \(p. 652\)](#) 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);
}
```

AWS Lambda function logging in C#

AWS Lambda automatically monitors Lambda functions on your behalf and sends function metrics 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.

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 \(p. 652\)](#)
- [Using the Lambda console \(p. 653\)](#)
- [Using the CloudWatch console \(p. 653\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 654\)](#)
- [Deleting logs \(p. 656\)](#)

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 \(p. 637\)](#) 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: $LATEST
ENVIRONMENT VARIABLES:
{
    "AWS_EXECUTION_ENV": "AWS_Lambda_dotnetcore2.1",
    "AWS_LAMBDA_FUNCTION_MEMORY_SIZE": "256",
    "AWS_LAMBDA_LOG_GROUP_NAME": "/aws/lambda/blank-csharp-function-WU56XMPLV2XA",
    "AWS_LAMBDA_FUNCTION_VERSION": "$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": "1523232000001"
      },
      ...
    }
  ]
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 TracId** – For traced requests, the [AWS X-Ray trace ID \(p. 450\)](#).
- **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 \(p. 694\)](#).

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 \(p. 200\)](#). 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 \(p. 472\)](#).

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 CLI – Install 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":  
        "U1RBULQgUmVxdWVzdElkOia4N2QwNDRiOC1mMTU0LTEzTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",  
    "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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#)

```
#!/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
```

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\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 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 \(p. 657\)](#)
- [How it works \(p. 659\)](#)
- [Using the Lambda console \(p. 660\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 660\)](#)
- [Error handling in other AWS services \(p. 661\)](#)
- [What's next? \(p. 661\)](#)

Syntax

In the initialization phase, exceptions can be thrown for invalid handler strings, a rule-breaking type or method (see [Lambda function handler restrictions \(p. 642\)](#)), 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": "ArithmeticException",  
      "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.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 \(p. 195\)](#) 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 [Invoke API Errors \(p. 841\)](#).

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 \(p. 450\)](#) is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Test**.
4. Select **New event** and then choose an **Event template** from the dropdown list.
5. Enter a name for the test event.
6. Enter the JSON for the test event.
7. Choose **Create event**.
8. Choose **Invoke**.

The Lambda console invokes your function [synchronously \(p. 138\)](#) 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 CLI – Install 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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#)

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.

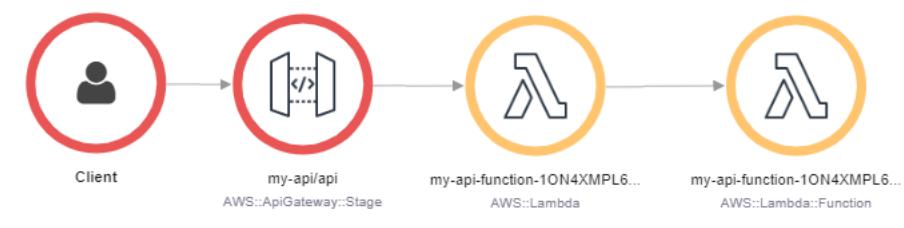
Lambda also records up to 256 KB of the error object in the function's logs. For more information, see [AWS Lambda function logging in C# \(p. 652\)](#).

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 \(p. 662\)](#).

What's next?

- Learn how to show logging events for your Lambda function on the [the section called "Logging" \(p. 652\)](#) page.

Instrumenting C# 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.

The X-Ray *service map* shows the flow of requests through your application. The following example from the [error processor \(p. 472\)](#) sample application shows an application with two functions. The primary function processes events and sometimes returns errors. The second function processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon S3 and Amazon CloudWatch Logs.



To trace requests that don't have a tracing header, enable active tracing in your function's configuration.

To enable active tracing

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring tools**.
4. Choose **Edit**.
5. Under **X-Ray**, enable **Active tracing**.
6. Choose **Save**.

Pricing

X-Ray has a perpetual free tier. Beyond the free tier threshold, X-Ray charges for trace storage and retrieval. For details, see [AWS X-Ray pricing](#).

Your function needs permission to upload trace data to X-Ray. When you enable active tracing in the Lambda console, Lambda adds the required permissions to your function's [execution role \(p. 47\)](#). Otherwise, add the [AWSXRayDaemonWriteAccess](#) policy to the execution role.

X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of the requests that your application serves. The default sampling rule is 1 request

per second and 5 percent of additional requests. This sampling rate cannot be configured for Lambda functions.

When active tracing is enabled, Lambda records a trace for a subset of invocations. Lambda records two *segments*, which creates two nodes on the service map. The first node represents the Lambda service that receives the invocation request. The second node is recorded by the function's [runtime \(p. 30\)](#).



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](#)

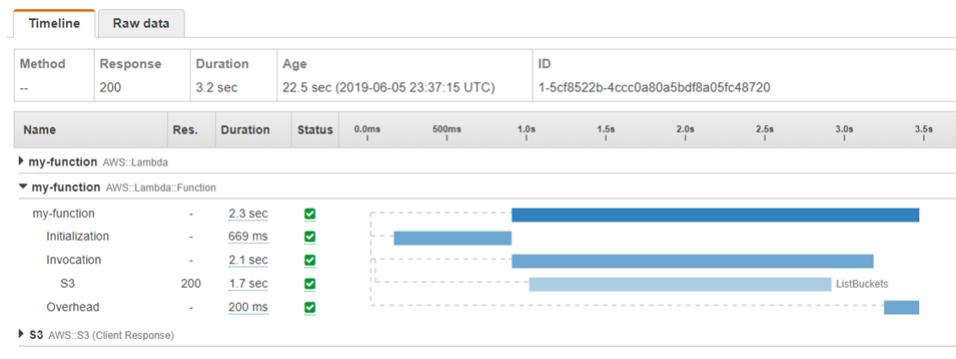
```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="12.0.3" />
    <PackageReference Include="Amazon.Lambda.Core" Version="1.1.0" />
    <PackageReference Include="Amazon.Lambda.SQSEvents" Version="1.1.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.Json" Version="1.7.0" />
    <PackageReference Include="AWSSDK.Core" Version="3.3.104.38" />
    <PackageReference Include="AWSSDK.Lambda" Version="3.3.108.11" />
    <PackageReference Include="AWSXRayRecorder.Core" Version="2.6.2" />
    <PackageReference Include="AWSXRayRecorder.Handlers.AwsSdk" Version="2.7.2" />
  </ItemGroup>
</Project>
```

To instrument AWS SDK clients, call the `RegisterXRayForAllServices` method in your initialization code.

Example [src/blank-csharp/Function.cs – Initialize X-Ray](#)

```
static async void initialize() {
    AWSSDKHandler.RegisterXRayForAllServices();
    lambdaClient = new AmazonLambdaClient();
    await callLambda();
}
```

The following example shows a trace with 2 segments. Both are named **my-function**, but one is type `AWS::Lambda` and the other is `AWS::Lambda::Function`. The function segment is expanded to show its subsegments.



The first segment represents the invocation request processed by the Lambda service. The second segment records the work done by your function. The function segment has 3 subsegments.

- **Initialization** – Represents time spent loading your function and running [initialization code \(p. 32\)](#). This subsegment only appears for the first event processed by each instance of your function.
- **Invocation** – Represents the work done by your handler code. By instrumenting your code, you can extend this subsegment with additional subsegments.
- **Overhead** – Represents the work done by the Lambda runtime to prepare 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 .NET](#) in the AWS X-Ray Developer Guide.

Sections

- [Enabling active tracing with the Lambda API \(p. 664\)](#)
- [Enabling active tracing with AWS CloudFormation \(p. 664\)](#)

Enabling active tracing with the Lambda API

To manage tracing configuration with the AWS CLI or AWS SDK, use the following API operations:

- [UpdateFunctionConfiguration \(p. 935\)](#)
- [GetFunctionConfiguration \(p. 816\)](#)
- [CreateFunction \(p. 762\)](#)

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 that is locked 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 enable active 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  
      ...
```

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 runtimes

Name	Identifier	Operating system	
.NET Core 3.1	dotnetcore3.1	Amazon Linux 2	
.NET Core 2.1	dotnetcore2.1	Amazon Linux	

Lambda provides the following sample applications for the PowerShell runtime:

Sample Lambda applications in PowerShell

- [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 \(p. 667\)](#).

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 \(p. 668\)](#).

Topics

- [Setting Up a PowerShell Development Environment \(p. 667\)](#)
- [Deploy PowerShell Lambda functions with .zip file archives \(p. 668\)](#)
- [AWS Lambda function handler in PowerShell \(p. 671\)](#)
- [AWS Lambda context object in PowerShell \(p. 672\)](#)
- [AWS Lambda function logging in PowerShell \(p. 673\)](#)
- [AWS Lambda function errors in PowerShell \(p. 678\)](#)

Setting Up a PowerShell Development Environment

To set up your development environment for writing PowerShell scripts, do the following:

1. **Install the correct version of PowerShell** – Lambda's support for PowerShell is based on the cross-platform PowerShell Core 6.0 release. This means that you can develop your PowerShell Lambda functions on Windows, Linux, or Mac. If you don't have this version of PowerShell installed, you can find instructions in [Installing PowerShell Core](#).
2. **Install the .NET Core 3.1 SDK** – Because PowerShell Core is built on top of .NET Core, the Lambda support for PowerShell uses the same .NET Core 3.1 Lambda runtime for both .NET Core and PowerShell Lambda functions. The .NET Core 3.1 SDK is used by the new Lambda PowerShell publishing cmdlets to create the Lambda deployment package. The .NET Core 3.1 SDK is available at [.NET downloads](#) on the Microsoft website. Be sure to install the SDK and not the runtime installation.
3. **Install the AWSLambdaPSCore module** – You can install this either from the [PowerShell Gallery](#), or you can install it by using the following PowerShell Core shell command:

```
Install-Module AWSLambdaPSCore -Scope CurrentUser
```

4. **(Optional) Install AWS Tools for PowerShell** – You can install either the modularized [AWS.Tools](#) or single-module [AWSPowerShell.NetCore](#) version in PowerShell Core 6.0 to use the Lambda API within your PowerShell environment\. For instructions, see [Installing the AWS Tools for PowerShell](#)

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.

Tools and libraries

Lambda provides the following sample applications for the PowerShell runtime:

Tools and libraries for PowerShell

- [AWS Lambda and AWS Tools for PowerShell](#): the steps to install PowerShell, the .NET Core 2.1 SDK, and the `AWSLambdaPSCore` module.
- [aws-lambda-dotnet](#): Lambda tools for Powershell on GitHub.

The `AWSLambdaPSCore` module has the following new cmdlets to help author and publish PowerShell Lambda functions:

`AWSLambdaPSCore` cmdlets

- `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.

Sample applications

Lambda provides the following sample applications for the PowerShell runtime:

Sample Lambda applications in PowerShell

- [blank-powershell](#) – A PowerShell function that shows the use of logging, environment variables, and the AWS SDK.

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
...	

- 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)
```

- 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
```

- 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.

- 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 \(p. 668\)](#).

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 \(p. 671\)](#). 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 \(p. 95\)](#) 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 \(p. 673\)](#) 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 function metrics 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.

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 \(p. 673\)](#)
- [Using the Lambda console \(p. 674\)](#)
- [Using the CloudWatch console \(p. 674\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 674\)](#)
- [Deleting logs \(p. 677\)](#)

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_dotnetcore2.1_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: 1-5e843da6-733cxmpl7d0c3c020510040 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 TraceId** – For traced requests, the [AWS X-Ray trace ID \(p. 450\)](#).
- **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 \(p. 694\)](#).

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 \(p. 200\)](#). 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 \(p. 472\)](#).

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 CLI – Install 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":  
        "U1RBULQgUmVxdWVzdElkOia4N2QwNDRiOC1mMTU0LTEzTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",  
    "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": "Ag0Jb3JpZ2luX2VjELj...", "_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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#)

```
#!/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
```

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: $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf\tINFO\tENVIRONMENT VARIABLES\r{\r\t\"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",\\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf\tINFO\tEVENT\r{\r\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.

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 \(p. 678\)](#)
- [How it works \(p. 679\)](#)
- [Using the Lambda console \(p. 679\)](#)
- [Using the AWS Command Line Interface \(AWS CLI\) \(p. 680\)](#)
- [Error handling in other AWS services \(p. 680\)](#)
- [What's next? \(p. 681\)](#)

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 \(p. 195\)](#) 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 [Invoke API Errors \(p. 841\)](#).

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 \(p. 450\)](#) is enabled, in AWS X-Ray.

To invoke a function on the Lambda console

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Test**.
4. Select **New event** and then choose an **Event template** from the dropdown list.
5. Enter a name for the test event.

6. Enter the JSON for the test event.
7. Choose **Create event**.
8. Choose **Invoke**.

The Lambda console invokes your function [synchronously \(p. 138\)](#) 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 CLI – Install 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 are using AWS CLI version 2. You can also configure this option in your [AWS CLI config file](#)

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 \(p. 673\)](#).

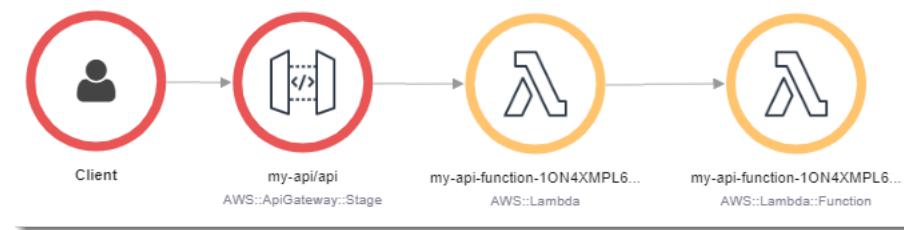
Error handling in other AWS services

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 \(p. 450\)](#).

What's next?

- Learn how to show logging events for your Lambda function on the [the section called "Logging" \(p. 673\)](#) page.

Monitoring and troubleshooting Lambda applications

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 more information about monitoring Lambda applications, see [Monitoring and observability](#) in the *Lambda operator guide*.

Sections

- [Monitoring functions in the AWS Lambda console \(p. 683\)](#)
- [Using Lambda Insights in Amazon CloudWatch \(p. 685\)](#)
- [Working with AWS Lambda function metrics \(p. 691\)](#)
- [Accessing Amazon CloudWatch logs for AWS Lambda \(p. 694\)](#)
- [Example workflows using other AWS services \(p. 695\)](#)

Monitoring functions in the AWS Lambda console

AWS 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 use the Lambda console's monitoring dashboard to monitor your Lambda functions and applications.

To monitor a function

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Monitoring**.

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 execution times.
- **Error count and success rate (%)** – The number of errors and the percentage of executions that completed without error.
- **Throttles** – The number of times that execution 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](#) on the Lambda console.
2. Choose a function.

3. Choose **Monitoring**.
4. 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, and then choose **View in metrics** to open the **Metrics** dashboard in CloudWatch.

Viewing queries in 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](#) on the Lambda console.
2. Choose a function.
3. Choose **Monitoring**.
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**.

What's next?

- Learn about the metrics that Lambda records and sends to CloudWatch in [Working with AWS Lambda function metrics \(p. 691\)](#).
- Learn how to use Lambda Insights to collect and aggregate Lambda function runtime performance metrics and logs in [Using Lambda Insights in Amazon CloudWatch \(p. 685\)](#).

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 \(p. 685\)](#)
- [Pricing \(p. 685\)](#)
- [Supported runtimes \(p. 685\)](#)
- [Enabling Lambda Insights in the Lambda console \(p. 685\)](#)
- [Enabling Lambda Insights programmatically \(p. 686\)](#)
- [Using the Lambda Insights dashboard \(p. 686\)](#)
- [Example workflow to detect function anomalies \(p. 687\)](#)
- [Example workflow using queries to troubleshoot a function \(p. 689\)](#)
- [What's next? \(p. 684\)](#)

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 \(p. 101\)](#). 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

For each Lambda function enabled for Lambda Insights, you only pay for what you use for metrics and logs. For a pricing example, see [Amazon CloudWatch pricing](#).

Supported runtimes

You can use Lambda Insights with any of the runtimes that support [Lambda extensions \(p. 209\)](#).

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](#) on 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 (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*.

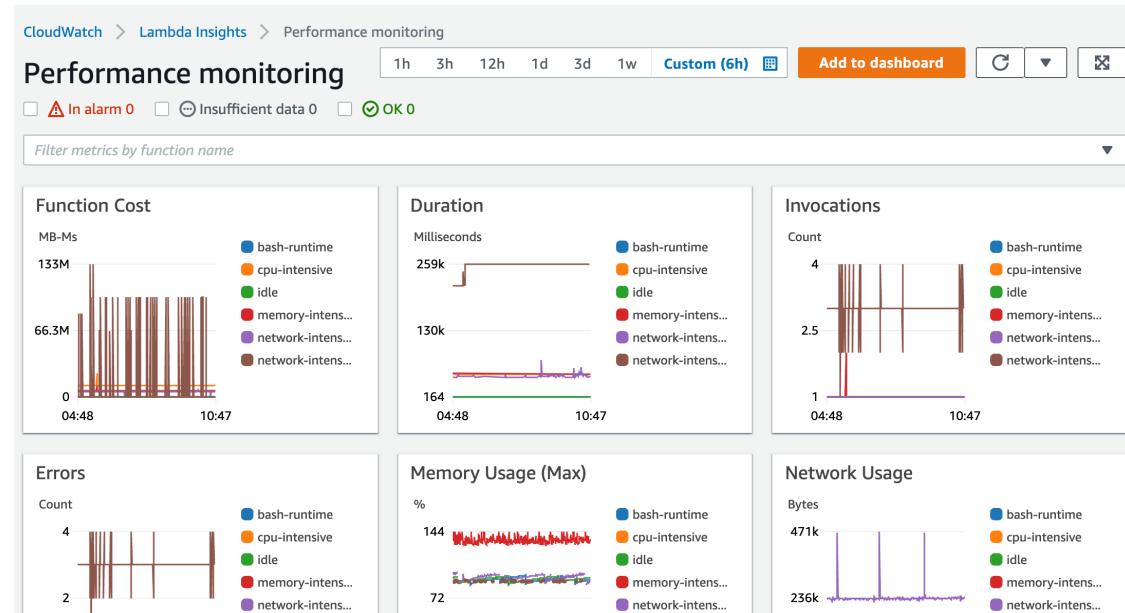
Using the Lambda Insights dashboard

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.

