**AWS Lambda**

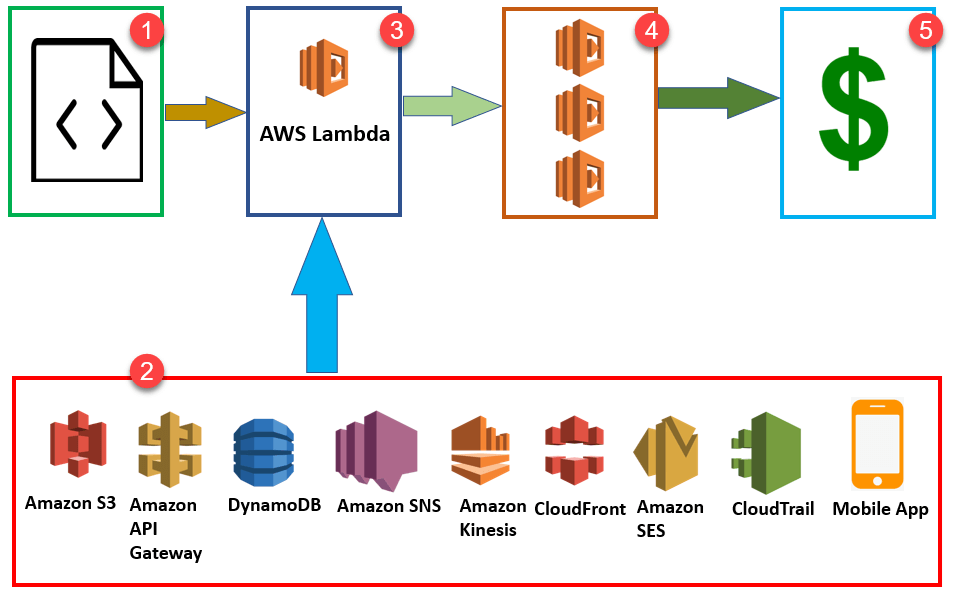
# What Is AWS Lambda?

* **What is Serverless:** Serverless is a term that generally refers to serverless applications. Serverless applications are ones that don't need any server provision and do not require to manage servers.
* AWS Lambda is a compute service that lets run code without provisioning or managing servers.
* AWS Lambda executes your code only when needed and scales automatically, from a few requests per day to thousands per second.
* Pay only for the compute time you consume - there is no charge when code is not running.
* With AWS Lambda, you can run code for virtually any type of application or backend service - all with zero administration.
* AWS 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. i.e Therefore you don't need to worry about which AWS resources to launch, or how will you manage them. Instead, you need to put the code on Lambda, and it runs.
* AWS Lambda supports multiple languages through the use of runtimes: Go, Java, Python, NodeJs, Ruby and .NET Core.
* We can use AWS Lambda to run our code in response to events, such as changes to data in an Amazon S3 bucket or an Amazon DynamoDB table; to run your code in response to HTTP requests using Amazon API Gateway; or invoke your code using API calls made using AWS SDKs.
* With these capabilities, we can use Lambda to easily build data processing triggers for AWS services like Amazon S3 and Amazon DynamoDB, process streaming data stored in Kinesis, or create your own back end that operates at AWS scale, performance, and security.
* AWS Lambda helps you to focus on your core product and business logic instead of managing operating system (OS) access control, OS patching, right-sizing, provisioning, scaling, etc.
* With AWS Lambda, you are charged for every 100ms your code executes and the number of times your code is triggered. You pay only for the compute time you consume.

## **1.1 When Should I Use AWS Lambda?**

* When using AWS Lambda, you are responsible only for your code.
* AWS Lambda manages the compute fleet that offers a balance of memory, CPU, network, and other resources. This is in exchange for flexibility, which means you cannot log in to compute instances, or customize the operating system on provided runtimes.
* These constraints enable AWS Lambda to perform operational and administrative activities on your behalf, including provisioning capacity, monitoring fleet health, applying security patches, deploying your code, and monitoring and logging your Lambda functions.

## **1.2 How does AWS Lambda work?**



**Step 1:** First upload your AWS Lambda code in any language supported by AWS Lambda. Java, Python, Go, and C# are some of the languages that are supported by AWS lambda.

