# WHAT IS AWS LAMBDA?

Recently, Lambda has become popular thanks to AWS, but many folks still use Lambda and serverless interchangeably. In this post, we are going to shed some light on AWS Lambda, including how it ties into serverless architecture, how to create it, and when to use it.

## Understanding serverless

To understand what AWS Lambda is, we have to first understand what serverless architecture is all about. Serverless applications in general are applications that don't require any provisioning of servers for them to run. When you run a serverless application, you get the benefit of not worrying about OS setup, patching, or scaling of servers that you would have to consider when you run your application on a physical server.

Serverless applications or platforms have four characteristics:

1. No server management
2. Flexible scaling
3. No idle capacity
4. High availability

Serverless applications have several components and layers:

- Compute
- API
- Storage
- Interprocess Messaging

- Orchestration

As a cloud provider, AWS offers services that can be used for each of these components that make up a serverless architecture. This is where AWS Lambda comes in.

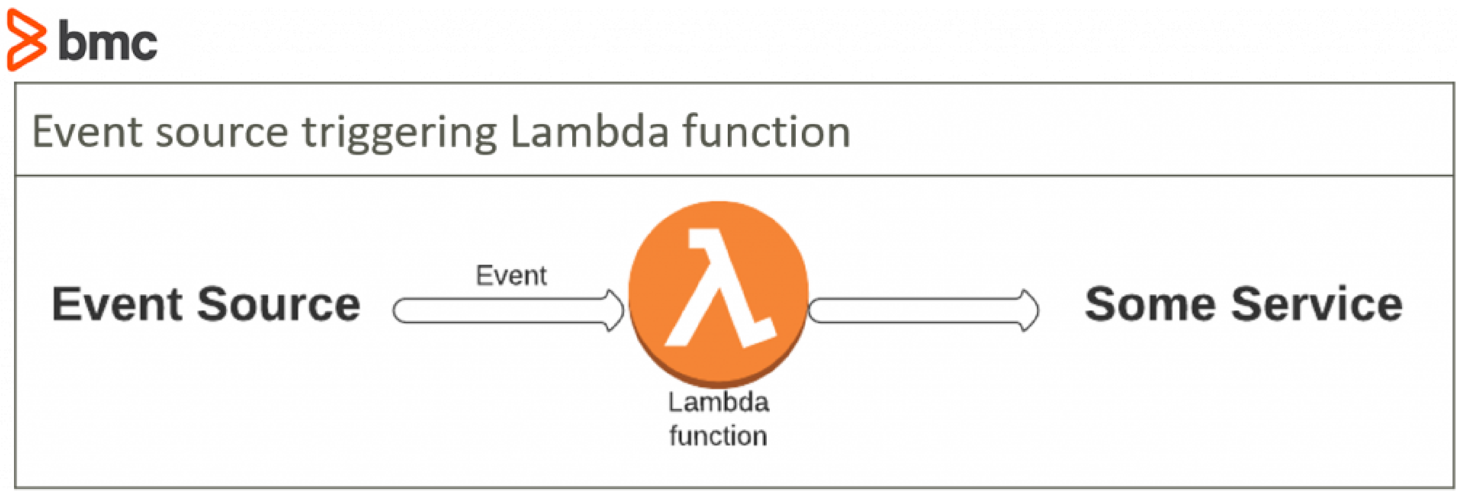
## Describing AWS Lambda

AWS Lambda service is a high-scale, provision-free serverless compute offering based on functions. It is used *only* for the compute layer of a serverless application. The purpose of AWS Lambda is to build event-driven applications that can be triggered by several events in AWS.

In the case where you have multiple simultaneous events, Lambda simply spins up multiple copies of the function to handle the events. In other words, Lambda can be described as a type of function as a service (FaaS). Three components comprise AWS Lambda:

- **A function.** This is the actual code that performs the task.
- **A configuration.** This specifies how your function is executed.
- **An event source (optional).** This is the event that triggers the function. You can trigger with several AWS services or a third-party service.