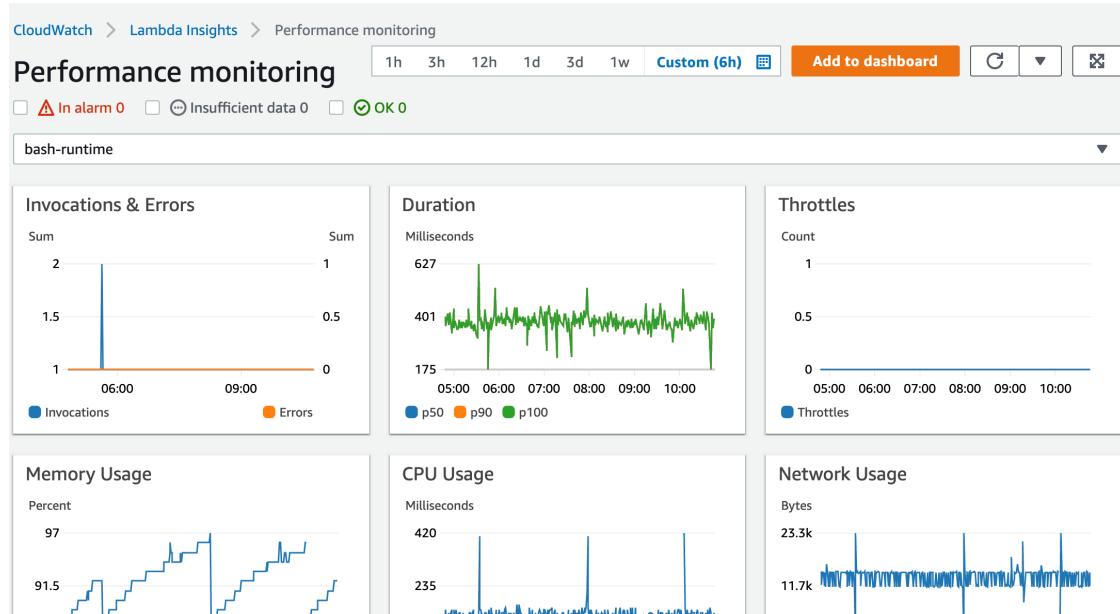
To view the runtime metrics for all functions

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.

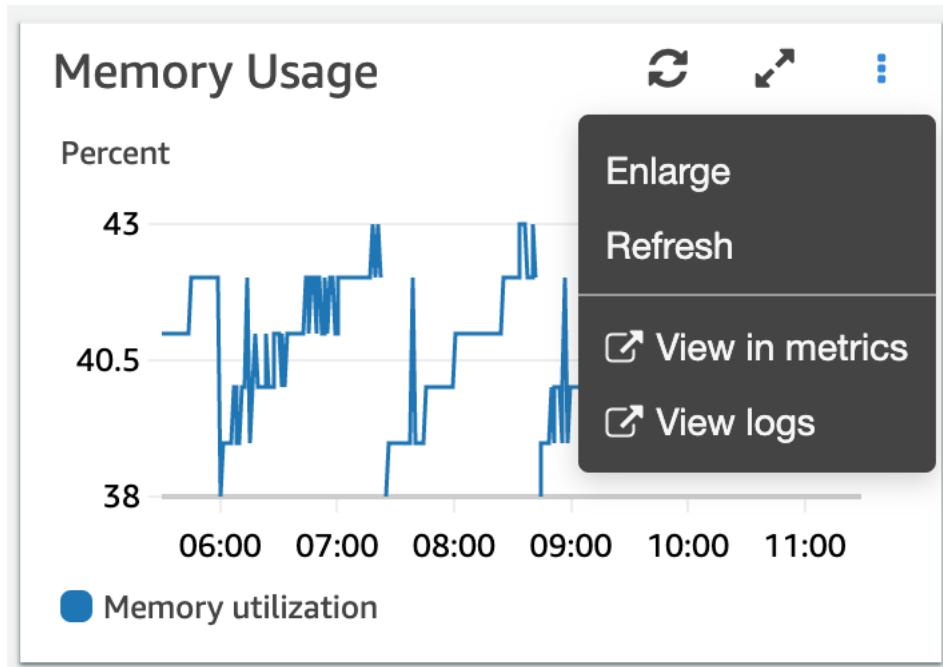
To enable anomaly detection for a function

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	▼	
□	bash-runtime	360	132.9167ms	4770 kB	<div style="width: 97%;">97%</div>	3		
□	cpu-intensive	359	6714.2897ms	4780 kB	<div style="width: 43%;">43%</div>	4		
□	idle	359	120.2507ms	4746 kB	<div style="width: 96%;">96%</div>	3		
□	memory-intensive	358	2385.9497ms	4794 kB	<div style="width: 144%;">144%</div>	4		
□	network-intensive	359	781.0585ms	82008 kB	<div style="width: 99%;">99%</div>	3		
□	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.

All metrics	Graphed metrics (6)	Graph options	Source	...	
Math expression ? Dynamic labels ?			Statistic: Maximum	Period: 1 Minute	Remove all
Label	Details	Statistic	Period	Y Axis	Actions
<input checked="" type="checkbox"/> bash-runtime	LambdaInsights • memory_utilization • functio...	Maximum	1 Minute	◀ ▶	alarm bell copy remove
<input checked="" type="checkbox"/> cpu-intensive	LambdaInsights • memory_utilization • functio...	Maximum	1 Minute	◀ ▶	alarm bell copy remove
<input checked="" type="checkbox"/> idle	LambdaInsights • memory_utilization • functio...	Maximum	1 Minute	◀ ▶	alarm bell copy remove
<input checked="" type="checkbox"/> memory-intensive	LambdaInsights • memory_utilization • functio...	Maximum	1 Minute	◀ ▶	alarm bell copy remove

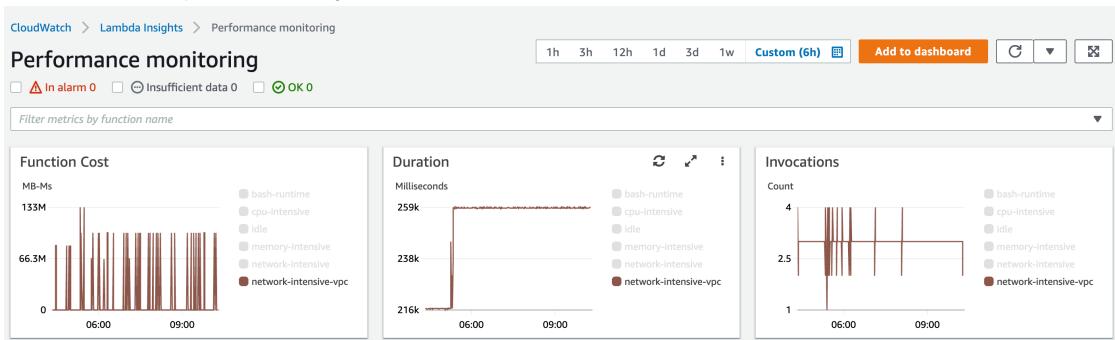
For more information, see [Using CloudWatch Anomaly Detection](#).

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)	
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

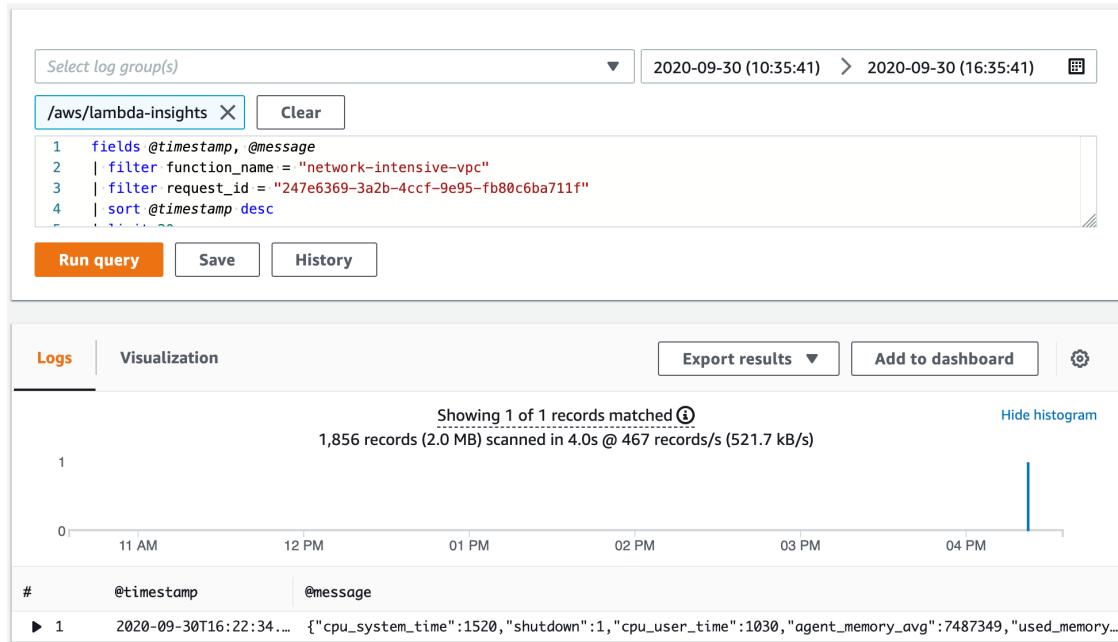
7. To show the **Most recent 1000 invocations**, choose the **Invocations** tab.
8. Select the **Timestamp** or **Message** for the invocation request that you want to troubleshoot.

Most recent 1000 invocations (1/45)						
Timestamp	Request ID	Trace	Memory %	Network IO	CPU time	Cold start
2020-09-30 16:22:34 (UTC-06:00)	247e6369-3a2b-...	-	91%	2 kB	2550ms	Yes
2020-09-30 16:13:39 (UTC-06:00)	311fb438-fa9d-4...	-	90%	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.



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 AWS Lambda function metrics

When your function finishes processing an event, Lambda sends metrics about the invocation to Amazon CloudWatch. You can build graphs and dashboards with these metrics in the CloudWatch console, and set alarms to respond to changes in utilization, performance, or error rates.

This page describes the Lambda function invocation, performance, and concurrency metrics available on the CloudWatch console.

Sections

- [Viewing metrics in the CloudWatch console \(p. 691\)](#)
- [Types of metrics \(p. 691\)](#)

Viewing metrics in the CloudWatch console

You can use the CloudWatch console to filter and sort function metrics by function name, alias, or version.

To view metrics in the CloudWatch console

1. Open the [Metrics page](#) (AWS/Lambda namespace) on the CloudWatch console.
2. Choose a dimension.
 - **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 \(p. 97\)](#).
 - **Across All Functions** (none) – View aggregate metrics for all functions in the current AWS Region.
3. Choose metrics to add them to the graph.

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 execution, this can be several minutes before the metric is emitted. If, for example, your function has a 10-minute timeout, 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 type of metrics available on the CloudWatch console.

Using invocation metrics

Invocation metrics are binary indicators of the outcome of an invocation. For example, if the function returns an error, 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 one minute.

You should view the following metrics with the `Sum` statistic.

Invocation metrics

- **Invocations** – The number of times your function code is executed, including successful executions and executions that result in a function error. Invocations aren't recorded if the invocation request is throttled or otherwise resulted in an [invocation error \(p. 841\)](#). This equals the number of requests billed.
- **Errors** – The number of invocations that result in a function error. Function errors include exceptions thrown by your code and exceptions thrown by the Lambda runtime. 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 \(p. 140\)](#), the number of times Lambda attempts to send an event to a dead-letter queue but fails. Dead-letter errors can occur due to permissions errors, misconfigured resources, or size limits.
- **DestinationDeliveryFailures** – For asynchronous invocation, the number of times Lambda attempts to send an event to a [destination \(p. 38\)](#) but fails. Delivery errors can occur due to permissions errors, misconfigured resources, or size limits.
- **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 [TooManyRequestsException \(p. 841\)](#). Throttled requests and other invocation errors don't count as `Invocations` or `Errors`.
- **ProvisionedConcurrencyInvocations** – The number of times your function code is executed on [provisioned concurrency \(p. 84\)](#).
- **ProvisionedConcurrencySpilloverInvocations** – The number of times your function code is executed on standard concurrency when all provisioned concurrency is in use.

Using performance metrics

Performance metrics provide performance details about a single 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.

Performance metrics

- **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 \(p. 148\)](#) that read from streams, the age of the last record in the event. The age is the amount of time between when the stream receives the record and when the event source mapping sends the event to the function.

`Duration` also supports [percentile 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 executions, excluding the slowest 5 percent.

Using 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, view these metrics with the `Max` statistic.

Concurrency metrics

- `ConcurrentExecutions` – The number of function instances that are processing events. If this number reaches your [concurrent executions quota \(p. 44\)](#) for the Region, or the [reserved concurrency limit \(p. 84\)](#) that you configured on the function, additional invocation requests are throttled.
- `ProvisionedConcurrentExecutions` – The number of function instances that are processing events on [provisioned concurrency \(p. 84\)](#). 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 an AWS Region, the number of events that are being processed by functions that don't have reserved concurrency.

Accessing Amazon CloudWatch logs for AWS Lambda

AWS Lambda automatically monitors Lambda functions on your behalf, reporting metrics through Amazon CloudWatch. To help you troubleshoot failures in a function, 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. This page describes how to view logs using the Lambda console.

Sections

- [Prerequisites \(p. 694\)](#)
- [Pricing \(p. 694\)](#)
- [Using the Lambda console \(p. 694\)](#)
- [Using the AWS CLI \(p. 695\)](#)
- [What's next? \(p. 695\)](#)

Prerequisites

Your [execution role \(p. 47\)](#) needs permission to upload logs to CloudWatch Logs. You can add CloudWatch Logs permissions using an AWS managed policy provided by Lambda. For more information, see [AWS managed policies for Lambda features \(p. 47\)](#).

Pricing

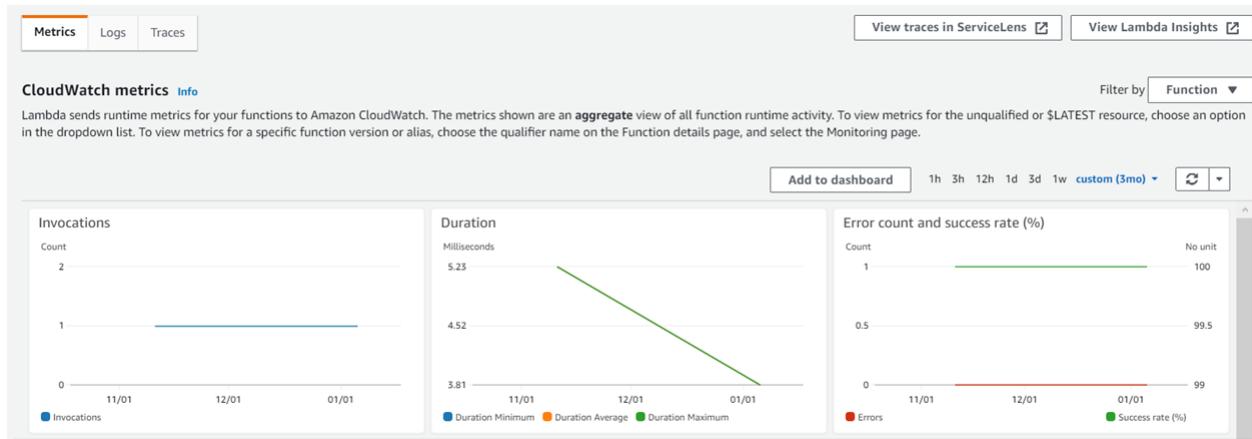
There is no additional charge for using Lambda logs; however, standard CloudWatch Logs charges apply. For more information, see [CloudWatch pricing](#).

Using the Lambda console

The following section describes how to view logs for your function on the Lambda console.

To view logs using the Lambda console

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Monitoring**.



A graphical representation of the metrics for the Lambda function are shown.

4. Choose [View logs in CloudWatch](#).

Note

Logs may not be immediately available after invocation. We recommend waiting 5 to 10 minutes before checking your execution role.

Using the AWS CLI

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 \(p. 495\)](#)
- [AWS Lambda function logging in Python \(p. 533\)](#)
- [AWS Lambda function logging in Ruby \(p. 556\)](#)
- [AWS Lambda function logging in Java \(p. 589\)](#)
- [AWS Lambda function logging in Go \(p. 623\)](#)
- [AWS Lambda function logging in C# \(p. 652\)](#)
- [AWS Lambda function logging in PowerShell \(p. 673\)](#)

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*.