**Step 2:** These are some AWS services which allow you to trigger AWS Lambda.

**Step 3:** AWS Lambda helps you to upload code and the event details on which it should be triggered.

**Step 4:** Executes AWS Lambda Code when it is triggered by AWS services:

**Step 5:** AWS charges only when the AWS lambda code executes, and not otherwise.

This will happen in the following scenarios:

* Upload files in an S3 bucket
* When HTTP get/post endpoint URL is hit
* For adding/modifying and deleting Dynamo DB tables
* In the process of data streams collection
* Push notification
* Hosting of website
* Email sending

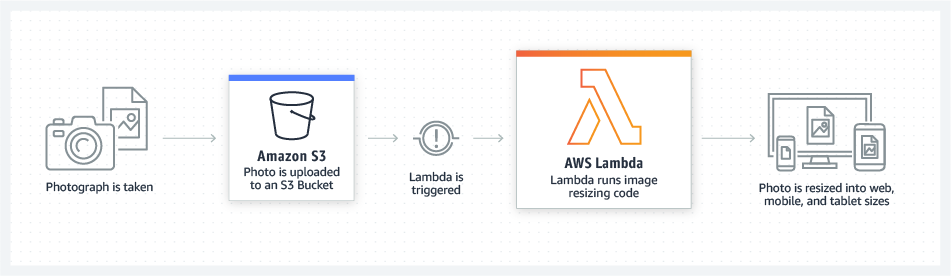
## **1.2.1 Use cases**

**A) Data processing**

You can use AWS Lambda to execute code in response to triggers such as changes in data, shifts in system state, or actions by users. Lambda can be directly triggered by AWS services such as S3, DynamoDB, Kinesis, SNS, and CloudWatch, or it can be orchestrated into workflows by AWS Step Functions. This allows you to build a variety of real-time serverless data processing systems.

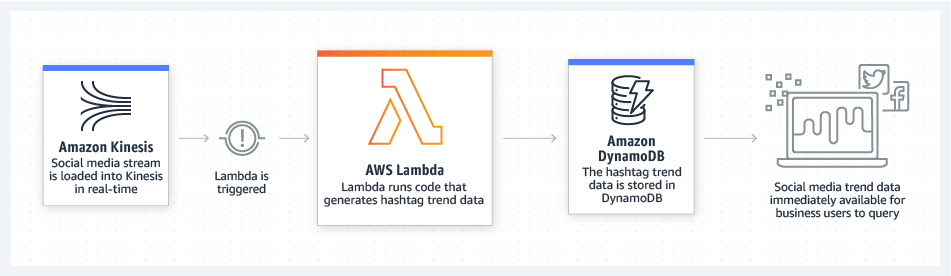
**A.1 REAL-TIME FILE PROCESSING**

You can use Amazon S3 to trigger AWS Lambda to process data immediately after an upload. For example, you can use Lambda to thumbnail images, transcode videos, index files, process logs, validate content, and aggregate and filter data in real-time.



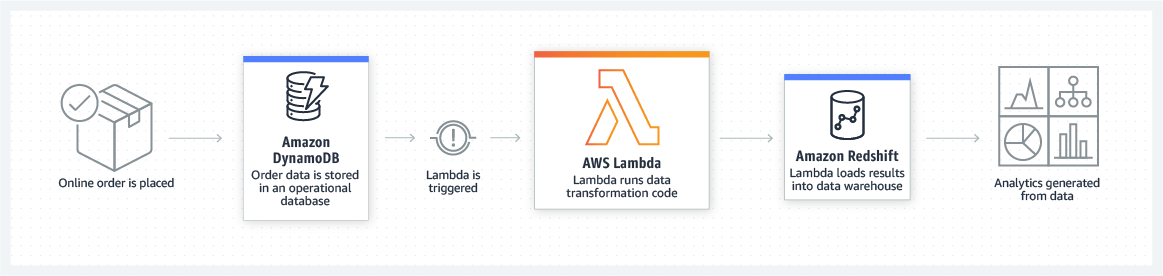
**A.2 REAL-TIME STREAM PROCESSING**

You can use AWS Lambda and Amazon Kinesis to process real-time streaming data for application activity tracking, transaction order processing, click stream analysis, data cleansing, metrics generation, log filtering, indexing, social media analysis, and IoT device data telemetry and metering.



**A.3 EXTRACT, TRANSFORM, LOAD**

You can use AWS Lambda to perform data validation, filtering, sorting, or other transformations for every data change in a DynamoDB table and load the transformed data to another data store.