When you specify an event source, your function is invoked when an event from that source occurs. The diagram below shows what this looks like:



## Running a Lambda function

When configuring a lambda function, you specify which **runtime environment** you'd like to run your code in. Depending on the language you use, each environment provides its own set of binaries available for use in your code. You are also allowed to package any libraries or binaries you like as long as you can use them within the runtime environment. All environments are based on Amazon Linux AMI.

The current available runtime environments are:

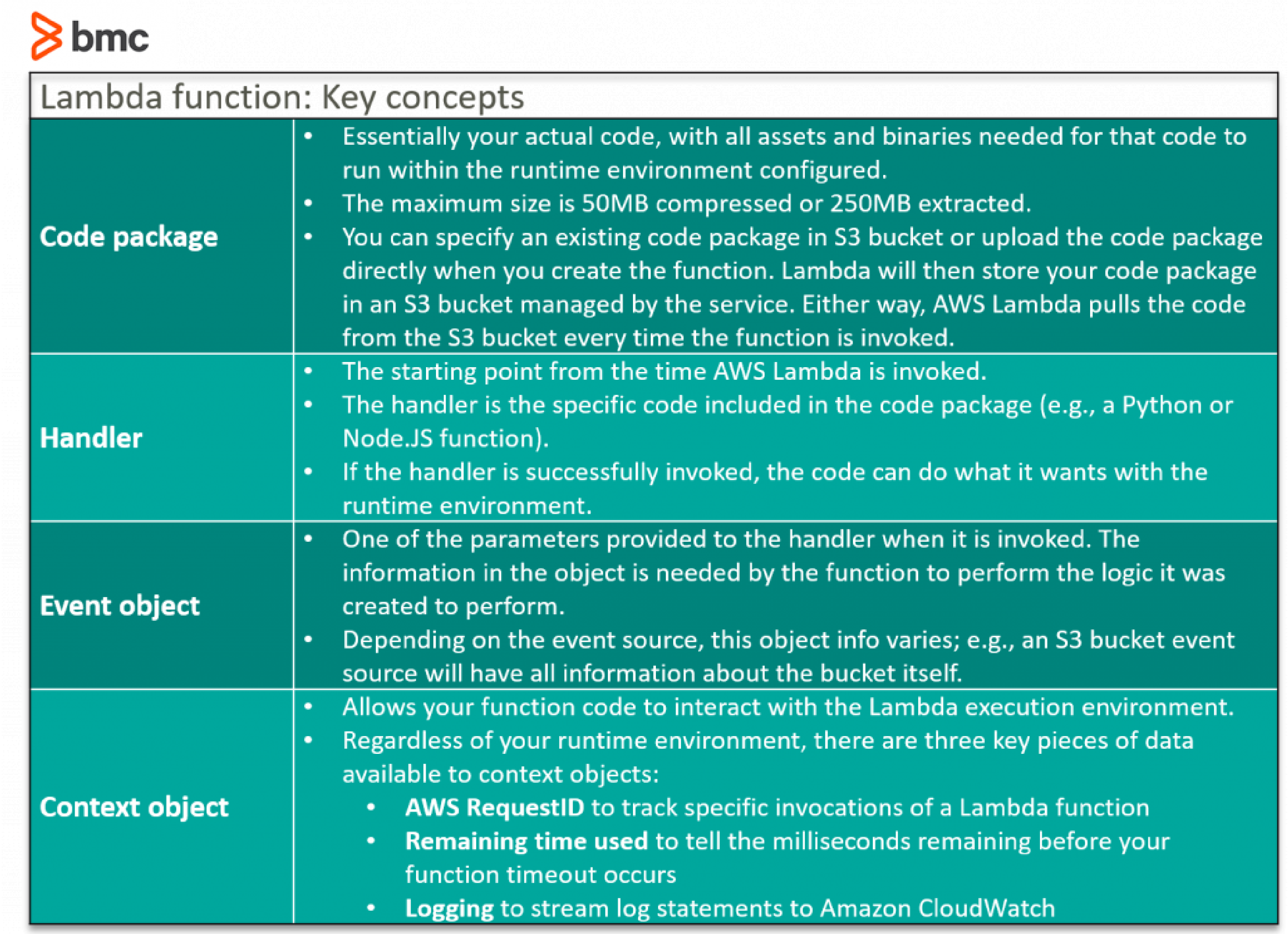
- nodeJS
- Python
- Go
- Java

- Ruby
- .Net
- C#

When running a lambda function, we only focus on the code because AWS manages capacity and all updates. AWS Lambda can be invoked synchronously using the **ResponseRequest InvocationType** and asynchronously using the **EventInvocationType**.