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 \(p. 696\)](#)
- [Pricing \(p. 696\)](#)
- [Example AWS X-Ray workflow to view a service map \(p. 697\)](#)
- [Example AWS X-Ray workflow to view trace details \(p. 697\)](#)
- [Example AWS Trusted Advisor workflow to view recommendations \(p. 698\)](#)
- [What's next? \(p. 698\)](#)

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](#) on 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

1. Open the [Functions page](#) on the Lambda console.
2. Choose a function.
3. Choose **Monitoring**.
4. Choose **View traces in X-Ray**.
5. Choose **Service map**.
6. Choose from the predefined time ranges, or choose a custom time range.
7. 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

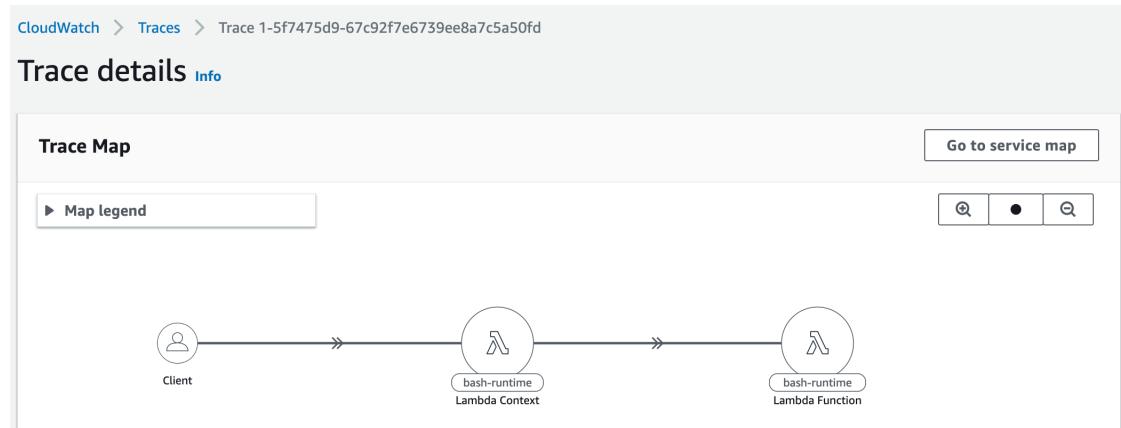
1. Open the [single-function view](#) in the CloudWatch console.
2. Choose the **Application logs** tab.
3. Use the **Timestamp** or **Message** to identify the invocation request that you want to troubleshoot.
4. To show the **Most recent 1000 invocations**, choose the **Invocations** tab.

The screenshot shows the CloudWatch Lambda Insights interface with the 'Invocations' tab selected. At the top, there's a button for 'View logs' and a navigation bar with pages 1 through 6. Below that is a table titled 'Most recent 1000 invocations (359)'. The table has columns for 'Timestamp', 'Request ID', 'Trace', 'Memory %', and 'Network IO'. The first three rows of data are:

	Timestamp	Request ID	Trace	Memory %	Network IO
<input type="checkbox"/>	2020-09-30 12:12:05 (UTC-06:00)	00c99bab-92f7-46cc-af28-ca71ad43f894	View	91%	14 kB
<input type="checkbox"/>	2020-09-30 14:35:05 (UTC-06:00)	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](#).

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 \(p. 699\)](#)
- [Identity and access management for Lambda \(p. 701\)](#)
- [Compliance validation for AWS Lambda \(p. 709\)](#)
- [Resilience in AWS Lambda \(p. 709\)](#)
- [Infrastructure security in AWS Lambda \(p. 710\)](#)
- [Configuration and vulnerability analysis in AWS Lambda \(p. 710\)](#)

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 user accounts with 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 recommend TLS 1.2 or later.
- 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 personal 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 sensitive identifying information, such as your customers' account numbers, into free-form 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 Lambda or other services might get picked up for inclusion in diagnostic logs. When you provide a URL to an external server, don't include credentials information in the URL to validate your request to that server.

Sections

- [Encryption in transit \(p. 700\)](#)
- [Encryption at rest \(p. 700\)](#)

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).

When you [connect your function to a file system \(p. 126\)](#), Lambda uses [Encryption in transit](#) for all connections.

For a full list of API endpoints, see [AWS Regions and endpoints](#) in the AWS General Reference.

Encryption at rest

You can use environment variables to store secrets securely for use with Lambda functions. Lambda always encrypts environment variables at rest.

Additionally, you can use the following features to customize how environment variables are encrypted.

- **Key configuration** – On a per-function basis, you can configure Lambda to use an encryption key that you create and manage in AWS Key Management Service. These are referred to as *customer managed* customer master keys (CMKs) or customer managed keys. If you don't configure a customer managed key, Lambda uses an AWS managed CMK named aws/lambda, which Lambda creates in your account.
- **Encryption helpers** – The Lambda console lets you encrypt environment variable values client side, before sending them to Lambda. This enhances security further by preventing secrets from being displayed unencrypted in the Lambda console, or in function configuration that's returned by the Lambda API. The console also provides sample code that you can adapt to decrypt the values in your function handler.

For more information, see [Using AWS Lambda environment variables \(p. 77\)](#).

Lambda always encrypts files that you upload to Lambda, including [deployment packages \(p. 12\)](#) and [layer archives \(p. 101\)](#).

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 \(p. 701\)](#)
- [Authenticating with identities \(p. 701\)](#)
- [Managing access using policies \(p. 703\)](#)
- [How AWS Lambda works with IAM \(p. 705\)](#)
- [AWS Lambda identity-based policy examples \(p. 705\)](#)
- [Troubleshooting AWS Lambda identity and access \(p. 706\)](#)

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 \(p. 706\)](#).

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 employees 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 \(p. 705\)](#).

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 \(p. 705\)](#).

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see [Signing in to the AWS Management Console as an IAM user or root user](#) in the *IAM User Guide*.

You must be *authenticated* (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company's single sign-on authentication or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles. When you access AWS using credentials from another company, you are assuming a role indirectly.

To sign in directly to the [AWS Management Console](#), use your password with your root user email address or your IAM user name. You can access AWS programmatically using your root user or IAM users access keys. AWS provides SDK and command line tools to cryptographically sign your request using your credentials. If you don't use AWS tools, you must sign the request yourself. Do this using

Signature Version 4, a protocol for authenticating inbound API requests. For more information about authenticating requests, see [Signature Version 4 signing process](#) in the *AWS General Reference*.

Regardless of the authentication method that you use, you might also 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 [Using multi-factor authentication \(MFA\) in AWS](#) in the *IAM User Guide*.

AWS account root user

When you first create an AWS account, you begin with a single 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 do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the [best practice of using the root user only to create your first IAM user](#). Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see [Managing access keys for IAM users](#) in the *IAM User Guide*. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.

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:

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.
- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as *federated users*. AWS assigns a role to a federated user when access is requested through an *identity provider*. For more information about federated users, see [Federated users and roles](#) in the *IAM User Guide*.
- **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. Service roles provide access only within your account and cannot be used to grant access to services in other accounts. 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 IAM 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 IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. You can sign in as the root user or an IAM user, or you can assume an IAM role. When you then make a request, AWS evaluates the related identity-based or resource-based policies. 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**.

Every IAM entity (user or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

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 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 [AWS Lambda permissions](#) (p. 46).

AWS Lambda identity-based policy examples

By default, IAM 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 IAM 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 IAM 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. 705)
- [Using the Lambda console](#) (p. 706)
- [Allow users to view their own permissions](#) (p. 706)

Policy best practices

Identity-based policies are very powerful. They 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 using AWS managed policies** – To start using Lambda quickly, use AWS managed policies to give your employees the permissions they need. These policies are already available in your account and are maintained and updated by AWS. For more information, see [Get started using permissions with AWS managed policies](#) in the *IAM User Guide*.
- **Grant least privilege** – When you create custom policies, grant only the permissions required to perform a task. Start with a minimum set of permissions and grant additional permissions as necessary. Doing so is more secure than starting with permissions that are too lenient and then trying to tighten them later. For more information, see [Grant least privilege](#) in the *IAM User Guide*.
- **Enable MFA for sensitive operations** – For extra security, require IAM users to use multi-factor authentication (MFA) to access sensitive resources or API operations. For more information, see [Using multi-factor authentication \(MFA\) in AWS](#) in the *IAM User Guide*.
- **Use policy conditions for extra security** – To the extent that it's practical, define the conditions under which your identity-based policies allow access to a resource. For example, you can write conditions to specify a range of allowable IP addresses that a request must come from. You can also write conditions

to allow requests only within a specified date or time range, or to require the use of SSL or MFA. For more information, see [IAM JSON policy elements: Condition](#) 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 (IAM users or roles) with that policy.

For an example policy that grants minimal access for function development, see [Function development \(p. 56\)](#). 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 \(p. 257\)](#).

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>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 \(p. 707\)](#)
- [I am not authorized to perform iam:PassRole \(p. 707\)](#)
- [I want to view my access keys \(p. 707\)](#)
- [I'm an administrator and want to allow others to access Lambda \(p. 708\)](#)
- [I'm an administrator and want to migrate from AWS managed policies for Lambda that will be deprecated \(p. 708\)](#)
- [I want to allow people outside of my AWS account to access my Lambda resources \(p. 708\)](#)

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 user name and password.

The following example error occurs when the `mateojackson` IAM 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, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password. Ask that person to update your policies 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 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 asks her administrator to update her policies to allow her to perform the `iam:PassRole` action.

I want to view my access keys

After you create your IAM user access keys, you can view your access key ID at any time. However, you can't view your secret access key again. If you lose your secret key, you must create a new access key pair.

Access keys consist of two parts: an access key ID (for example, `AKIAIOSFODNN7EXAMPLE`) and a secret access key (for example, `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`). Like a user name and password, you must use both the access key ID and secret access key together to authenticate your requests. Manage your access keys as securely as you do your user name and password.

Important

Do not provide your access keys to a third party, even to help [find your canonical user ID](#). By doing this, you might give someone permanent access to your account.

When you create an access key pair, you are prompted to save the access key ID and secret access key in a secure location. The secret access key is available only at the time you create it. If you lose your secret access key, you must add new access keys to your IAM user. You can have a maximum of two access keys. If you already have two, you must delete one key pair before creating a new one. To view instructions, see [Managing access keys](#) in the *IAM User Guide*.

I'm an administrator and want to allow others to access Lambda

To allow others to access Lambda, you must create an IAM entity (user or role) for the person or application that needs access. They will use the credentials for that entity to access AWS. You must then attach a policy to the entity that grants them the correct permissions in Lambda.

To get started right away, see [Creating your first IAM delegated user and group](#) in the *IAM User Guide*.

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 IAM 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 \(p. 705\)](#).
- 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 whitepaper](#) – 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 \(p. 95\)](#).
- **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 \(p. 44\)](#) that can be increased if needed. For details, see [AWS Lambda function scaling \(p. 153\)](#).
- **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 \(p. 111\)](#).
- **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 [Managing concurrency for a Lambda function \(p. 84\)](#).
- **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 \(p. 158\)](#).
- **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 [AWS Lambda function dead-letter queues \(p. 145\)](#).

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 [Amazon Web Services: Overview of security processes](#) whitepaper.

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 \(p. 195\)](#) 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 \(p. 12\)](#), or in [layers \(p. 101\)](#) that you attach to your function. You can also build [custom runtimes \(p. 236\)](#) 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 support policy \(p. 198\)](#).

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 \(p. 712\)](#)
- [Troubleshoot invocation issues in Lambda \(p. 714\)](#)
- [Troubleshoot execution issues in Lambda \(p. 718\)](#)
- [Troubleshoot networking issues in Lambda \(p. 719\)](#)
- [Troubleshoot container image issues in Lambda \(p. 720\)](#)