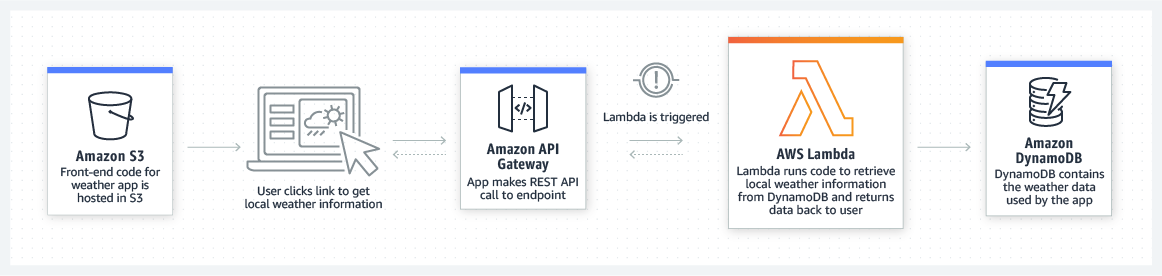


**B. Backends**

You can build serverless backends using AWS Lambda to handle web, mobile, Internet of Things (IoT), and 3rd party API requests. Take advantage of Lambda’s consistent performance controls, such as multiple memory configurations and Provisioned Concurrency, for building latency-sensitive applications at any scale.

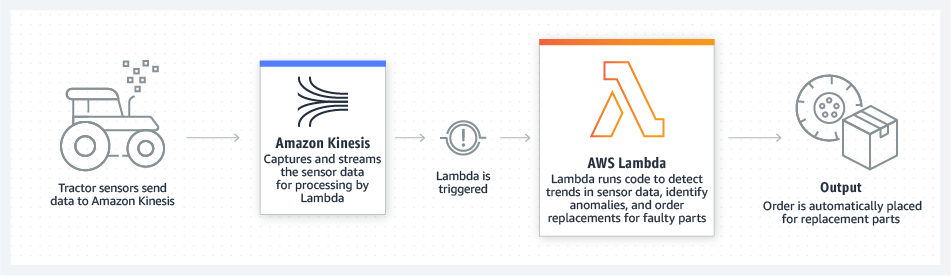
**B.1 WEB APPLICATIONS**

By combining AWS Lambda with other AWS services, developers can build powerful web applications that automatically scale up and down and run in a highly available configuration across multiple data centers – with zero administrative effort required for scalability, back-ups or multi-data center redundancy.



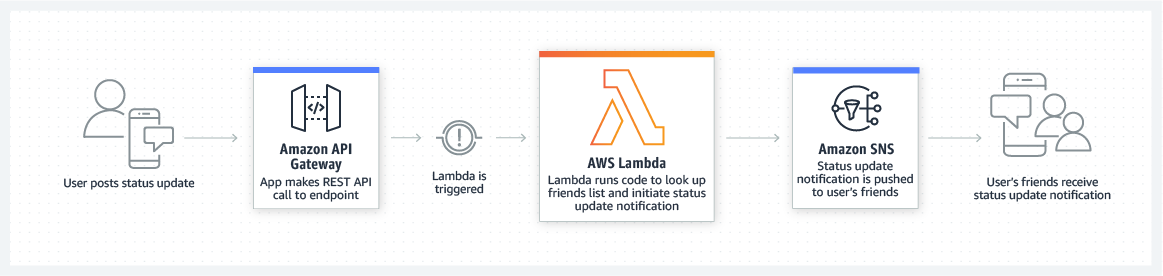
**B.2 IOT BACKENDS**

You can build serverless backends using AWS Lambda to handle web, mobile, Internet of Things (IoT), and 3rd party API requests.



**B.3 MOBILE BACKENDS**

AWS Lambda makes it easy to create rich, personalized app experiences. You can build backends using AWS Lambda and Amazon API Gateway to authenticate and process API requests. Use AWS Amplify to easily integrate your backend with your iOS, Android, Web, and React Native frontends.



# AWS Lambda Concepts

**Function:**

A function is a program or a script which runs in AWS Lambda. Lambda passes invocation events into your function, which processes an event and returns its response.

**Runtimes:**

Runtime allows functions in various languages which runs on the same base execution environment. This helps you to configure your function in runtime. It also matches your selected programming language.