## Concepts of Lambda function

To better understand how lambda function works, there are key concepts to understand.



Lambda function: Key concepts

| | |
|---|---|
| **Code package** | • Essentially your actual code, with all assets and binaries needed for that code to run within the runtime environment configured.<br>• The maximum size is 50MB compressed or 250MB extracted.<br>• You can specify an existing code package in S3 bucket or upload the code package directly when you create the function. Lambda will then store your code package in an S3 bucket managed by the service. Either way, AWS Lambda pulls the code from the S3 bucket every time the function is invoked. |
| **Handler** | • The starting point from the time AWS Lambda is invoked.<br>• The handler is the specific code included in the code package (e.g., a Python or Node.JS function).<br>• If the handler is successfully invoked, the code can do what it wants with the runtime environment. |
| **Event object** | • One of the parameters provided to the handler when it is invoked. The information in the object is needed by the function to perform the logic it was created to perform.<br>• Depending on the event source, this object info varies; e.g., an S3 bucket event source will have all information about the bucket itself. |
| **Context object** | • Allows your function code to interact with the Lambda execution environment.<br>• Regardless of your runtime environment, there are three key pieces of data available to context objects:<br>  • **AWS RequestID** to track specific invocations of a Lambda function<br>  • **Remaining time used** to tell the milliseconds remaining before your function timeout occurs<br>  • **Logging** to stream log statements to Amazon CloudWatch |

## Event source

Although AWS Lambda can be triggered using the Invoke API, the recommended way of triggering lambda is through event sources from within AWS.

There are two models of invocation supported:

(a) **Push** which get triggered by other events such as API gateway, new object in S3 or Amazon Alexa.

(b) **Pull** where the lambda function goes and poll an event source for new objects. Examples of such event sources are DynamoDB or Amazon Kinesis.

## Lambda configuration

There are few configuration settings that can be used with lambda functions:

- **Memory dial** which controls not just the memory but also affects how much CPU and network resources is allocated to the function.
- **Version/Aliases** are used to revert function back to older versions. This is also key in implementing a deployment strategy such as blue/green or separating production from lower environments.
- **IAM Role** gives the lambda function permission to interact with other AWS services and APIs.
- **Lambda function permission** defines which push model event source is allowed to invoke the lambda function.
- **Network configuration for outbound connectivity**. There are two choices:
    - *Default* which allows internet connectivity but no connectivity to private resources in your VPC services
    - *VPC* which allows your function to be provisioned inside your VPC and use an ENI. You can then attach things like security like you would any other ENIs.
- **Environment variables** for dynamically injecting values that are consumed by code. This idea of separating code from config is one of the 12-factor app methodology around cloud native applications.
- **Dead letter queue** is where you send all failed invocation events. This can either be SNS topic or SQS

**Timeouts** which is the allowed amount of time a function is allowed to run before it is timed out.

## Create an AWS Lambda

There are few ways to create a lambda function in AWS, but the most common way to create it is with the console but this method should only be used if testing in dev. For production, it is best practice to automate the deployment of the lambda.

There are few third-party tools to set up automation, like Terraform, but since we are specifically talking about an AWS service, AWS recommends using Serverless Application Model (SAM) for this task. SAM is pretty much built on top of AWS CloudFormation and the template looks like a normal CloudFormation template except it has a transform block that specifically says we want the template to be a SAM template as opposed to a normal CloudFormation template. You can take a look at some example templates in the AWSlabs.

## AWS Lambda use cases

You can use AWS Lambda in a variety of situations, including but not limited to:

- Log transfers where a lambda function is invoked every time there is a new log event in CloudWatch to transfer the logs into tools like Elasticsearch and Kibana.
- A website where you can invoke your Lambda function over HTTP using Amazon API Gateway as the HTTP endpoint.
- Mobile applications where you can create a Lambda function to process events published by

your custom application.