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 644 $(find . -type f)
chmod 755 $(find . -type d)
```

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](#) (p. 927) 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. 926), 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:0x000055b76ccebf98>*

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. 483)
- [Building Lambda functions with Python](#) (p. 509)

- [Building Lambda functions with Ruby \(p. 546\)](#)
- [Building Lambda functions with Java \(p. 569\)](#)
- [Building Lambda functions with Go \(p. 609\)](#)
- [Building Lambda functions with C# \(p. 637\)](#)
- [Building Lambda functions with PowerShell \(p. 666\)](#)

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": "nodejs12.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 \(p. 80\)](#).

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 \(p. 839\)](#).

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 AWS Identity and Access Management (IAM) 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 IAM 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 [AWS Lambda permissions \(p. 46\)](#).

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 [Monitoring the state of a function with the Lambda API \(p. 151\)](#).

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 \(p. 926\)](#)
- [UpdateFunctionConfiguration \(p. 935\)](#)
- [PublishVersion \(p. 882\)](#)

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 \(p. 84\)](#). 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 \(p. 257\)](#) and accounts permission to invoke a function in the function's [resource-based policy \(p. 51\)](#). If the invoker is in another account, that user must also have [permission to invoke functions \(p. 56\)](#).

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 \(p. 406\)](#). To stop the function from running, on the [function configuration page \(p. 70\)](#), choose **Throttle**. Then, identify the code path or configuration error that caused the recursive invocation.

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 \(p. 87\)](#), 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 \(p. 87\)](#) on the function version or alias.

Note

Provisioned concurrency is not configurable on the [\\$LATEST version \(p. 72\)](#).

- 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 \(p. 691\)](#). 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 \(p. 691\)](#). A non-zero value indicates that all provisioned concurrency is in use and some invocation occurred with a cold start.
- Check your [invocation frequency \(p. 44\)](#) (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: 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 \(p. 126\)](#) 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 \(p. 126\)](#) 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 \(p. 126\)](#), but the mount operation timed out. Try again after a short time and consider limiting the function's [concurrency \(p. 84\)](#) 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 \(p. 352\)](#).

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 \(p. 714\)](#).

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 \(p. 70\)](#) 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 \(p. 47\)](#) to grant it permission. Add the following managed policies to enable logs and tracing.

- **AWSLambdaBasicExecutionRole**
- **AWSXRayDaemonWriteAccess**

When you add permissions to your function, make an update to its code or configuration as well. This forces running instances of your function, which have out-of-date credentials, to stop and be replaced.

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 \(p. 486\)](#).

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 \(p. 195\)](#). 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 \(p. 101\)](#). For details on creating a deployment package with dependencies, see the following topics:

- [Deploy Node.js Lambda functions with .zip file archives \(p. 488\)](#)
- [Deploy Python Lambda functions with .zip file archives \(p. 514\)](#)
- [Deploy Ruby Lambda functions with .zip file archives \(p. 550\)](#)
- [Deploy Java Lambda functions with .zip or JAR file archives \(p. 577\)](#)
- [Deploy Go Lambda functions with .zip file archives \(p. 616\)](#)
- [Deploy C# Lambda functions with .zip file archives \(p. 644\)](#)
- [Deploy PowerShell Lambda functions with .zip file archives \(p. 668\)](#)

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 \(p. 514\)](#).

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 \(p. 111\)](#). When you use this feature, you manage the function's internet access and network connectivity with VPC resources.

Network connectivity errors can result from issues in routing configuration, security group rules, role permissions, network address translation, or the availability of resources such as IP addresses or network interfaces. They may result in a specific error or, if a request can't reach its destination, a timeout.

VPC: Function loses internet access or times out

Issue: *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*

When you connect a function to a VPC, all outbound requests go through your 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 \(p. 116\)](#).

VPC: Function needs access to AWS services without using the internet

Issue: *Function needs access to AWS services without using the internet*

To connect to AWS services from a private subnet with no internet access, use VPC endpoints. For a sample template with VPC endpoints for DynamoDB and Amazon S3, see [???](#) (p. 116).

VPC: Limit was reached for the function's VPC

Error: *ENILimitReachedException: The elastic network interface limit was reached for the function's VPC.*

When you connect a function to a VPC, Lambda creates an elastic network interface for each combination of subnet and security group attached to the function. These network interfaces are limited to 250 per VPC, but this limit can be increased. To request an increase, use the [Support Center console](#).

Troubleshoot container image issues in Lambda

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 ENTRYPOINT to your container image includes the absolute path as the location. Also verify that the image does not contain a symlink as the ENTRYPOINT.

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: ENTRypoint is being overridden with a null or empty value

Error: You are using an AWS CloudFormation template, and your container ENTRypoint 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 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](#).

update-history-change	update-history-description	update-history-date
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. 722)	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
Code signing for Lambda functions	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
Preview: Lambda Runtime Logs API	Lambda now supports the Runtime Logs API. Lambda extensions can use the Logs API to subscribe to log streams in the execution environment. For	November 12, 2020

	details, see Lambda Runtime Logs API .	
New event source to for Amazon MQ	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
Preview: Lambda Extensions API	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
Support for Java 8 and custom runtimes on AL2	Lambda now supports Java 8 and custom runtimes on Amazon Linux 2. For details, see Lambda runtimes .	August 12, 2020
New event source for Amazon Managed Streaming for Apache Kafka	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
IAM condition keys for Amazon VPC settings	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
Concurrency settings for Kinesis HTTP/2 stream consumers	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

Batch window for Kinesis HTTP/2 stream consumers	You can now configure a batch window (<code>MaximumBatchingWindowInSeconds</code>) 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
Support for Amazon EFS file systems	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
AWS CDK sample applications in the Lambda console	The Lambda console now includes sample applications that use the AWS Cloud Development Kit (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
Support for .NET Core 3.1.0 runtime in AWS Lambda	AWS Lambda now supports the .NET Core 3.1.0 runtime. For details, see .NET Core CLI .	March 31, 2020
Support for API Gateway HTTP APIs	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
Ruby 2.7	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

Concurrency metrics	Lambda now reports the <code>ConcurrentExecutions</code> metric for all functions, aliases, and versions. You can view a graph for this metric on the monitoring page for your function. Previously, <code>ConcurrentExecutions</code> 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. 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 24, 2020
Updates to function configuration API output	Added reason codes to <code>StateReasonCode</code> (<code>InvalidSubnet</code> , <code>InvalidSecurityGroup</code>) and <code>LastUpdateStatusReasonCode</code> (<code>SubnetOutOfRangeAddresses</code> , <code>InvalidSubnet</code> , <code>InvalidSecurityGroup</code>) for functions that connect to a VPC. For more information about states, see AWS Lambda function states .	January 20, 2020

Provisioned concurrency	You can now allocate provisioned concurrency for a function version or alias. Provisioned concurrency enables a function to scale without fluctuations in latency. For details, see Managing concurrency for a Lambda function .	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

Use Kinesis HTTP/2 stream consumers as a trigger	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
Python 3.7	AWS Lambda now supports Python 3.7 with a new runtime. For more information, see Building Lambda functions with Python .	November 19, 2018
Payload limit increase for asynchronous function invocation	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
AWS GovCloud (US-East) Region	AWS Lambda is now available in the AWS GovCloud (US-East) Region.	November 12, 2018
Moved AWS SAM topics to a separate Developer Guide	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
View Lambda applications in the console	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
Function execution timeout limit	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. 483) .	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. 609) and Building Lambda functions with C# (p. 637) .	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 Creating functions using the AWS Lambda console editor (p. 17) .	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 Managing concurrency for a Lambda function (p. 84) .	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. 187) .	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. 450) .	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. 483) .	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.	<p>AWS Lambda added support for C#. For more information, see Building Lambda functions with C# (p. 637).</p> <p>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. 305).</p>	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. 386) .	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. 174).</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. 77).</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. For more information, see Create a Lambda function with the console (p. 9) .	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. 483) .	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. 111) .	February 11, 2016
Lambda runtime has been updated.	The execution environment (p. 195) 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. 509).</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. 95).</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 CloudWatch Events (p. 295).</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. 324) .	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. 261).</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. 8) .	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. 569) .	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. 8)</p> <p>AWS Lambda</p>	April 9, 2015
Preview release	Preview release of the <i>AWS Lambda Developer Guide</i> .	November 13, 2014

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 \(p. 736\)](#)
- [Data Types \(p. 949\)](#)

Actions

The following actions are supported:

- [AddLayerVersionPermission \(p. 738\)](#)
- [AddPermission \(p. 742\)](#)
- [CreateAlias \(p. 746\)](#)
- [CreateCodeSigningConfig \(p. 750\)](#)
- [CreateEventSourceMapping \(p. 753\)](#)
- [CreateFunction \(p. 762\)](#)
- [DeleteAlias \(p. 774\)](#)
- [DeleteCodeSigningConfig \(p. 776\)](#)
- [DeleteEventSourceMapping \(p. 778\)](#)
- [DeleteFunction \(p. 783\)](#)
- [DeleteFunctionCodeSigningConfig \(p. 785\)](#)
- [DeleteFunctionConcurrency \(p. 787\)](#)
- [DeleteFunctionEventInvokeConfig \(p. 789\)](#)
- [DeleteLayerVersion \(p. 791\)](#)
- [DeleteProvisionedConcurrencyConfig \(p. 793\)](#)
- [GetAccountSettings \(p. 795\)](#)
- [GetAlias \(p. 797\)](#)
- [GetCodeSigningConfig \(p. 800\)](#)
- [GetEventSourceMapping \(p. 802\)](#)
- [GetFunction \(p. 807\)](#)
- [GetFunctionCodeSigningConfig \(p. 811\)](#)
- [GetFunctionConcurrency \(p. 814\)](#)
- [GetFunctionConfiguration \(p. 816\)](#)
- [GetFunctionEventInvokeConfig \(p. 823\)](#)
- [GetLayerVersion \(p. 826\)](#)

- [GetLayerVersionByArn \(p. 829\)](#)
- [GetLayerVersionPolicy \(p. 832\)](#)
- [GetPolicy \(p. 834\)](#)
- [GetProvisionedConcurrencyConfig \(p. 836\)](#)
- [Invoke \(p. 839\)](#)
- [InvokeAsync \(p. 845\)](#)
- [ListAliases \(p. 847\)](#)
- [ListCodeSigningConfigs \(p. 850\)](#)
- [ListEventSourceMappings \(p. 852\)](#)
- [ListFunctionEventInvokeConfigs \(p. 855\)](#)
- [ListFunctions \(p. 858\)](#)
- [ListFunctionsByCodeSigningConfig \(p. 862\)](#)
- [ListLayers \(p. 864\)](#)
- [ListLayerVersions \(p. 866\)](#)
- [ListProvisionedConcurrencyConfigs \(p. 869\)](#)
- [ListTags \(p. 872\)](#)
- [ListVersionsByFunction \(p. 874\)](#)
- [PublishLayerVersion \(p. 878\)](#)
- [PublishVersion \(p. 882\)](#)
- [PutFunctionCodeSigningConfig \(p. 890\)](#)
- [PutFunctionConcurrency \(p. 893\)](#)
- [PutFunctionEventInvokeConfig \(p. 896\)](#)
- [PutProvisionedConcurrencyConfig \(p. 900\)](#)
- [RemoveLayerVersionPermission \(p. 903\)](#)
- [RemovePermission \(p. 905\)](#)
- [TagResource \(p. 907\)](#)
- [UntagResource \(p. 909\)](#)
- [UpdateAlias \(p. 911\)](#)
- [UpdateCodeSigningConfig \(p. 915\)](#)
- [UpdateEventSourceMapping \(p. 918\)](#)
- [UpdateFunctionCode \(p. 926\)](#)
- [UpdateFunctionConfiguration \(p. 935\)](#)
- [UpdateFunctionEventInvokeConfig \(p. 946\)](#)

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 \(p. 903\)](#) 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 \(p. 738\)](#)

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 \(p. 738\)](#)

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 \(p. 738\)](#)

The version number.

Required: Yes

Request Body

The request accepts the following data in JSON format.

[Action \(p. 738\)](#)

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 \(p. 738\)](#)

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 \(p. 738\)](#)

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 \(p. 738\)](#)

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 \(p. 739\)](#)

A unique identifier for the current revision of the policy.

Type: String

[Statement \(p. 739\)](#)

The permission statement.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

HTTP Status Code: 400

PolicyLengthExceededException

The permissions policy for the resource is too large. [Learn more](#)

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 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.

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 or another account 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.

To grant permission to another account, specify the account ID as the `Principal`. For AWS services, the principal is a domain-style identifier defined by the service, like `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 action adds a statement to a resource-based permissions policy for the function. For more information about function policies, see [Lambda Function Policies](#).

Request Syntax

```
POST /2015-03-31/functions/FunctionName/policy?Qualifier=Qualifier HTTP/1.1
Content-type: application/json

{
  "Action": "string",
  "EventSourceToken": "string",
  "Principal": "string",
  "RevisionId": "string",
  "SourceAccount": "string",
  "SourceArn": "string",
  "StatementId": "string"
}
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 742\)](#)

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 \(p. 742\)](#)

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. 742)

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. 742)

For Alexa Smart Home functions, a token that must be supplied by the invoker.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Pattern: [a-zA-Z0-9._\-_]+

Required: No

Principal (p. 742)

The AWS service or 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

RevisionId (p. 742)

Only update the policy 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. 742)

For Amazon S3, the ID of the account that owns the resource. Use this together with `SourceArn` to ensure that the resource is owned by the specified account. 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 \(p. 742\)](#)

For AWS services, the ARN of the AWS resource that invokes the function. For example, an Amazon S3 bucket or Amazon SNS topic.

Type: String

Pattern: arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+:(a-z){2}(-gov)?-[a-zA-Z]+\d{1})?:(\d{12})?:(.*)

Required: No

[StatementId \(p. 742\)](#)

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 \(p. 744\)](#)

The permission statement that's added to the function policy.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

HTTP Status Code: 400

PolicyLengthExceeded

The permissions policy for the resource is too large. [Learn more](#)

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 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.

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. 746)

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. 746)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

[FunctionVersion \(p. 746\)](#)

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 \(p. 746\)](#)

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 \(p. 746\)](#)

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 955\)](#) 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 \(p. 747\)](#)

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 \(p. 747\)](#)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[FunctionVersion \(p. 747\)](#)

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST|[0-9]+)

[Name \(p. 747\)](#)

The name of the alias.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (?![0-9]+\$)([a-zA-Z0-9-_]+)

[RevisionId \(p. 747\)](#)

A unique identifier that changes when you update the alias.

Type: String

[RoutingConfig \(p. 747\)](#)

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 955\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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 \(p. 750\)](#)

Signing profiles for this code signing configuration.

Type: [AllowedPublishers \(p. 956\)](#) object

Required: Yes

[CodeSigningPolicies \(p. 750\)](#)

The code signing policies define the actions to take if the validation checks fail.

Type: [CodeSigningPolicies \(p. 959\)](#) object

Required: No

[Description \(p. 750\)](#)

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 \(p. 750\)](#)

The code signing configuration.

Type: [CodeSigningConfig \(p. 957\)](#) object

Errors

InvalidArgumentException

One of the parameters in the request is invalid.

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 triggers the function.

For details about each event source type, see the following topics. In particular, each of the topics describes the required and optional parameters for the specific event source.

- [Configuring a Dynamo DB stream as an event source](#)
- [Configuring a Kinesis stream as an event source](#)
- [Configuring an SQS queue as an event source](#)
- [Configuring an MQ broker as an event source](#)
- [Configuring MSK as an event source](#)
- [Configuring Self-Managed Apache Kafka as an event source](#)

The following error handling options are only available 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.

Request Syntax

```
POST /2015-03-31/event-source-mappings/ HTTP/1.1
Content-type: application/json

{
    "BatchSize": number,
    "BisectBatchOnFunctionError": boolean,
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "Enabled": boolean,
    "EventSourceArn": "string",
    "FunctionName": "string",
    "FunctionResponseTypes": [ "string" ],
    "MaximumBatchingWindowInSeconds": number,
    "MaximumRecordAgeInSeconds": number,
    "MaximumRetryAttempts": number,
    "ParallelizationFactor": number,
    "Queues": [ "string" ],
    "SelfManagedEventSource": {
        "Endpoints": {
            "string": [ "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.

[BatchSize \(p. 753\)](#)

The maximum number of items to retrieve in a single batch.

- **Amazon Kinesis** - Default 100. Max 10,000.
- **Amazon DynamoDB Streams** - Default 100. Max 1,000.
- **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.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

Required: No

[BisectBatchOnFunctionError \(p. 753\)](#)

(Streams only) If the function returns an error, split the batch in two and retry.

Type: Boolean

Required: No

[DestinationConfig \(p. 753\)](#)

(Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: [DestinationConfig \(p. 962\)](#) object

Required: No

[Enabled \(p. 753\)](#)

If true, the event source mapping is active. Set to false to pause polling and invocation.

Type: Boolean

Required: No

[EventSourceArn \(p. 753\)](#)

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.

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

[FunctionName \(p. 753\)](#)

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 \(p. 753\)](#)

(Streams only) 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 \(p. 753\)](#)

(Streams and SQS standard queues) The maximum amount of time to gather records before invoking the function, in seconds.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

Required: No

[MaximumRecordAgeInSeconds \(p. 753\)](#)

(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 \(p. 753\)](#)

(Streams only) Discard records after the specified number of retries. The default value is infinite (-1). When set to infinite (-1), failed records will be retried until the record expires.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

Required: No

[ParallelizationFactor \(p. 753\)](#)

(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 \(p. 753\)](#)

(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

[SelfManagedEventSource \(p. 753\)](#)

The Self-Managed Apache Kafka cluster to send records.

Type: [SelfManagedEventSource \(p. 994\)](#) object

Required: No

[SourceAccessConfigurations \(p. 753\)](#)

An array of authentication protocols or VPC components required to secure your event source.

Type: Array of [SourceAccessConfiguration \(p. 995\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

Required: No

[StartingPosition \(p. 753\)](#)

The position in a stream from which to start reading. Required for Amazon Kinesis, Amazon DynamoDB, and Amazon MSK Streams sources. AT_TIMESTAMP is only supported for Amazon Kinesis streams.

Type: String

Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

Required: No

[StartingPositionTimestamp \(p. 753\)](#)

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds.

Type: Timestamp

Required: No

[Topics \(p. 753\)](#)

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 \(p. 753\)](#)

(Streams only) The duration in seconds of a processing window. The range is between 1 second up to 900 seconds.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

Required: No

Response Syntax

```
HTTP/1.1 202
Content-type: application/json

{
    "BatchSize": number,
    "BisectBatchOnFunctionError": boolean,
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "EventSourceArn": "string",
    "FunctionArn": "string",
    "FunctionResponseTypes": [ "string" ],
    "LastModified": number,
    "LastProcessingResult": "string",
    "MaximumBatchingWindowInSeconds": number,
    "MaximumRecordAgeInSeconds": number,
    "MaximumRetryAttempts": number,
    "ParallelizationFactor": number,
```

```

    "Queues": [ "string" ],
    "SelfManagedEventSource": {
        "Endpoints": {
            "string" : [ "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.

[BatchSize \(p. 757\)](#)

The maximum number of items to retrieve in a single batch.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

[BisectBatchOnFunctionError \(p. 757\)](#)

(Streams only) If the function returns an error, split the batch in two and retry. The default value is false.

Type: Boolean

[DestinationConfig \(p. 757\)](#)

(Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: [DestinationConfig \(p. 962\)](#) object

[EventSourceArn \(p. 757\)](#)

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})?:(.*)

[FunctionArn \(p. 757\)](#)

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 \(p. 757\)](#)

(Streams only) 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 \(p. 757\)](#)

The date that the event source mapping was last updated, or its state changed, in Unix time seconds.

Type: Timestamp

[LastProcessingResult \(p. 757\)](#)

The result of the last AWS Lambda invocation of your Lambda function.

Type: String

[MaximumBatchingWindowInSeconds \(p. 757\)](#)

(Streams and SQS standard queues) The maximum amount of time to gather records before invoking the function, in seconds. The default value is zero.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

[MaximumRecordAgeInSeconds \(p. 757\)](#)

(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.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 604800.

[MaximumRetryAttempts \(p. 757\)](#)

(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 \(p. 757\)](#)

(Streams only) The number of batches to process from each shard concurrently. The default value is 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10.

[Queues \(p. 757\)](#)

(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]*

[SelfManagedEventSource \(p. 757\)](#)

The Self-Managed Apache Kafka cluster for your event source.

Type: [SelfManagedEventSource \(p. 994\)](#) object

[SourceAccessConfigurations \(p. 757\)](#)

An array of the authentication protocol, or the VPC components to secure your event source.

Type: Array of [SourceAccessConfiguration \(p. 995\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

[StartingPosition \(p. 757\)](#)

The position in a stream from which to start reading. Required for Amazon Kinesis, Amazon DynamoDB, and Amazon MSK Streams sources. AT_TIMESTAMP is only supported for Amazon Kinesis streams.

Type: String

Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

[StartingPositionTimestamp \(p. 757\)](#)

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds.

Type: Timestamp

[State \(p. 757\)](#)

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 \(p. 757\)](#)

Indicates whether the last change to the event source mapping was made by a user, or by the Lambda service.

Type: String

[Topics \(p. 757\)](#)

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\\-_\\.]+)+

[TumblingWindowInSeconds \(p. 757\)](#)

(Streams only) The duration in seconds of a processing window. The range is between 1 second up to 900 seconds.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

UUID (p. 757)

The identifier of the event source mapping.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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 X-Ray for request tracing.

You set the package type to `Image` if the deployment package is a [container 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.

You set the package type to `Zip` if the deployment package is a [.zip file archive](#). For a .zip file archive, the `code` property specifies the location of the .zip file. You must also specify the handler and runtime properties.

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 \(p. 816\)](#) indicate when the function is ready to invoke. For more information, see [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 \(p. 935\)](#). Function-level settings apply to both the unpublished and published versions of the function, and include tags ([TagResource \(p. 907\)](#)) and per-function concurrency limits ([PutFunctionConcurrency \(p. 893\)](#)).

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 \(p. 926\)](#), Lambda checks that the code package has a valid signature from a trusted publisher. The code-signing configuration includes set set of signing profiles, which define the trusted publishers for this function.

If another account or an AWS service invokes your function, use [AddPermission \(p. 742\)](#) to grant permission by creating a resource-based IAM policy. You can grant permissions at the function level, on a version, or on an alias.

To invoke your function directly, use [Invoke \(p. 839\)](#). To invoke your function in response to events in other AWS services, create an event source mapping ([CreateEventSourceMapping \(p. 753\)](#)), or configure a function trigger in the other service. For more information, see [Invoking Functions](#).