**Event source:**

* An event is a JSON formatted document that contains data for a function to process. The Lambda 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.
* When an AWS service invokes your function, the service defines the shape of the event.
* An event source is an AWS service, such as Amazon SNS, or a custom service. This triggers function helps you to executes its logic.

**Trigger**

* A trigger is a resource or configuration that invokes a Lambda function. This includes AWS services that can be configured to invoke a function, applications that you develop, and event source mappings. An event source mapping is a resource in Lambda that reads items from a stream or queue and invokes a function.

**Lambda Layers:**

Lambda layers are an important distribution mechanism for libraries, custom runtimes, and other important function dependencies. This AWS component also helps you to manage your development function code separately from the unchanging code and resources that it uses.

**Log streams:**

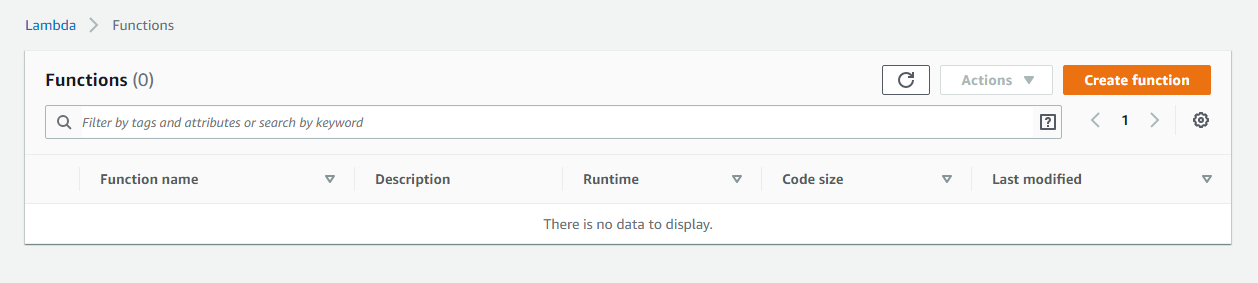
Log stream allows you to annotate your function code with custom logging statements which helps you to analyse the execution flow and performance of your Lambda functions.

# Create a Lambda Function

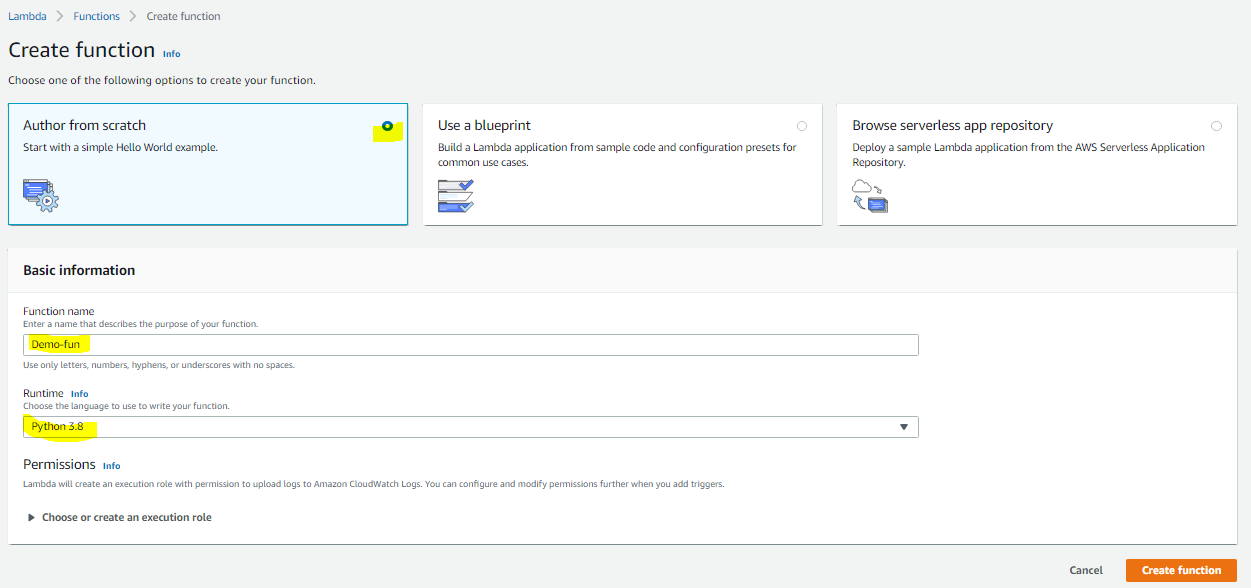
Create a Lambda function using the AWS Lambda console. AWS Lambda executes the Lambda function and returns results. You then verify execution results, including the logs that your Lambda function created and various CloudWatch metrics.