Request Syntax

```
POST /2015-03-31/functions HTTP/1.1
Content-type: application/json

{
  "Code": {
    "ImageUri": "string",
    "S3Bucket": "string",
    "S3Key": "string",
    "S3ObjectVersion": "string",
    "ZipFile": blob
  },
}
```

```
"CodeSigningConfigArn": "string",
"DeadLetterConfig": {
    "TargetArn": "string"
},
"Description": "string",
"Environment": {
    "Variables": {
        "string" : "string"
    }
},
"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",
"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.

Code (p. 762)

The code for the function.

Type: [FunctionCode \(p. 971\)](#) object

Required: Yes

CodeSigningConfigArn (p. 762)

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-zA-Z0-9]{17}

Required: No

[DeadLetterConfig \(p. 762\)](#)

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 \(p. 961\)](#) object

Required: No

[Description \(p. 762\)](#)

A description of the function.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

[Environment \(p. 762\)](#)

Environment variables that are accessible from function code during execution.

Type: [Environment \(p. 963\)](#) object

Required: No

[FileSystemConfigs \(p. 762\)](#)

Connection settings for an Amazon EFS file system.

Type: Array of [FileSystemConfig \(p. 970\)](#) objects

Array Members: Maximum number of 1 item.

Required: No

[FunctionName \(p. 762\)](#)

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 \(p. 762\)](#)

The name of the method within your code that Lambda calls to execute your function. The format includes the file name. It can also include namespaces and other qualifiers, depending on the runtime. For more information, see [Programming Model](#).

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

Required: No

[ImageConfig \(p. 762\)](#)

Container image configuration values that override the values in the container image Dockerfile.

Type: [ImageConfig \(p. 981\)](#) object

Required: No

[KMSKeyArn \(p. 762\)](#)

The ARN of the AWS Key Management Service (AWS KMS) key that's used to encrypt your function's environment variables. If it's not provided, AWS Lambda uses a default service key.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-.]+:[.]*|())`

Required: No

[Layers \(p. 762\)](#)

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 \(p. 762\)](#)

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 \(p. 762\)](#)

The type of deployment package. Set to `Image` for container image and set `Zip` for ZIP archive.

Type: String

Valid Values: Zip | Image

Required: No

Publish (p. 762)

Set to true to publish the first version of the function during creation.

Type: Boolean

Required: No

Role (p. 762)

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. 762)

The identifier of the function's [runtime](#).

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2

Required: No

Tags (p. 762)

A list of [tags](#) to apply to the function.

Type: String to string map

Required: No

Timeout (p. 762)

The amount of time that Lambda allows a function to run before stopping it. The default is 3 seconds. The maximum allowed value is 900 seconds. For additional information, see [Lambda execution environment](#).

Type: Integer

Valid Range: Minimum value of 1.

Required: No

TracingConfig (p. 762)

Set Mode to Active to sample and trace a subset of incoming requests with [X-Ray](#).

Type: [TracingConfig \(p. 996\)](#) object

Required: No

VpcConfig (p. 762)

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 only access resources and the internet through that VPC. For more information, see [VPC Settings](#).

Type: [VpcConfig \(p. 998\)](#) object

Required: No

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
    "CodeSha256": "string",
    "CodeSize": number,
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
    "Description": "string",
    "Environment": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "Variables": {
            "string" : "string"
        }
    },
    "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",
"SigningJobArn": "string",
"SigningProfileVersionArn": "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.

[CodeSha256 \(p. 767\)](#)

The SHA256 hash of the function's deployment package.

Type: String

[CodeSize \(p. 767\)](#)

The size of the function's deployment package, in bytes.

Type: Long

[DeadLetterConfig \(p. 767\)](#)

The function's dead letter queue.

Type: [DeadLetterConfig \(p. 961\)](#) object

[Description \(p. 767\)](#)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[Environment \(p. 767\)](#)

The function's [environment variables](#).

Type: [EnvironmentResponse \(p. 965\)](#) object

[FileSystemConfigs \(p. 767\)](#)

Connection settings for an [Amazon EFS file system](#).

Type: Array of [FileSystemConfig \(p. 970\)](#) objects

Array Members: Maximum number of 1 item.

[FunctionArn \(p. 767\)](#)

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 \(p. 767\)](#)

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 \(p. 767\)](#)

The function that Lambda calls to begin executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

[ImageConfigResponse \(p. 767\)](#)

The function's image configuration values.

Type: [ImageConfigResponse \(p. 983\)](#) object

[KMSKeyArn \(p. 767\)](#)

The KMS key that's used to encrypt the function's environment variables. This key is only returned if you've configured a customer managed CMK.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-_\.]+:*)|()`

[LastModified \(p. 767\)](#)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[LastUpdateStatus \(p. 767\)](#)

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 \(p. 767\)](#)

The reason for the last update that was performed on the function.

Type: String

[LastUpdateStatusReasonCode \(p. 767\)](#)

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`

[Layers \(p. 767\)](#)

The function's [layers](#).

Type: Array of [Layer \(p. 984\)](#) objects

[MasterArn \(p. 767\)](#)

For Lambda@Edge functions, the ARN of the master 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 \(p. 767\)](#)

The amount of memory available to the function at runtime.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 10240.

[PackageType \(p. 767\)](#)

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 \(p. 767\)](#)

The latest updated revision of the function or alias.

Type: String

[Role \(p. 767\)](#)

The function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\\-/]+`

[Runtime \(p. 767\)](#)

The runtime environment for the Lambda function.

Type: String

Valid Values: `nodejs` | `nodejs4.3` | `nodejs6.10` | `nodejs8.10` | `nodejs10.x` | `nodejs12.x` | `nodejs14.x` | `java8` | `java8.al2` | `java11` | `python2.7` | `python3.6` | `python3.7` | `python3.8` | `dotnetcore1.0` | `dotnetcore2.0` | `dotnetcore2.1`

| dotnetcore3.1 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided |
provided.al2

[SigningJobArn \(p. 767\)](#)

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 \(p. 767\)](#)

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})?:(.*)

[State \(p. 767\)](#)

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 \(p. 767\)](#)

The reason for the function's current state.

Type: String

[StateReasonCode \(p. 767\)](#)

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`

[Timeout \(p. 767\)](#)

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 \(p. 767\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 997\)](#) object

[Version \(p. 767\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (`\$LATEST` | [0-9]+)

[VpcConfig \(p. 767\)](#)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 999\)](#) object

Errors

CodeSigningConfigNotFoundException

The specified code signing configuration does not exist.

HTTP Status Code: 404

CodeStorageExceededException

You have exceeded your maximum total code size per account. [Learn more](#)

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. Lambda always blocks deployment if the integrity check fails, even if code signing policy is set to WARN.

HTTP Status Code: 400

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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 \(p. 774\)](#)

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 \(p. 774\)](#)

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 invalid.

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.

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. 776)

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

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 invalid.

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 \(p. 852\)](#).

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. 778)

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

{
    "BatchSize": number,
    "BisectBatchOnFunctionError": boolean,
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "EventSourceArn": "string",
    "FunctionArn": "string",
    "FunctionResponseTypes": [ "string" ],
    "LastModified": number,
    "LastProcessingResult": "string",
    "MaximumBatchingWindowInSeconds": number,
    "MaximumRecordAgeInSeconds": number,
    "MaximumRetryAttempts": number,
    "ParallelizationFactor": number,
    "Queues": [ "string" ],
    "SelfManagedEventSource": {
        "Endpoints": {
            "string": [ "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.

BatchSize (p. 778)

The maximum number of items to retrieve in a single batch.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

BisectBatchOnFunctionError (p. 778)

(Streams only) If the function returns an error, split the batch in two and retry. The default value is false.

Type: Boolean

DestinationConfig (p. 778)

(Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: [DestinationConfig \(p. 962\)](#) object

EventSourceArn (p. 778)

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})?:(.*)

FunctionArn (p. 778)

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. 778)

(Streams only) 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 \(p. 778\)](#)

The date that the event source mapping was last updated, or its state changed, in Unix time seconds.

Type: Timestamp

[LastProcessingResult \(p. 778\)](#)

The result of the last AWS Lambda invocation of your Lambda function.

Type: String

[MaximumBatchingWindowInSeconds \(p. 778\)](#)

(Streams and SQS standard queues) The maximum amount of time to gather records before invoking the function, in seconds. The default value is zero.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

[MaximumRecordAgeInSeconds \(p. 778\)](#)

(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.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 604800.

[MaximumRetryAttempts \(p. 778\)](#)

(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 \(p. 778\)](#)

(Streams only) The number of batches to process from each shard concurrently. The default value is 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10.

[Queues \(p. 778\)](#)

(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]*

[SelfManagedEventSource \(p. 778\)](#)

The Self-Managed Apache Kafka cluster for your event source.

Type: [SelfManagedEventSource \(p. 994\)](#) object

[SourceAccessConfigurations \(p. 778\)](#)

An array of the authentication protocol, or the VPC components to secure your event source.

Type: Array of [SourceAccessConfiguration \(p. 995\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

[StartingPosition \(p. 778\)](#)

The position in a stream from which to start reading. Required for Amazon Kinesis, Amazon DynamoDB, and Amazon MSK Streams sources. `AT_TIMESTAMP` is only supported for Amazon Kinesis streams.

Type: String

Valid Values: `TRIM_HORIZON` | `LATEST` | `AT_TIMESTAMP`

[StartingPositionTimestamp \(p. 778\)](#)

With `StartingPosition` set to `AT_TIMESTAMP`, the time from which to start reading, in Unix time seconds.

Type: Timestamp

[State \(p. 778\)](#)

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 \(p. 778\)](#)

Indicates whether the last change to the event source mapping was made by a user, or by the Lambda service.

Type: String

[Topics \(p. 778\)](#)

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 \(p. 778\)](#)

(Streams only) The duration in seconds of a processing window. The range is between 1 second up to 900 seconds.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

[UUID \(p. 778\)](#)

The identifier of the event source mapping.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

HTTP Status Code: 400

ResourceInUseException

The operation conflicts with the resource's availability. For example, you attempted to update an EventSource Mapping in CREATING, or tried to delete a EventSource mapping currently in the UPDATING state.

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.

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 \(p. 778\)](#). 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. 783)

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. 783)

Specify a version to delete. You can't delete a version that's referenced by an alias.

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 invalid.

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.

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. 785)

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 invalid.

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.

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. 787)

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 invalid.

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.

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 \(p. 896\)](#).

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 \(p. 789\)](#)

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 \(p. 789\)](#)

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 invalid.

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.

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. 791)

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. 791)

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.

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 \(p. 793\)](#)

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 \(p. 793\)](#)

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 invalid.

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.

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 \(p. 795\)](#)

Limits that are related to concurrency and code storage.

Type: [AccountLimit \(p. 951\)](#) object

[AccountUsage \(p. 795\)](#)

The number of functions and amount of storage in use.

Type: [AccountUsage \(p. 952\)](#) object

Errors

[ServiceException](#)

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded.

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 \(p. 797\)](#)

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 \(p. 797\)](#)

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 \(p. 797\)](#)

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 \(p. 797\)](#)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[FunctionVersion \(p. 797\)](#)

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (\\$LATEST|[0-9]+)

[Name \(p. 797\)](#)

The name of the alias.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: (?![0-9]+\\$)([a-zA-Z0-9-_]+)

[RevisionId \(p. 797\)](#)

A unique identifier that changes when you update the alias.

Type: String

[RoutingConfig \(p. 797\)](#)

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 955\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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. 800)

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

{
  "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. 800)

The code signing configuration

Type: [CodeSigningConfig \(p. 957\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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 \(p. 852\)](#).

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. 802)

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

{
    "BatchSize": number,
    "BisectBatchOnFunctionError": boolean,
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "EventSourceArn": "string",
    "FunctionArn": "string",
    "FunctionResponseTypes": [ "string" ],
    "LastModified": number,
    "LastProcessingResult": "string",
    "MaximumBatchingWindowInSeconds": number,
    "MaximumRecordAgeInSeconds": number,
    "MaximumRetryAttempts": number,
    "ParallelizationFactor": number,
    "Queues": [ "string" ],
    "SelfManagedEventSource": {
        "Endpoints": {
            "string": [ "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.

[BatchSize \(p. 802\)](#)

The maximum number of items to retrieve in a single batch.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

[BisectBatchOnFunctionError \(p. 802\)](#)

(Streams only) If the function returns an error, split the batch in two and retry. The default value is false.

Type: Boolean

[DestinationConfig \(p. 802\)](#)

(Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: [DestinationConfig \(p. 962\)](#) object

[EventSourceArn \(p. 802\)](#)

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})?:(.*)

[FunctionArn \(p. 802\)](#)

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 \(p. 802\)](#)

(Streams only) 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 \(p. 802\)](#)

The date that the event source mapping was last updated, or its state changed, in Unix time seconds.

Type: Timestamp

[LastProcessingResult \(p. 802\)](#)

The result of the last AWS Lambda invocation of your Lambda function.

Type: String

[MaximumBatchingWindowInSeconds \(p. 802\)](#)

(Streams and SQS standard queues) The maximum amount of time to gather records before invoking the function, in seconds. The default value is zero.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

[MaximumRecordAgeInSeconds \(p. 802\)](#)

(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.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 604800.

[MaximumRetryAttempts \(p. 802\)](#)

(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 \(p. 802\)](#)

(Streams only) The number of batches to process from each shard concurrently. The default value is 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10.

[Queues \(p. 802\)](#)

(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]*

[SelfManagedEventSource \(p. 802\)](#)

The Self-Managed Apache Kafka cluster for your event source.

Type: [SelfManagedEventSource \(p. 994\)](#) object

[SourceAccessConfigurations \(p. 802\)](#)

An array of the authentication protocol, or the VPC components to secure your event source.

Type: Array of [SourceAccessConfiguration \(p. 995\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

[StartingPosition \(p. 802\)](#)

The position in a stream from which to start reading. Required for Amazon Kinesis, Amazon DynamoDB, and Amazon MSK Streams sources. `AT_TIMESTAMP` is only supported for Amazon Kinesis streams.

Type: String

Valid Values: `TRIM_HORIZON` | `LATEST` | `AT_TIMESTAMP`

[StartingPositionTimestamp \(p. 802\)](#)

With `StartingPosition` set to `AT_TIMESTAMP`, the time from which to start reading, in Unix time seconds.

Type: Timestamp

[State \(p. 802\)](#)

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 \(p. 802\)](#)

Indicates whether the last change to the event source mapping was made by a user, or by the Lambda service.

Type: String

[Topics \(p. 802\)](#)

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 \(p. 802\)](#)

(Streams only) The duration in seconds of a processing window. The range is between 1 second up to 900 seconds.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

[UUID \(p. 802\)](#)

The identifier of the event source mapping.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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 \(p. 807\)](#)

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 \(p. 807\)](#)

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": {
    "ImageUri": "string",
    "Location": "string",
    "RepositoryType": "string",
```

```
    "ResolvedImageUri": "string"
},
"Concurrency": {
    "ReservedConcurrentExecutions": number
},
"Configuration": {
    "CodeSha256": "string",
    "CodeSize": number,
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
    "Description": "string",
    "Environment": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "Variables": {
            "string" : "string"
        }
    },
    "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",
    "SigningJobArn": "string",
    "SigningProfileVersionArn": "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 \(p. 807\)](#)

The deployment package of the function or version.

Type: [FunctionCodeLocation \(p. 973\)](#) object

[Concurrency \(p. 807\)](#)

The function's reserved concurrency.

Type: [Concurrency \(p. 960\)](#) object

[Configuration \(p. 807\)](#)

The configuration of the function or version.

Type: [FunctionConfiguration \(p. 974\)](#) object

[Tags \(p. 807\)](#)

The function's tags.

Type: String to string map

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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. 811)

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. 811)

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 \(p. 811\)](#)

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 invalid.

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.

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 \(p. 893\)](#).

Request Syntax

```
GET /2019-09-30/functions/FunctionName/concurrency HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName (p. 814)

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. 814)

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 invalid.

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.

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 \(p. 935\)](#).

To get all of a function's details, including function-level settings, use [GetFunction \(p. 807\)](#).

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 \(p. 816\)](#)

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 \(p. 816\)](#)

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

{
    "CodeSha256": "string",
    "CodeSize": number,
```

```

"DeadLetterConfig": {
    "TargetArn": "string"
},
"Description": "string",
"Environment": {
    "Error": {
        "ErrorCode": "string",
        "Message": "string"
    },
    "Variables": {
        "string" : "string"
    }
},
"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",
"SigningJobArn": "string",
"SigningProfileVersionArn": "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.

CodeSha256 (p. 816)

The SHA256 hash of the function's deployment package.

Type: String

CodeSize (p. 816)

The size of the function's deployment package, in bytes.

Type: Long

DeadLetterConfig (p. 816)

The function's dead letter queue.

Type: [DeadLetterConfig](#) (p. 961) object

Description (p. 816)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Environment (p. 816)

The function's [environment variables](#).

Type: [EnvironmentResponse](#) (p. 965) object

FileSystemConfigs (p. 816)

Connection settings for an [Amazon EFS file system](#).

Type: Array of [FileSystemConfig](#) (p. 970) objects

Array Members: Maximum number of 1 item.

FunctionArn (p. 816)

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. 816)

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 \(p. 816\)](#)

The function that Lambda calls to begin executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

[ImageConfigResponse \(p. 816\)](#)

The function's image configuration values.

Type: [ImageConfigResponse \(p. 983\)](#) object

[KMSKeyArn \(p. 816\)](#)

The KMS key that's used to encrypt the function's environment variables. This key is only returned if you've configured a customer managed CMK.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-.]+:[^.]+|()`

[LastModified \(p. 816\)](#)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[LastUpdateStatus \(p. 816\)](#)

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 \(p. 816\)](#)

The reason for the last update that was performed on the function.

Type: String

[LastUpdateStatusReasonCode \(p. 816\)](#)

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`

[Layers \(p. 816\)](#)

The function's [layers](#).

Type: Array of [Layer \(p. 984\)](#) objects

[MasterArn \(p. 816\)](#)

For Lambda@Edge functions, the ARN of the master 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 \(p. 816\)](#)

The amount of memory available to the function at runtime.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 10240.

[PackageType \(p. 816\)](#)

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 \(p. 816\)](#)

The latest updated revision of the function or alias.

Type: String

[Role \(p. 816\)](#)

The function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/[a-zA-Z_0-9+=,.@\-_/.]+`

[Runtime \(p. 816\)](#)

The runtime environment for the Lambda function.

Type: String

Valid Values: `nodejs` | `nodejs4.3` | `nodejs6.10` | `nodejs8.10` | `nodejs10.x` | `nodejs12.x` | `nodejs14.x` | `java8` | `java8.al2` | `java11` | `python2.7` | `python3.6` | `python3.7` | `python3.8` | `dotnetcore1.0` | `dotnetcore2.0` | `dotnetcore2.1` | `dotnetcore3.1` | `nodejs4.3-edge` | `go1.x` | `ruby2.5` | `ruby2.7` | `provided` | `provided.al2`

[SigningJobArn \(p. 816\)](#)

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 \(p. 816\)](#)

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})?:(.*)

[State \(p. 816\)](#)

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 \(p. 816\)](#)

The reason for the function's current state.

Type: String

[StateReasonCode \(p. 816\)](#)

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`

[Timeout \(p. 816\)](#)

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 \(p. 816\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 997\)](#) object

[Version \(p. 816\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

[VpcConfig \(p. 816\)](#)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 999\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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 \(p. 896\)](#).

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 \(p. 823\)](#)

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 \(p. 823\)](#)

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": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    }
}
```

```
        },
        "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 \(p. 823\)](#)

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 an SQS queue.
- **Topic** - The ARN of an SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: [DestinationConfig \(p. 962\)](#) object

[FunctionArn \(p. 823\)](#)

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 \(p. 823\)](#)

The date and time that the configuration was last updated, in Unix time seconds.

Type: Timestamp

[MaximumEventAgeInSeconds \(p. 823\)](#)

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 \(p. 823\)](#)

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 invalid.

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.

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 \(p. 826\)](#)

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 \(p. 826\)](#)

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

{
    "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.

CompatibleRuntimes (p. 826)

The layer's compatible runtimes.

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 | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2

Content (p. 826)

Details about the layer version.

Type: [LayerVersionContentOutput \(p. 987\)](#) object

CreatedDate (p. 826)

The date that the layer version was created, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

Description (p. 826)

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

LayerArn (p. 826)

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. 826)

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. 826)

The layer's software license.

Type: String

Length Constraints: Maximum length of 512.

Version (p. 826)

The version number.

Type: Long

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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. 829)

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

{
    "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.

[CompatibleRuntimes \(p. 829\)](#)

The layer's compatible runtimes.

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 | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2

[Content \(p. 829\)](#)

Details about the layer version.

Type: [LayerVersionContentOutput \(p. 987\)](#) object

[CreatedDate \(p. 829\)](#)

The date that the layer version was created, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[Description \(p. 829\)](#)

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[LayerArn \(p. 829\)](#)

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 \(p. 829\)](#)

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 \(p. 829\)](#)

The layer's software license.

Type: String

Length Constraints: Maximum length of 512.

[Version \(p. 829\)](#)

The version number.

Type: Long

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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 \(p. 738\)](#).

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. 832)

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. 832)

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. 832)

The policy document.

Type: String

[RevisionId \(p. 832\)](#)

A unique identifier for the current revision of the policy.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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. 834)

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. 834)

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 \(p. 834\)](#)

The resource-based policy.

Type: String

[RevisionId \(p. 834\)](#)

A unique identifier for the current revision of the policy.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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. 836)

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. 836)

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 \(p. 836\)](#)

The amount of provisioned concurrency allocated.

Type: Integer

Valid Range: Minimum value of 0.

[AvailableProvisionedConcurrentExecutions \(p. 836\)](#)

The amount of provisioned concurrency available.

Type: Integer

Valid Range: Minimum value of 0.

[LastModified \(p. 836\)](#)

The date and time that a user last updated the configuration, in [ISO 8601 format](#).

Type: String

[RequestedProvisionedConcurrentExecutions \(p. 836\)](#)

The amount of provisioned concurrency requested.

Type: Integer

Valid Range: Minimum value of 1.

[Status \(p. 836\)](#)

The status of the allocation process.

Type: String

Valid Values: IN_PROGRESS | READY | FAILED

[StatusReason \(p. 836\)](#)

For failed allocations, the reason that provisioned concurrency could not be allocated.

Type: String

Errors

[InvalidParameterValueException](#)

One of the parameters in the request is invalid.

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.

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 [Retry Behavior](#).

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, [limit errors](#), or issues with your function's code and configuration. For example, Lambda returns `TooManyRequestsException` if executing 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 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.

This operation requires permission for the [lambda:InvokeFunction](#) action.

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 \(p. 839\)](#)

Up to 3583 bytes of base64-encoded data about the invoking client to pass to the function in the context object.

[FunctionName \(p. 839\)](#)

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 \(p. 839\)](#)

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 it's 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 \(p. 839\)](#)

Set to Tail to include the execution log in the response.

Valid Values: None | Tail

[Qualifier \(p. 839\)](#)

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 \(p. 839\)](#)

The JSON that you want to provide to your Lambda function as input.

Response Syntax

```
HTTP/1.1 StatusCode
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 \(p. 840\)](#)

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 \(p. 840\)](#)

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 \(p. 840\)](#)

If present, indicates that an error occurred during function execution. Details about the error are included in the response payload.

[LogResult \(p. 840\)](#)

The last 4 KB of the execution log, which is base64 encoded.

The response returns the following as the HTTP body.

[Payload \(p. 840\)](#)

The response from the function, or an error object.

Errors

EC2AccessDeniedException

Need additional permissions to configure VPC settings.

HTTP Status Code: 502

EC2ThrottledException

AWS Lambda was throttled by Amazon EC2 during Lambda function initialization using the execution role provided for the Lambda function.

HTTP Status Code: 502

EC2UnexpectedException

AWS Lambda received an unexpected 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 function couldn't make a network connection to the configured file system.

HTTP Status Code: 408

EFSMountFailureException

The function couldn't mount the configured file system due to a permission or configuration issue.

HTTP Status Code: 403

EFSMountTimeoutException

The function was able to make a network connection to the configured file system, but the mount operation timed out.

HTTP Status Code: 408

ENILimitReachedException

AWS Lambda was not able to create an elastic network interface in the VPC, specified as part of Lambda function configuration, because the limit for network interfaces has been reached.

HTTP Status Code: 502

InvalidParameterValueException

One of the parameters in the request is invalid.

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 invalid.

HTTP Status Code: 502

InvalidSubnetIDException

The Subnet ID provided in the Lambda function VPC configuration is invalid.

HTTP Status Code: 502

InvalidZipFileException

AWS Lambda could not unzip the deployment package.

HTTP Status Code: 502

KMSAccessDeniedException

Lambda was unable to decrypt the environment variables because KMS access was denied. Check the Lambda function's KMS permissions.

HTTP Status Code: 502

KMSDisabledException

Lambda was unable to decrypt the environment variables because the KMS key used is disabled. Check the Lambda function's KMS key settings.

HTTP Status Code: 502

KMSInvalidStateException

Lambda was unable to decrypt the environment variables because the KMS key used is in an invalid state for Decrypt. Check the function's KMS key settings.

HTTP Status Code: 502

KMSNotFoundException

Lambda was unable to decrypt the environment variables because the 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 limit. For more information, see [Limits](#).

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

SubnetIPAddressLimitReachedException

AWS Lambda was not able to 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.

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 \(p. 839\)](#).

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 \(p. 845\)](#)

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 \(p. 845\)](#)

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 \(p. 845\)](#)

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](#)

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. 847)

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. 847)

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. 847)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MaxItems](#) (p. 847)

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 \(p. 847\)](#)

A list of aliases.

Type: Array of [AliasConfiguration \(p. 953\)](#) objects

[NextMarker \(p. 847\)](#)

The pagination token that's included if more results are available.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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 \(p. 850\)](#)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MaxItems \(p. 850\)](#)

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 \(p. 850\)](#)

The code signing configurations

Type: Array of [CodeSigningConfig \(p. 957\)](#) objects

[NextMarker \(p. 850\)](#)

The pagination token that's included if more results are available.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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 only show 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 \(p. 852\)](#)

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.

Pattern: `arn:(aws[a-zA-Z0-9-]*):([a-zA-Z0-9\-])+:([a-z]{2}(-gov)?-[a-z]+\-\d{1}):(\d{12}):(.*)`

[FunctionName \(p. 852\)](#)

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 \(p. 852\)](#)

A pagination token returned by a previous call.

[MaxItems \(p. 852\)](#)

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": [
        {
            "BatchSize": number,
            "BisectBatchOnFunctionError": boolean,
            "DestinationConfig": {
                "OnFailure": {
                    "Destination": "string"
                },
                "OnSuccess": {
                    "Destination": "string"
                }
            },
            "EventSourceArn": "string",
            "FunctionArn": "string",
            "FunctionResponseTypes": [ "string" ],
            "LastModified": number,
            "LastProcessingResult": "string",
            "MaximumBatchingWindowInSeconds": number,
            "MaximumRecordAgeInSeconds": number,
            "MaximumRetryAttempts": number,
            "ParallelizationFactor": number,
            "Queues": [ "string" ],
            "SelfManagedEventSource": {
                "Endpoints": {
                    "string": [ "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 \(p. 853\)](#)

A list of event source mappings.

Type: Array of [EventSourceMappingConfiguration \(p. 966\)](#) objects

[NextMarker \(p. 853\)](#)

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 invalid.

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.

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 \(p. 896\)](#).

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 \(p. 855\)](#)

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 \(p. 855\)](#)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MaxItems \(p. 855\)](#)

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  
    }  
,  
    "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.

[FunctionEventInvokeConfigs \(p. 855\)](#)

A list of configurations.

Type: Array of [FunctionEventInvokeConfig \(p. 979\)](#) objects

[NextMarker \(p. 855\)](#)

The pagination token that's included if more results are available.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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` action returns a subset of the [FunctionConfiguration \(p. 974\)](#) fields.

To get the additional fields (`State`, `StateReasonCode`, `StateReason`, `LastUpdateStatus`, `LastUpdateStatusReason`, `LastUpdateStatusReasonCode`) for a function or version, use [GetFunction \(p. 807\)](#).

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 \(p. 858\)](#)

Set to `ALL` to include entries for all published versions of each function.

Valid Values: `ALL`

[Marker \(p. 858\)](#)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MasterRegion \(p. 858\)](#)

For Lambda@Edge functions, the AWS Region of the master function. For example, `us-east-1` filters the list of functions to only include 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 \(p. 858\)](#)

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": [
    {
      "CodeSha256": "string",
      "CodeSize": number,
      "DeadLetterConfig": {
        "TargetArn": "string"
      },
      "Description": "string",
      "Environment": {
        "Error": {
          "ErrorCode": "string",
          "Message": "string"
        },
        "Variables": {
          "string" : "string"
        }
      },
      "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",
      "SigningJobArn": "string",
      "SigningProfileVersionArn": "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. 858)

A list of Lambda functions.

Type: Array of [FunctionConfiguration \(p. 974\)](#) objects

NextMarker (p. 858)

The pagination token that's included if more results are available.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

HTTP Status Code: 400

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded.

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. 862)

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. 862)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

MaxItems (p. 862)

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 \(p. 862\)](#)

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 \(p. 862\)](#)

The pagination token that's included if more results are available.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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](#)

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.

Request Syntax

```
GET /2018-10-31/layers?CompatibleRuntime=CompatibleRuntime&Marker=Marker&MaxItems=MaxItems
HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

CompatibleRuntime ([p. 864](#))

A runtime identifier. For example, go1.x.

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2

Marker ([p. 864](#))

A pagination token returned by a previous call.

MaxItems ([p. 864](#))

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": {
        "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 \(p. 864\)](#)

A list of function layers.

Type: Array of [LayersListItem \(p. 985\)](#) objects

[NextMarker \(p. 864\)](#)

A pagination token returned when the response doesn't contain all layers.

Type: String

Errors

InvalidArgumentException

One of the parameters in the request is invalid.

HTTP Status Code: 400

ServiceException

The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

The request throughput limit was exceeded.

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.

Request Syntax

```
GET /2018-10-31/layers/LayerName/versions?  
CompatibleRuntime=CompatibleRuntime&Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

CompatibleRuntime (p. 866)

A runtime identifier. For example, go1.x.

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2

LayerName (p. 866)

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. 866)

A pagination token returned by a previous call.

MaxItems (p. 866)

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": [  
        {  
            "CompatibleRuntimes": [ "string" ],  
            "LayerVersionArn": "string",  
            "LayerVersionId": "string",  
            "LastModified": "string",  
            "LayerVersionNumber": 123,  
            "Sha256": "string",  
            "Size": 123  
        }  
    ]  
}
```

```
        "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 \(p. 866\)](#)

A list of versions.

Type: Array of [LayerVersionsListItem \(p. 988\)](#) objects

[NextMarker \(p. 866\)](#)

A pagination token returned when the response doesn't contain all versions.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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 \(p. 869\)](#)

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 \(p. 869\)](#)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MaxItems \(p. 869\)](#)

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 \(p. 869\)](#)

The pagination token that's included if more results are available.

Type: String

[ProvisionedConcurrencyConfigs \(p. 869\)](#)

A list of provisioned concurrency configurations.

Type: Array of [ProvisionedConcurrencyConfigListItem \(p. 992\)](#) objects

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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 \(p. 807\)](#).

Request Syntax

```
GET /2017-03-31/tags/ARN HTTP/1.1
```

URI Request Parameters

The request uses the following URI parameters.

[ARN \(p. 872\)](#)

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 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 \(p. 872\)](#)

The function's tags.

Type: String to string map

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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. 874)

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. 874)

Specify the pagination token that's returned by a previous request to retrieve the next page of results.

[MaxItems](#) (p. 874)

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": [
```

```
{
  "CodeSha256": "string",
  "CodeSize": number,
  "DeadLetterConfig": {
    "TargetArn": "string"
  },
  "Description": "string",
  "Environment": {
    "Error": {
      "ErrorCode": "string",
      "Message": "string"
    },
    "Variables": {
      "string" : "string"
    }
  },
  "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",
  "SigningJobArn": "string",
  "SigningProfileVersionArn": "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 \(p. 874\)](#)

The pagination token that's included if more results are available.

Type: String

[Versions \(p. 874\)](#)

A list of Lambda function versions.

Type: Array of [FunctionConfiguration \(p. 974\)](#) objects

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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 \(p. 762\)](#) or [UpdateFunctionConfiguration \(p. 935\)](#).

Request Syntax

```
POST /2018-10-31/layers/LayerName/versions HTTP/1.1
Content-type: application/json