## **Step-1: To create a Lambda function**

1. Open the [AWS Lambda console](https://console.aws.amazon.com/lambda/home).
2. Choose **Create a function**.



1. For **Function name**, enter **my-function**.
2. Choose **Create function**.

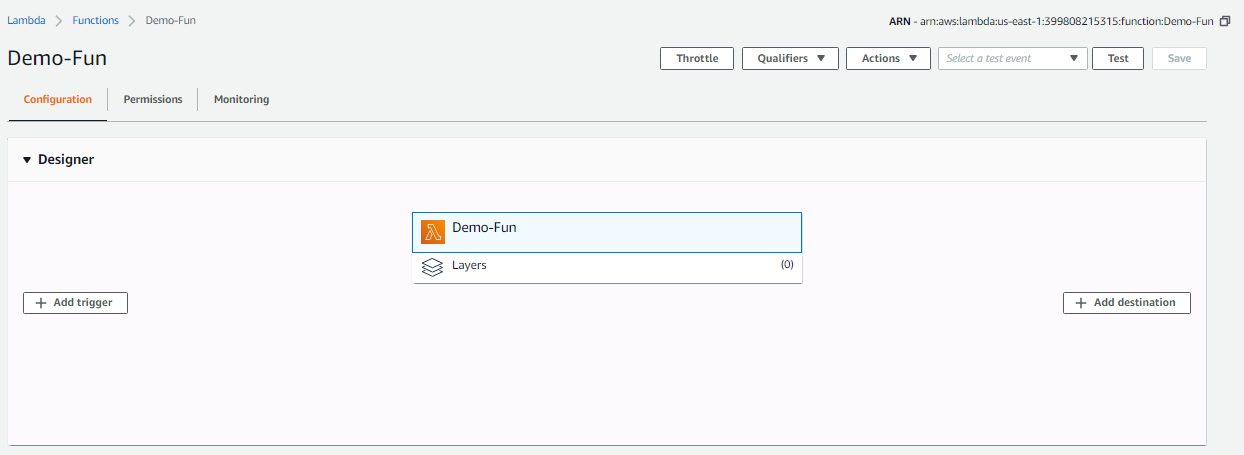


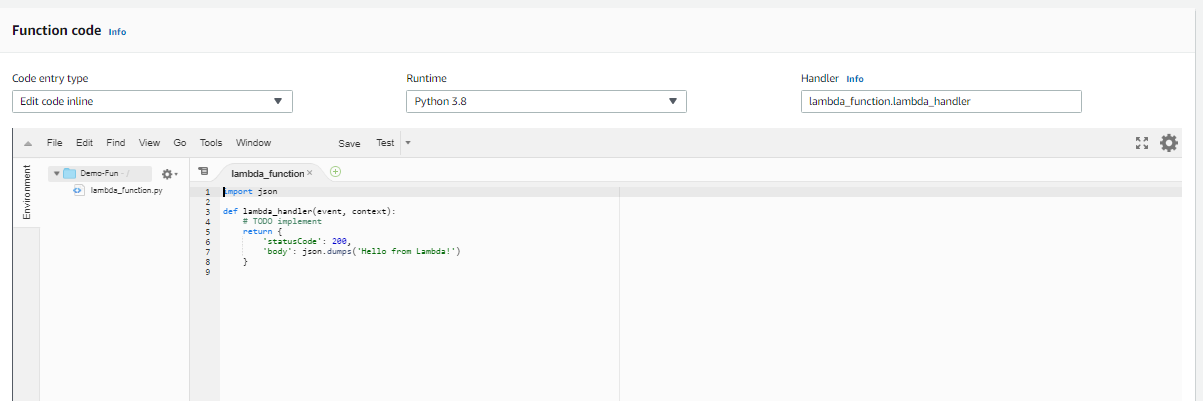
Lambda creates a Python function and an execution role that grants the function permission to upload logs.

**Use the Designer**

The Designer shows an overview of your function and its upstream and downstream resources. You can use it to configure triggers, layers, and destinations.


        A Lambda function with an Amazon S3 trigger and Amazon EventBridge destination.
      