{
  "CompatibleRuntimes": [ "string" ],
  "Content": {
    "S3Bucket": "string",
    "S3Key": "string",
    "S3ObjectVersion": "string",
    "ZipFile": blob
  },
  "Description": "string",
  "LicenseInfo": "string"
}
```

URI Request Parameters

The request uses the following URI parameters.

LayerName (p. 878)

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.

CompatibleRuntimes (p. 878)

A list of compatible [function runtimes](#). Used for filtering with [ListLayers \(p. 864\)](#) and [ListLayerVersions \(p. 866\)](#).

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 | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2

Required: No

[Content \(p. 878\)](#)

The function layer archive.

Type: [LayerVersionContentInput \(p. 986\)](#) object

Required: Yes

[Description \(p. 878\)](#)

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

[LicenseInfo \(p. 878\)](#)

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

{
  "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.

[CompatibleRuntimes \(p. 879\)](#)

The layer's compatible runtimes.

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 | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2

[Content \(p. 879\)](#)

Details about the layer version.

Type: [LayerVersionContentOutput \(p. 987\)](#) object

[CreatedDate \(p. 879\)](#)

The date that the layer version was created, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[Description \(p. 879\)](#)

The description of the version.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[LayerArn \(p. 879\)](#)

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 \(p. 879\)](#)

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 \(p. 879\)](#)

The layer's software license.

Type: String

Length Constraints: Maximum length of 512.

[Version \(p. 879\)](#)

The version number.

Type: Long

Errors

CodeStorageExceededException

You have exceeded your maximum total code size per account. [Learn more](#)

HTTP Status Code: 400

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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 \(p. 926\)](#) or [UpdateFunctionConfiguration \(p. 935\)](#) to update the function before publishing a version.

Clients can invoke versions directly or with an alias. To create an alias, use [CreateAlias \(p. 746\)](#).

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 \(p. 882\)](#)

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 \(p. 882\)](#)

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 \(p. 926\)](#).

Type: String

Required: No

Description (p. 882)

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. 882)

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

{
    "CodeSha256": "string",
    "CodeSize": number,
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
    "Description": "string",
    "Environment": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "Variables": {
            "string" : "string"
        }
    },
    "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",
    "Layers": [
        {
            "Arn": "string",
            "CodeSize": number,
            "DependencyType": "string",
            "LayerName": "string",
            "Published": "2015-01-01T12:00:00Z",
            "Signature": "string",
            "SignatureType": "string",
            "Version": "string"
        }
    ],
    "MemorySize": number,
    "Role": "string",
    "Runtime": "string",
    "Timeout": number
}
```

```
"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",
"SigningJobArn": "string",
"SigningProfileVersionArn": "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.

CodeSha256 (p. 883)

The SHA256 hash of the function's deployment package.

Type: String

CodeSize (p. 883)

The size of the function's deployment package, in bytes.

Type: Long

DeadLetterConfig (p. 883)

The function's dead letter queue.

Type: [DeadLetterConfig](#) (p. 961) object

Description (p. 883)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[Environment \(p. 883\)](#)

The function's [environment variables](#).

Type: [EnvironmentResponse \(p. 965\)](#) object

[FileSystemConfigs \(p. 883\)](#)

Connection settings for an [Amazon EFS file system](#).

Type: Array of [FileSystemConfig \(p. 970\)](#) objects

Array Members: Maximum number of 1 item.

[FunctionArn \(p. 883\)](#)

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 \(p. 883\)](#)

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 \(p. 883\)](#)

The function that Lambda calls to begin executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

[ImageConfigResponse \(p. 883\)](#)

The function's image configuration values.

Type: [ImageConfigResponse \(p. 983\)](#) object

[KMSKeyArn \(p. 883\)](#)

The KMS key that's used to encrypt the function's environment variables. This key is only returned if you've configured a customer managed CMK.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-_\.]+:*)|()`

[LastModified \(p. 883\)](#)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[LastUpdateStatus \(p. 883\)](#)

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 \(p. 883\)](#)

The reason for the last update that was performed on the function.

Type: String

[LastUpdateStatusReasonCode \(p. 883\)](#)

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

[Layers \(p. 883\)](#)

The function's [layers](#).

Type: Array of [Layer \(p. 984\)](#) objects

[MasterArn \(p. 883\)](#)

For Lambda@Edge functions, the ARN of the master 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 \(p. 883\)](#)

The amount of memory available to the function at runtime.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 10240.

[PackageType \(p. 883\)](#)

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 \(p. 883\)](#)

The latest updated revision of the function or alias.

Type: String

[Role \(p. 883\)](#)

The function's execution role.

Type: String

Pattern: arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@-_/_]+

[Runtime \(p. 883\)](#)

The runtime environment for the Lambda function.

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2

[SigningJobArn \(p. 883\)](#)

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 \(p. 883\)](#)

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})?:(.*?)

[State \(p. 883\)](#)

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 \(p. 883\)](#)

The reason for the function's current state.

Type: String

[StateReasonCode \(p. 883\)](#)

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

[Timeout \(p. 883\)](#)

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 \(p. 883\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 997\)](#) object

[Version \(p. 883\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (`\$LATEST` | [0-9]+)

[VpcConfig \(p. 883\)](#)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 999\)](#) object

Errors

CodeStorageExceededException

You have exceeded your maximum total code size per account. [Learn more](#)

HTTP Status Code: 400

InvalidParameterValueException

One of the parameters in the request is invalid.

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 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.

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. 890)

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. 890)

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 \(p. 891\)](#)

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 \(p. 891\)](#)

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)|(-iso(b?)))?-([a-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 invalid.

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.

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 \(p. 807\)](#) to see the current setting for a function.

Use [GetAccountSettings \(p. 795\)](#) 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 [Managing Concurrency](#).

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 \(p. 893\)](#)

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 \(p. 893\)](#)

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 \(p. 894\)](#)

The number of concurrent executions that are reserved for this function. For more information, see [Managing Concurrency](#).

Type: Integer

Valid Range: Minimum value of 0.

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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 \(p. 946\)](#).

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 \(p. 935\)](#).

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

{
  "DestinationConfig": {
    "OnFailure": {
      "Destination": "string"
    },
    "OnSuccess": {
      "Destination": "string"
    }
  },
  "MaximumEventAgeInSeconds": number,
  "MaximumRetryAttempts": number
}
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName (p. 896)

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 \(p. 896\)](#)

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 \(p. 896\)](#)

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 an SQS queue.
- **Topic** - The ARN of an SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: [DestinationConfig \(p. 962\)](#) object

Required: No

[MaximumEventAgeInSeconds \(p. 896\)](#)

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 \(p. 896\)](#)

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": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    }
}
```

```
    },
    "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 \(p. 897\)](#)

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 an SQS queue.
- **Topic** - The ARN of an SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: [DestinationConfig \(p. 962\)](#) object

[FunctionArn \(p. 897\)](#)

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 \(p. 897\)](#)

The date and time that the configuration was last updated, in Unix time seconds.

Type: Timestamp

[MaximumEventAgeInSeconds \(p. 897\)](#)

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 \(p. 897\)](#)

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 invalid.

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.

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 \(p. 900\)](#)

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 \(p. 900\)](#)

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 \(p. 900\)](#)

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 \(p. 901\)](#)

The amount of provisioned concurrency allocated.

Type: Integer

Valid Range: Minimum value of 0.

[AvailableProvisionedConcurrentExecutions \(p. 901\)](#)

The amount of provisioned concurrency available.

Type: Integer

Valid Range: Minimum value of 0.

[LastModified \(p. 901\)](#)

The date and time that a user last updated the configuration, in [ISO 8601 format](#).

Type: String

[RequestedProvisionedConcurrentExecutions \(p. 901\)](#)

The amount of provisioned concurrency requested.

Type: Integer

Valid Range: Minimum value of 1.

[Status \(p. 901\)](#)

The status of the allocation process.

Type: String

Valid Values: IN_PROGRESS | READY | FAILED

[StatusReason \(p. 901\)](#)

For failed allocations, the reason that provisioned concurrency could not be allocated.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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.

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. 738).

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. 903)

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. 903)

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. 903)

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. 903)

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 invalid.

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 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.

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 account. You can get the ID of the statement from the output of [GetPolicy \(p. 834\)](#).

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 \(p. 905\)](#)

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 \(p. 905\)](#)

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 \(p. 905\)](#)

Only update the policy 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 \(p. 905\)](#)

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

InvalidArgumentException

One of the parameters in the request is invalid.

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 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.

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 \(p. 907\)](#)

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 \(p. 907\)](#)

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 invalid.

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.

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. 909)

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. 909)

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 invalid.

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.

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. 911)

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. 911)

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. 911)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

FunctionVersion (p. 911)

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. 911)

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. 911)

The routing configuration of the alias.

Type: AliasRoutingConfiguration (p. 955) 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 \(p. 912\)](#)

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 \(p. 912\)](#)

A description of the alias.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[FunctionVersion \(p. 912\)](#)

The function version that the alias invokes.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$\$LATEST|[0-9]+)`

[Name \(p. 912\)](#)

The name of the alias.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 128.

Pattern: `(?!^[\0-9]+\$)([a-zA-Z0-9-_]+)`

[RevisionId \(p. 912\)](#)

A unique identifier that changes when you update the alias.

Type: String

[RoutingConfig \(p. 912\)](#)

The [routing configuration](#) of the alias.

Type: [AliasRoutingConfiguration \(p. 955\)](#) object

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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 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.

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

{
  "AllowedPublishers": {
    "SigningProfileVersionArns": [ "string" ]
  },
  "CodeSigningPolicies": {
    "UntrustedArtifactOnDeployment": "string"
  },
  "Description": "string"
}
```

URI Request Parameters

The request uses the following URI parameters.

CodeSigningConfigArn (p. 915)

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. 915)

Signing profiles for this code signing configuration.

Type: [AllowedPublishers](#) (p. 956) object

Required: No

CodeSigningPolicies (p. 915)

The code signing policy.

Type: [CodeSigningPolicies](#) (p. 959) object

Required: No

Description (p. 915)

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 \(p. 916\)](#)

The code signing configuration

Type: [CodeSigningConfig \(p. 957\)](#) object

Errors

InvalidArgumentException

One of the parameters in the request is invalid.

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.

The following error handling options are only available 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.

Request Syntax

```
PUT /2015-03-31/event-source-mappings/UUID HTTP/1.1
Content-type: application/json

{
    "BatchSize": number,
    "BisectBatchOnFunctionError": boolean,
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "Enabled": boolean,
    "FunctionName": "string",
    "FunctionResponseTypes": [ "string" ],
    "MaximumBatchingWindowInSeconds": number,
    "MaximumRecordAgeInSeconds": number,
    "MaximumRetryAttempts": number,
    "ParallelizationFactor": number,
    "SourceAccessConfigurations": [
        {
            "Type": "string",
            "URI": "string"
        }
    ],
    "TumblingWindowInSeconds": number
}
```

URI Request Parameters

The request uses the following URI parameters.

UUID (p. 918)

The identifier of the event source mapping.

Required: Yes

Request Body

The request accepts the following data in JSON format.

[BatchSize \(p. 918\)](#)

The maximum number of items to retrieve in a single batch.

- **Amazon Kinesis** - Default 100. Max 10,000.
- **Amazon DynamoDB Streams** - Default 100. Max 1,000.
- **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.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

Required: No

[BisectBatchOnFunctionError \(p. 918\)](#)

(Streams only) If the function returns an error, split the batch in two and retry.

Type: Boolean

Required: No

[DestinationConfig \(p. 918\)](#)

(Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: [DestinationConfig \(p. 962\)](#) object

Required: No

[Enabled \(p. 918\)](#)

If true, the event source mapping is active. Set to false to pause polling and invocation.

Type: Boolean

Required: No

[FunctionName \(p. 918\)](#)

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 \(p. 918\)](#)

(Streams only) 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 \(p. 918\)](#)

(Streams and SQS standard queues) The maximum amount of time to gather records before invoking the function, in seconds.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

Required: No

[MaximumRecordAgeInSeconds \(p. 918\)](#)

(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 \(p. 918\)](#)

(Streams only) Discard records after the specified number of retries. The default value is infinite (-1). When set to infinite (-1), failed records will be retried until the record expires.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 10000.

Required: No

[ParallelizationFactor \(p. 918\)](#)

(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

[SourceAccessConfigurations \(p. 918\)](#)

An array of authentication protocols or VPC components required to secure your event source.

Type: Array of [SourceAccessConfiguration \(p. 995\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

Required: No

TumblingWindowInSeconds (p. 918)

(Streams only) The duration in seconds of a processing window. The range is between 1 second up to 900 seconds.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

Required: No

Response Syntax

```
HTTP/1.1 202
Content-type: application/json

{
    "BatchSize": number,
    "BisectBatchOnFunctionError": boolean,
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "string"
        },
        "OnSuccess": {
            "Destination": "string"
        }
    },
    "EventSourceArn": "string",
    "FunctionArn": "string",
    "FunctionResponseTypes": [ "string" ],
    "LastModified": number,
    "LastProcessingResult": "string",
    "MaximumBatchingWindowInSeconds": number,
    "MaximumRecordAgeInSeconds": number,
    "MaximumRetryAttempts": number,
    "ParallelizationFactor": number,
    "Queues": [ "string" ],
    "SelfManagedEventSource": {
        "Endpoints": {
            "string": [ "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.

BatchSize (p. 921)

The maximum number of items to retrieve in a single batch.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

BisectBatchOnFunctionError (p. 921)

(Streams only) If the function returns an error, split the batch in two and retry. The default value is false.

Type: Boolean

DestinationConfig (p. 921)

(Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: [DestinationConfig \(p. 962\)](#) object

EventSourceArn (p. 921)

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})?:(.*))?`

FunctionArn (p. 921)

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. 921)

(Streams only) 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. 921)

The date that the event source mapping was last updated, or its state changed, in Unix time seconds.

Type: Timestamp

LastProcessingResult (p. 921)

The result of the last AWS Lambda invocation of your Lambda function.

Type: String

MaximumBatchingWindowInSeconds (p. 921)

(Streams and SQS standard queues) The maximum amount of time to gather records before invoking the function, in seconds. The default value is zero.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

[MaximumRecordAgeInSeconds \(p. 921\)](#)

(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.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 604800.

[MaximumRetryAttempts \(p. 921\)](#)

(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 \(p. 921\)](#)

(Streams only) The number of batches to process from each shard concurrently. The default value is 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10.

[Queues \(p. 921\)](#)

(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]*

[SelfManagedEventSource \(p. 921\)](#)

The Self-Managed Apache Kafka cluster for your event source.

Type: [SelfManagedEventSource \(p. 994\)](#) object

[SourceAccessConfigurations \(p. 921\)](#)

An array of the authentication protocol, or the VPC components to secure your event source.

Type: Array of [SourceAccessConfiguration \(p. 995\)](#) objects

Array Members: Minimum number of 0 items. Maximum number of 22 items.

[StartingPosition \(p. 921\)](#)

The position in a stream from which to start reading. Required for Amazon Kinesis, Amazon DynamoDB, and Amazon MSK Streams sources. AT_TIMESTAMP is only supported for Amazon Kinesis streams.

Type: String

Valid Values: TRIM_HORIZON | LATEST | AT_TIMESTAMP

[StartingPositionTimestamp \(p. 921\)](#)

With StartingPosition set to AT_TIMESTAMP, the time from which to start reading, in Unix time seconds.

Type: Timestamp

[State \(p. 921\)](#)

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 \(p. 921\)](#)

Indicates whether the last change to the event source mapping was made by a user, or by the Lambda service.

Type: String

[Topics \(p. 921\)](#)

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\-_\.]+)

[TumblingWindowInSeconds \(p. 921\)](#)

(Streams only) The duration in seconds of a processing window. The range is between 1 second up to 900 seconds.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 900.

[UUID \(p. 921\)](#)

The identifier of the event source mapping.

Type: String

Errors

InvalidParameterValueException

One of the parameters in the request is invalid.

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 attempted to update an EventSource Mapping in CREATING, or tried to delete a EventSource mapping currently in the UPDATING state.

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.

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](#).

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

{
  "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. 926)

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.

[DryRun \(p. 926\)](#)

Set to true to validate the request parameters and access permissions without modifying the function code.

Type: Boolean

Required: No

[ImageUri \(p. 926\)](#)

URI of a container image in the Amazon ECR registry.

Type: String

Required: No

[Publish \(p. 926\)](#)

Set to true to publish a new version of the function after updating the code. This has the same effect as calling [PublishVersion \(p. 882\)](#) separately.

Type: Boolean

Required: No

[RevisionId \(p. 926\)](#)

Only update the function 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 \(p. 926\)](#)

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 \(p. 926\)](#)

The Amazon S3 key of the deployment package.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Required: No

[S3ObjectVersion \(p. 926\)](#)

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. 926)

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

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "CodeSha256": "string",
    "CodeSize": number,
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
    "Description": "string",
    "Environment": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "Variables": {
            "string" : "string"
        }
    },
    "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",
"SigningJobArn": "string",
"SigningProfileVersionArn": "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.

[CodeSha256 \(p. 928\)](#)

The SHA256 hash of the function's deployment package.

Type: String

[CodeSize \(p. 928\)](#)

The size of the function's deployment package, in bytes.

Type: Long

[DeadLetterConfig \(p. 928\)](#)

The function's dead letter queue.

Type: [DeadLetterConfig \(p. 961\)](#) object

[Description \(p. 928\)](#)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[Environment \(p. 928\)](#)

The function's [environment variables](#).

Type: [EnvironmentResponse \(p. 965\)](#) object

[FileSystemConfigs \(p. 928\)](#)

Connection settings for an [Amazon EFS file system](#).

Type: Array of [FileSystemConfig \(p. 970\)](#) objects

Array Members: Maximum number of 1 item.

[FunctionArn \(p. 928\)](#)

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 \(p. 928\)](#)

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 \(p. 928\)](#)

The function that Lambda calls to begin executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

[ImageConfigResponse \(p. 928\)](#)

The function's image configuration values.

Type: [ImageConfigResponse \(p. 983\)](#) object

[KMSKeyArn \(p. 928\)](#)

The KMS key that's used to encrypt the function's environment variables. This key is only returned if you've configured a customer managed CMK.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-_\.]+\.:*)|()`

[LastModified \(p. 928\)](#)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[LastUpdateStatus \(p. 928\)](#)

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 \(p. 928\)](#)

The reason for the last update that was performed on the function.

Type: String

[LastUpdateStatusReasonCode \(p. 928\)](#)

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`

[Layers \(p. 928\)](#)

The function's [layers](#).

Type: Array of [Layer \(p. 984\)](#) objects

[MasterArn \(p. 928\)](#)

For Lambda@Edge functions, the ARN of the master 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 \(p. 928\)](#)

The amount of memory available to the function at runtime.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 10240.

[PackageType \(p. 928\)](#)

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 \(p. 928\)](#)

The latest updated revision of the function or alias.

Type: String

[Role \(p. 928\)](#)

The function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\\-_/.+]`

[Runtime \(p. 928\)](#)

The runtime environment for the Lambda function.

Type: String

Valid Values: `nodejs` | `nodejs4.3` | `nodejs6.10` | `nodejs8.10` | `nodejs10.x` | `nodejs12.x` | `nodejs14.x` | `java8` | `java8.al2` | `java11` | `python2.7` | `python3.6` | `python3.7` | `python3.8` | `dotnetcore1.0` | `dotnetcore2.0` | `dotnetcore2.1`

| dotnetcore3.1 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided |
provided.al2

[SigningJobArn \(p. 928\)](#)

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 \(p. 928\)](#)

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})?:(.*)

[State \(p. 928\)](#)

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 \(p. 928\)](#)

The reason for the function's current state.

Type: String

[StateReasonCode \(p. 928\)](#)

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`

[Timeout \(p. 928\)](#)

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 \(p. 928\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 997\)](#) object

[Version \(p. 928\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: (`\$LATEST` | [0-9]+)

VpcConfig (p. 928)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 999\)](#) object

Errors

CodeSigningConfigNotFoundException

The specified code signing configuration does not exist.

HTTP Status Code: 404

CodeStorageExceededException

You have exceeded your maximum total code size per account. [Learn more](#)

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. Lambda always blocks deployment if the integrity check fails, even if code signing policy is set to WARN.

HTTP Status Code: 400

InvalidParameterValueException

One of the parameters in the request is invalid.

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 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.

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 \(p. 816\)](#) indicate when the update is complete and the function is processing events with the new configuration. For more information, see [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 \(p. 893\)](#). To grant invoke permissions to an account or AWS service, use [AddPermission \(p. 742\)](#).

Request Syntax