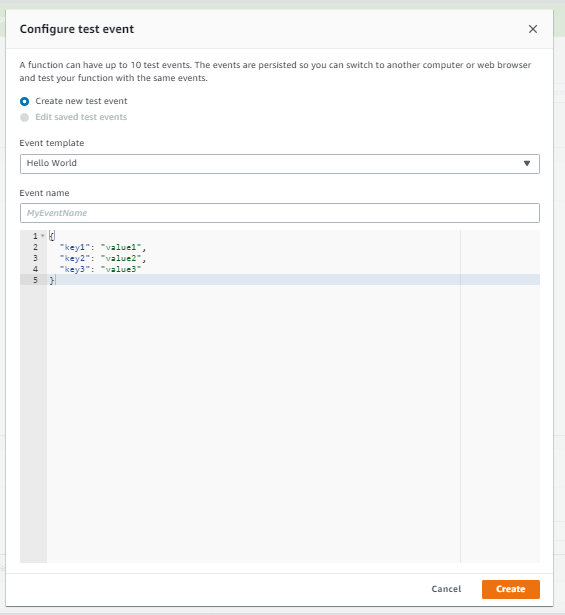


Choose my-function in the designer to return to the function's code and configuration. For scripting languages, Lambda includes sample code that returns a success response.

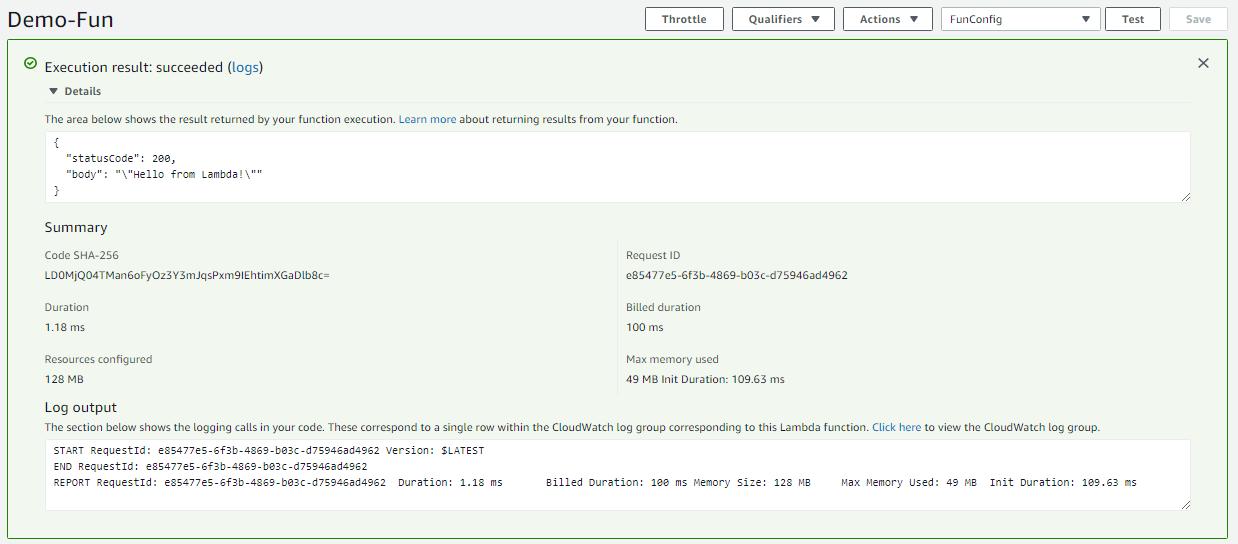
## **Step-2: Invoke the Lambda Function**

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

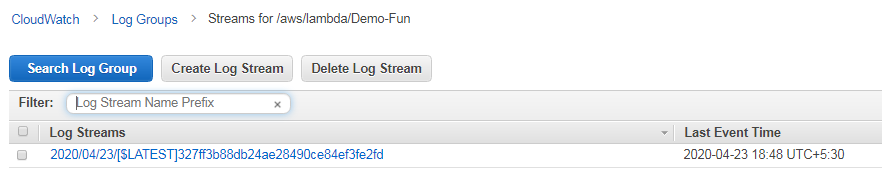
1. In the upper right corner, choose **Test**.
2. In the **Configure test event** page, choose **Create new test event** and in **Event template**, leave the default **Hello World** option. Enter an **Event name** and note the following sample event template:
3. Choose **Create** and then choose **Test**. Each user can create up to 10 test events per function. Those test events are not available to other users.

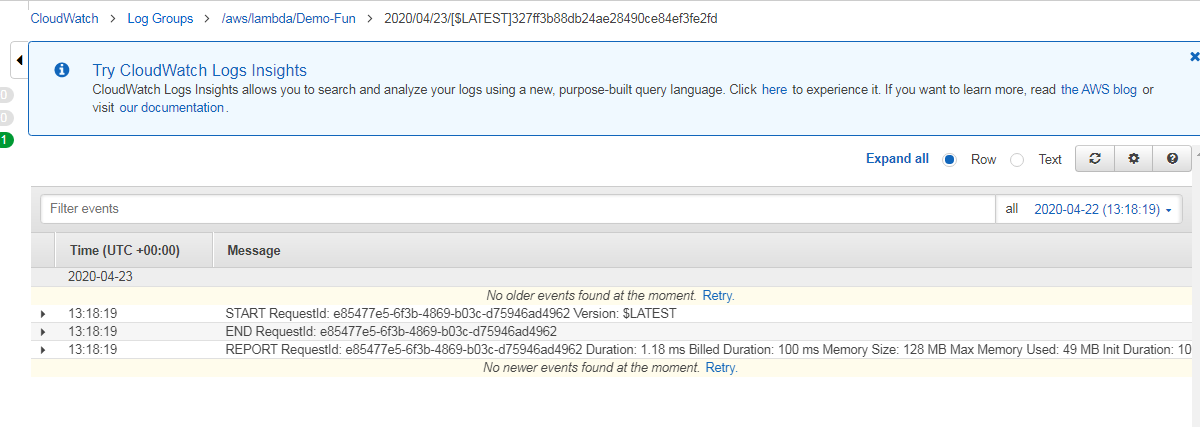


1. AWS Lambda executes your function on your behalf. The handler in your Lambda function receives and then processes the sample event.
2. Upon successful execution, view results in the console.
   1. The **Execution result** section shows the execution status as **succeeded** and also shows the function execution results, returned by the return statement.
   2. The **Summary** section shows the key information reported in the **Log output** section (the *REPORT* line in the execution log).
   3. The **Log output** section shows the log AWS Lambda generates for each execution. These are the logs written to CloudWatch by the Lambda function. The AWS Lambda console shows these logs for your convenience.

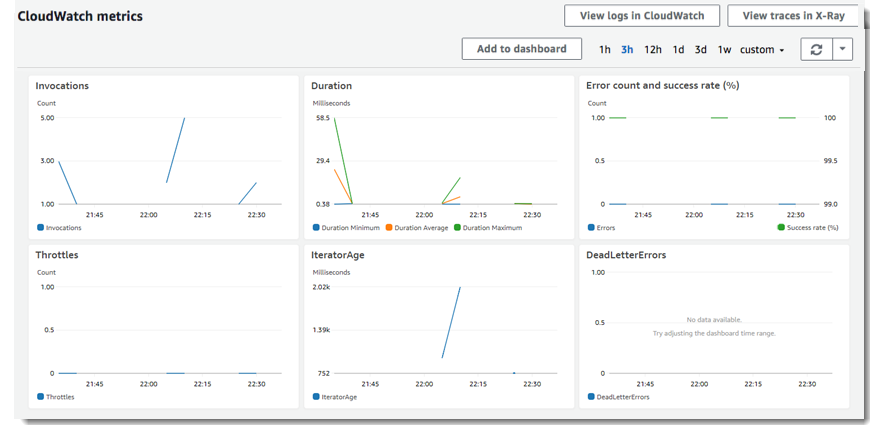


Note that the **Click here** link shows logs in the CloudWatch console. The function then adds logs to Amazon CloudWatch in the log group that corresponds to the Lambda function.





1. Run the Lambda function a few times to gather some metrics that you can view in the next step.
2. Choose **Monitoring**. This page shows graphs for the metrics that Lambda sends to CloudWatch.



**Lambda Monitoring Graphs**

* **Invocations** – The number of times that the function was invoked in each 5-minute period.
* **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.

# AWS Lambda Features

AWS 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.