```
PUT /2015-03-31/functions/FunctionName/configuration HTTP/1.1
Content-type: application/json

{
  "DeadLetterConfig": {
    "TargetArn": "string"
  },
  "Description": "string",
  "Environment": {
    "Variables": {
      "string": "string"
    }
  },
  "FileSystemConfigs": [
    {
      "Arn": "string",
      "LocalMountPath": "string"
    }
  ],
  "Handler": "string",
  "ImageConfig": {
    "Command": [ "string" ],
    "EntryPoint": [ "string" ],
    "WorkingDirectory": "string"
  },
  "KMSKeyArn": "string",
  "Layers": [ "string" ],
  "MemorySize": number,
  "RevisionId": "string",
  "Role": "string",
  "Runtime": "string",
  "Timeout": number,
  "TracingConfig": {
    "Mode": "string"
  },
  "VpcConfig": {
    "SecurityGroupIds": [ "string" ],
    "SubnetIds": [ "string" ]
  }
}
```

URI Request Parameters

The request uses the following URI parameters.

FunctionName (p. 935)

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. 935)

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. 961) object

Required: No

Description (p. 935)

A description of the function.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

Required: No

Environment (p. 935)

Environment variables that are accessible from function code during execution.

Type: [Environment](#) (p. 963) object

Required: No

FileSystemConfigs (p. 935)

Connection settings for an Amazon EFS file system.

Type: Array of [FileSystemConfig](#) (p. 970) objects

Array Members: Maximum number of 1 item.

Required: No

[Handler \(p. 935\)](#)

The name of the method within your code that Lambda calls to execute your function. The format includes the file name. It can also include namespaces and other qualifiers, depending on the runtime. For more information, see [Programming Model](#).

Type: String

Length Constraints: Maximum length of 128.

Pattern: [^\s]+

Required: No

[ImageConfig \(p. 935\)](#)

Container image configuration values that override the values in the container image Dockerfile.

Type: [ImageConfig \(p. 981\)](#) object

Required: No

[KMSKeyArn \(p. 935\)](#)

The ARN of the AWS Key Management Service (AWS KMS) key that's used to encrypt your function's environment variables. If it's not provided, AWS Lambda uses a default service key.

Type: String

Pattern: (arn:(aws[a-zA-Z-]*)?:[a-z0-9-.]+:[.]*|()

Required: No

[Layers \(p. 935\)](#)

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 \(p. 935\)](#)

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 \(p. 935\)](#)

Only update the function 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 \(p. 935\)](#)

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 \(p. 935\)](#)

The identifier of the function's [runtime](#).

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2

Required: No

[Timeout \(p. 935\)](#)

The amount of time that Lambda allows a function to run before stopping it. The default is 3 seconds. The maximum allowed value is 900 seconds. For additional information, see [Lambda execution environment](#).

Type: Integer

Valid Range: Minimum value of 1.

Required: No

[TracingConfig \(p. 935\)](#)

Set `Mode` to `Active` to sample and trace a subset of incoming requests with [X-Ray](#).

Type: [TracingConfig \(p. 996\)](#) object

Required: No

[VpcConfig \(p. 935\)](#)

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 only access resources and the internet through that VPC. For more information, see [VPC Settings](#).

Type: [VpcConfig \(p. 998\)](#) object

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json
```

```
{
    "CodeSha256": "string",
    "CodeSize": number,
    "DeadLetterConfig": {
        "TargetArn": "string"
    },
    "Description": "string",
    "Environment": {
        "Error": {
            "ErrorCode": "string",
            "Message": "string"
        },
        "Variables": {
            "string" : "string"
        }
    },
    "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",
    "SigningJobArn": "string",
    "SigningProfileVersionArn": "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.

[CodeSha256 \(p. 938\)](#)

The SHA256 hash of the function's deployment package.

Type: String

[CodeSize \(p. 938\)](#)

The size of the function's deployment package, in bytes.

Type: Long

[DeadLetterConfig \(p. 938\)](#)

The function's dead letter queue.

Type: [DeadLetterConfig \(p. 961\)](#) object

[Description \(p. 938\)](#)

The function's description.

Type: String

Length Constraints: Minimum length of 0. Maximum length of 256.

[Environment \(p. 938\)](#)

The function's [environment variables](#).

Type: [EnvironmentResponse \(p. 965\)](#) object

[FileSystemConfigs \(p. 938\)](#)

Connection settings for an [Amazon EFS file system](#).

Type: Array of [FileSystemConfig \(p. 970\)](#) objects

Array Members: Maximum number of 1 item.

[FunctionArn \(p. 938\)](#)

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 \(p. 938\)](#)

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 \(p. 938\)](#)

The function that Lambda calls to begin executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

[ImageConfigResponse \(p. 938\)](#)

The function's image configuration values.

Type: [ImageConfigResponse \(p. 983\)](#) object

[KMSKeyArn \(p. 938\)](#)

The KMS key that's used to encrypt the function's environment variables. This key is only returned if you've configured a customer managed CMK.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-Z0-9-.]+:[^.]+|()`

[LastModified \(p. 938\)](#)

The date and time that the function was last updated, in [ISO-8601 format](#) (YYYY-MM-DDThh:mm:ss.sTZD).

Type: String

[LastUpdateStatus \(p. 938\)](#)

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 \(p. 938\)](#)

The reason for the last update that was performed on the function.

Type: String

[LastUpdateStatusReasonCode \(p. 938\)](#)

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`

[Layers \(p. 938\)](#)

The function's [layers](#).

Type: Array of [Layer \(p. 984\)](#) objects

MasterArn (p. 938)

For Lambda@Edge functions, the ARN of the master 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. 938)

The amount of memory available to the function at runtime.

Type: Integer

Valid Range: Minimum value of 128. Maximum value of 10240.

PackageType (p. 938)

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. 938)

The latest updated revision of the function or alias.

Type: String

Role (p. 938)

The function's execution role.

Type: String

Pattern: `arn:(aws[a-zA-Z-]*)?:iam::\d{12}:role/[a-zA-Z_0-9+=,.@\-_/.]+`

Runtime (p. 938)

The runtime environment for the Lambda function.

Type: String

Valid Values: `nodejs` | `nodejs4.3` | `nodejs6.10` | `nodejs8.10` | `nodejs10.x` | `nodejs12.x` | `nodejs14.x` | `java8` | `java8.al2` | `java11` | `python2.7` | `python3.6` | `python3.7` | `python3.8` | `dotnetcore1.0` | `dotnetcore2.0` | `dotnetcore2.1` | `dotnetcore3.1` | `nodejs4.3-edge` | `go1.x` | `ruby2.5` | `ruby2.7` | `provided` | `provided.al2`

SignedJobArn (p. 938)

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}):(.*)`

SignedProfileVersionArn (p. 938)

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})?:(.*)

[State \(p. 938\)](#)

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 \(p. 938\)](#)

The reason for the function's current state.

Type: String

[StateReasonCode \(p. 938\)](#)

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`

[Timeout \(p. 938\)](#)

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 \(p. 938\)](#)

The function's AWS X-Ray tracing configuration.

Type: [TracingConfigResponse \(p. 997\)](#) object

[Version \(p. 938\)](#)

The version of the Lambda function.

Type: String

Length Constraints: Minimum length of 1. Maximum length of 1024.

Pattern: `(\$LATEST|[0-9]+)`

[VpcConfig \(p. 938\)](#)

The function's networking configuration.

Type: [VpcConfigResponse \(p. 999\)](#) 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. Lambda always blocks deployment if the integrity check fails, even if code signing policy is set to WARN.

HTTP Status Code: 400

InvalidParameterValueException

One of the parameters in the request is invalid.

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 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.

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 \(p. 896\)](#).

Request Syntax

```
POST /2019-09-25/functions/FunctionName/event-invoke-config?Qualifier=Qualifier HTTP/1.1
Content-type: application/json

{
  "DestinationConfig": {
    "OnFailure": {
      "Destination": "string"
    },
    "OnSuccess": {
      "Destination": "string"
    }
  },
  "MaximumEventAgeInSeconds": number,
  "MaximumRetryAttempts": number
}
```

URI Request Parameters

The request uses the following URI parameters.

[FunctionName \(p. 946\)](#)

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 \(p. 946\)](#)

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 \(p. 946\)](#)

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 an SQS queue.
- **Topic** - The ARN of an SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: [DestinationConfig \(p. 962\)](#) object

Required: No

[MaximumEventAgeInSeconds \(p. 946\)](#)

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 \(p. 946\)](#)

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": {
      "Destination": "string"
    },
    "OnSuccess": {
      "Destination": "string"
    }
  },
  "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 \(p. 947\)](#)

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 an SQS queue.
- **Topic** - The ARN of an SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: [DestinationConfig \(p. 962\)](#) object

[FunctionArn \(p. 947\)](#)

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 \(p. 947\)](#)

The date and time that the configuration was last updated, in Unix time seconds.

Type: Timestamp

[MaximumEventAgeInSeconds \(p. 947\)](#)

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 \(p. 947\)](#)

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 invalid.

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.

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 \(p. 951\)](#)
- [AccountUsage \(p. 952\)](#)
- [AliasConfiguration \(p. 953\)](#)
- [AliasRoutingConfiguration \(p. 955\)](#)
- [AllowedPublishers \(p. 956\)](#)
- [CodeSigningConfig \(p. 957\)](#)
- [CodeSigningPolicies \(p. 959\)](#)
- [Concurrency \(p. 960\)](#)
- [DeadLetterConfig \(p. 961\)](#)
- [DestinationConfig \(p. 962\)](#)
- [Environment \(p. 963\)](#)
- [EnvironmentError \(p. 964\)](#)
- [EnvironmentResponse \(p. 965\)](#)
- [EventSourceMappingConfiguration \(p. 966\)](#)
- [FileSystemConfig \(p. 970\)](#)
- [FunctionCode \(p. 971\)](#)
- [FunctionCodeLocation \(p. 973\)](#)
- [FunctionConfiguration \(p. 974\)](#)
- [FunctionEventInvokeConfig \(p. 979\)](#)
- [ImageConfig \(p. 981\)](#)
- [ImageConfigError \(p. 982\)](#)
- [ImageConfigResponse \(p. 983\)](#)
- [Layer \(p. 984\)](#)

- [LayersListItem \(p. 985\)](#)
- [LayerVersionContentInput \(p. 986\)](#)
- [LayerVersionContentOutput \(p. 987\)](#)
- [LayerVersionsListItem \(p. 988\)](#)
- [OnFailure \(p. 990\)](#)
- [OnSuccess \(p. 991\)](#)
- [ProvisionedConcurrencyConfigListItem \(p. 992\)](#)
- [SelfManagedEventSource \(p. 994\)](#)
- [SourceAccessConfiguration \(p. 995\)](#)
- [TracingConfig \(p. 996\)](#)
- [TracingConfigResponse \(p. 997\)](#)
- [VpcConfig \(p. 998\)](#)
- [VpcConfigResponse \(p. 999\)](#)

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 \(p. 893\)](#).

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 \(p. 955\)](#) 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](#)

CodeSigningConfig

Details about a [Code signing configuration](#).

Contents

AllowedPublishers

List of allowed publishers.

Type: [AllowedPublishers \(p. 956\)](#) 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 \(p. 959\)](#) 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 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](#)

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-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 \(p. 990\)](#) object

Required: No

OnSuccess

The destination configuration for successful invocations.

Type: [OnSuccess \(p. 991\)](#) 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](#)

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 is successful, the response contains the environment variables. If it failed, the response contains details about the error.

Contents

Error

Error messages for environment variables that couldn't be applied.

Type: [EnvironmentError \(p. 964\)](#) object

Required: No

Variables

Environment variable key-value pairs.

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](#)

EventSourceMappingConfiguration

A mapping between an AWS resource and an AWS Lambda function. See [CreateEventSourceMapping \(p. 753\)](#) for details.

Contents

BatchSize

The maximum number of items to retrieve in a single batch.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10000.

Required: No

BisectBatchOnFunctionError

(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

(Streams only) An Amazon SQS queue or Amazon SNS topic destination for discarded records.

Type: [DestinationConfig \(p. 962\)](#) 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

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

(Streams only) 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 its state changed, in Unix time seconds.

Type: Timestamp

Required: No

LastProcessingResult

The result of the last AWS Lambda invocation of your Lambda function.

Type: String

Required: No

MaximumBatchingWindowInSeconds

(Streams and SQS standard queues) The maximum amount of time to gather records before invoking the function, in seconds. The default value is zero.

Type: Integer

Valid Range: Minimum value of 0. Maximum value of 300.

Required: No

MaximumRecordAgeInSeconds

(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.

Type: Integer

Valid Range: Minimum value of -1. Maximum value of 604800.

Required: No

MaximumRetryAttempts

(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

(Streams only) The number of batches to process from each shard concurrently. The default value is 1.

Type: Integer

Valid Range: Minimum value of 1. Maximum value of 10.

Required: No

Queues

(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

SelfManagedEventSource

The Self-Managed Apache Kafka cluster for your event source.

Type: [SelfManagedEventSource \(p. 994\)](#) object

Required: No

SourceAccessConfigurations

An array of the authentication protocol, or the VPC components to secure your event source.

Type: Array of [SourceAccessConfiguration \(p. 995\)](#) 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 Streams sources. AT_TIMESTAMP is only supported for Amazon Kinesis streams.

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 the last change to the event source mapping was made by a user, or by the Lambda service.

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: ^[^ .]([a-zA-Z0-9\-_ .]+)

Required: No

TumblingWindowInSeconds

(Streams only) The duration in seconds of a processing window. The range is between 1 second up to 900 seconds.

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-zA-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](#)

FunctionCode

The code for the Lambda function. You can specify either 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

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 \(p. 961\)](#) 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](#).

Type: [EnvironmentResponse \(p. 965\)](#) object

Required: No

FileSystemConfigs

Connection settings for an [Amazon EFS file system](#).

Type: Array of [FileSystemConfig \(p. 970\)](#) 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 executing your function.

Type: String

Length Constraints: Maximum length of 128.

Pattern: `[^\s]+`

Required: No

ImageConfigResponse

The function's image configuration values.

Type: [ImageConfigResponse \(p. 983\)](#) object

Required: No

KMSKeyArn

The KMS key that's used to encrypt the function's environment variables. This key is only returned if you've configured a customer managed CMK.

Type: String

Pattern: `(arn:(aws[a-zA-Z-]*)?:[a-zA-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` | `SubnetOutOfIPAddresses` |
`InvalidSubnet` | `InvalidSecurityGroup` | `ImageDeleted` | `ImageAccessDenied` |
`InvalidImage`

Required: No

Layers

The function's [layers](#).

Type: Array of [Layer \(p. 984\)](#) objects

Required: No

MasterArn

For Lambda@Edge functions, the ARN of the master 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 runtime environment for the Lambda function.

Type: String

Valid Values: nodejs | nodejs4.3 | nodejs6.10 | nodejs8.10 | nodejs10.x | nodejs12.x | nodejs14.x | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2

Required: No

SignedJobArn

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

SignedProfileVersionArn

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

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`

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 \(p. 997\)](#) 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 \(p. 999\)](#) 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 an SQS queue.
- **Topic** - The ARN of an SNS topic.
- **Event Bus** - The ARN of an Amazon EventBridge event bus.

Type: [DestinationConfig \(p. 962\)](#) 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](#)

ImageConfig

Configuration values that override the container image Dockerfile settings. See [Container 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 GetFunctionConfiguration request.

Contents

Error

Error response to GetFunctionConfiguration.

Type: [ImageConfigError \(p. 982\)](#) object

Required: No

ImageConfig

Configuration values that override the container image Dockerfile.

Type: [ImageConfig \(p. 981\)](#) 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 \(p. 988\)](#) 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

CompatibleRuntimes

The layer's compatible runtimes.

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 | java8 | java8.al2 | java11 | python2.7 | python3.6 | python3.7 | python3.8 | dotnetcore1.0 | dotnetcore2.0 | dotnetcore2.1 | dotnetcore3.1 | nodejs4.3-edge | go1.x | ruby2.5 | ruby2.7 | provided | provided.al2

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.

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](#)

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](#)

SourceAccessConfiguration

You can specify the authentication protocol, or the VPC components to secure access to your event source.

Contents

Type

The type of authentication protocol or the VPC components for your event source. For example:

"Type" : "SASL_SCRAM_512_AUTH".

- **BASIC_AUTH** - (MQ) The 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** - The subnets associated with your VPC. Lambda connects to these subnets to fetch data from your Kafka cluster.
- **VPC_SECURITY_GROUP** - The VPC security group used to manage access to your Kafka brokers.
- **SASL_SCRAM_256_AUTH** - The Secrets Manager ARN of your secret key used for SASL SCRAM-256 authentication of your Kafka brokers.
- **SASL_SCRAM_512_AUTH** - The Secrets Manager ARN of your secret key used for SASL SCRAM-512 authentication of your Kafka brokers.

Type: String

Valid Values: **BASIC_AUTH** | **VPC_SUBNET** | **VPC_SECURITY_GROUP** |
SASL_SCRAM_512_AUTH | **SASL_SCRAM_256_AUTH**

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 [VPC Settings](#).

Contents

SecurityGroupIds

A list of VPC security groups 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 groups 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](#).

AWS glossary

For the latest AWS terminology, see the [AWS glossary](#) in the *AWS General Reference*.