## **4.1 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.
* 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.
* Initialization is billed as part of the duration for the first invocation processed by an instance of your function. When X-Ray tracing is enabled, the runtime records separate subsegments for initialization and execution.
* 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 execution 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.
* 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 the size of your deployment package and the amount of data that you transfer onto the execution environment to a minimum.

## **4.2 Deployment Package**

Function's code consists of scripts or compiled programs and their dependencies. When you author functions in the Lambda console or a toolkit, the client creates a ZIP archive of your code called a deployment package. The client then sends the package to the Lambda service. When you manage functions with the Lambda API, command line tools, or SDKs, you create the deployment package. You also need to create a deployment package manually for compiled languages and to add dependencies to your function.

## **4.3 Layers**

* Lambda layers are a distribution mechanism for libraries, custom runtimes, and other function dependencies.
* Layers let you 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 provided by AWS, or layers from other AWS customers.

## **4.4 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.

## **4.5 Concurrency Controls**

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

To enable functions to scale without fluctuations in latency, use provisioned concurrency. For functions that take a long time to initialize, or 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.

## **4.6 Asynchronous Invocation**

When you invoke a function, you can choose to invoke it synchronously or asynchronously. With synchronous invocation, 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.


        Lambda queues asynchronous invocation events before sending them to the function.
      

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.

## **4.7 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 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.


        An event source mapping reading records from a 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.

## **4.8 Destinations**

A destination is an AWS resource that receives invocation records for a function. For asynchronous invocation, 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.


        Lambda sends invocation records to a queue or event bus destination, depending on the result.
      

For event source mappings 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.

## **4.9 Function Blueprints**

When you create a function in the Lambda console, you can choose to start from scratch, use a blueprint, 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.

## **4.10 Application Templates**

By using 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.

# AWS Lambda Limits

AWS Lambda limits the amount of compute and storage resources that you can use to run and store functions.

|  |  |
| --- | --- |
| **Resource** | **Default Limit** |
| Concurrent executions | 1,000 |
| Function and layer storage | 75 GB |
| [Elastic network interfaces per VPC](https://docs.aws.amazon.com/lambda/latest/dg/configuration-vpc.html) | 250 |

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

|  |  |
| --- | --- |
| **Resource** | **Limit** |
| Function [memory allocation](https://docs.aws.amazon.com/lambda/latest/dg/configuration-console.html) | 128 MB to 3,008 MB, in 64 MB increments. |
| Function [timeout](https://docs.aws.amazon.com/lambda/latest/dg/configuration-console.html) | 900 seconds (15 minutes) |
| Function [environment variables](https://docs.aws.amazon.com/lambda/latest/dg/configuration-envvars.html) | 4 KB |
| Function [resource-based policy](https://docs.aws.amazon.com/lambda/latest/dg/access-control-resource-based.html) | 20 KB |
| Function [layers](https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html) | 5 layers |
| Function [burst concurrency](https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html) | 500 - 3000 ([varies per region](https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html)) |
| Invocation frequency per Region (requests per second) | 10 x concurrent executions limit ([synchronous](https://docs.aws.amazon.com/lambda/latest/dg/invocation-sync.html) – all sources)  10 x concurrent executions limit ([asynchronous](https://docs.aws.amazon.com/lambda/latest/dg/invocation-async.html) – non-AWS sources)  Unlimited (asynchronous – [AWS service sources](https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html)) |
| Invocation frequency per function version or alias (requests per second) | 10 x allocated [provisioned concurrency](https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html)  This limit only applies to functions that use provisioned concurrency. |
| [Invocation payload](https://docs.aws.amazon.com/lambda/latest/dg/lambda-invocation.html) (request and response) | 6 MB (synchronous)  256 KB (asynchronous) |
| [Deployment package](https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-features.html#gettingstarted-features-package) size | 50 MB (zipped, for direct upload)  250 MB (unzipped, including layers)  3 MB (console editor) |
| Test events (console editor) | 10 |
| /tmp directory storage | 512 MB |
| File descriptors | 1,024 |
| Execution processes/threads | 1,024 |

# AWS Lambda Permissions

* AWS 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,** 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, 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 to trigger your function.
* Use **resource-based policies** 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, or write your own. The Lambda console uses multiple services to get information about your function's configuration and triggers. We can use the managed policies as-is, or as a starting point for more restrictive policies.
* We 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.

## **6.1 AWS Lambda execution role**

* An AWS Lambda function's execution role grants it 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 upload trace data to AWS X-Ray.