AWS Lambda Lifecycle

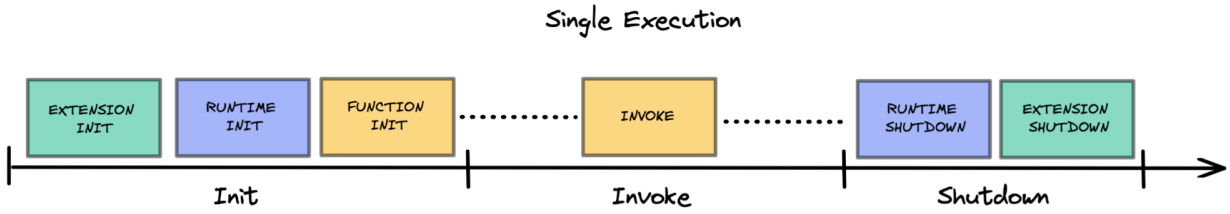
Understanding the Internal Behavior of Lambda Functions

AWS Lambda is a highly reliable serverless compute service that developers use to create cost-effective and scalable applications. Despite its popularity, many developers that are new to Lambda do not know nor understand its inner workings. Understanding the internal behavior of Lambda functions, i.e. getting to know the Lambda Lifecycle, can help developers realize cost savings while also improving the performance of their solutions.

This blog aims to give readers a better understanding of the AWS Lambda lifecycle and expands upon the internal behavior of Lambda.

The lifecycle of an AWS Lambda execution is split into 3 phases:

1. Init
2. Invoke
3. Shutdown



Single Lambda Execution

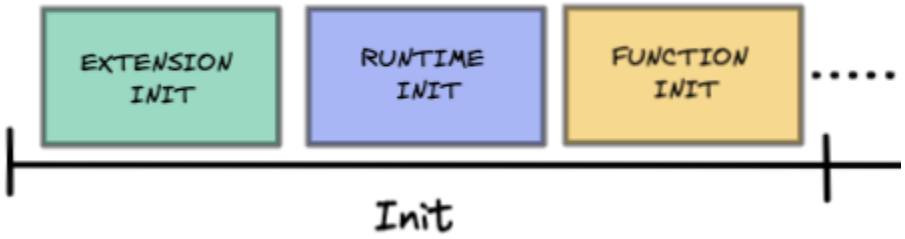
Init

The goal of this phase is to prepare the environment for the Lambda to be invoked. The Lambda function will download the code, initialize the extensions along with the runtime environment, and load code in the environment.

The Init phase is split into 3 sub-phases:

1. Extension Init

2. Runtime Init
3. Function Init



Sub-phases of initialization

Extension Init

An extension is a means to integrate AWS Lambda with monitoring, governance, and security tools without coupling it to business logic. Extensions live in a different container that is initialized before the Runtime and Function init, and they are shutdown after the Shutdown Runtime.

For example:

Datatog (Monitoring and Security)
https://www.datadoghq.com/

Datadog can be used to monitor all of the metrics supplied by Lambda, along with function logs and performance data, to provide a complete picture of your serverless applications.

To learn more about extensions, watch this video (https://youtu.be/sAgUcJOwElU) from Julian Wood (Senior Developer Advocate @ AWS). This is a complete course about Lambda Extensions, internal logic, capabilities, etc.

In the Extension Init phase the system starts all extensions to let them apply their logic to the Lambda. (visit https://docs.aws.amazon.com/lambda/latest/dg/runtimes-extensions-api.html for more details)

Runtime Init

The runtime is the combination of an operating system, programming language and a combination of software libraries required to invoke the Lambda function. You can see the list of available runtimes on AWS' Runtime Support Policy guide

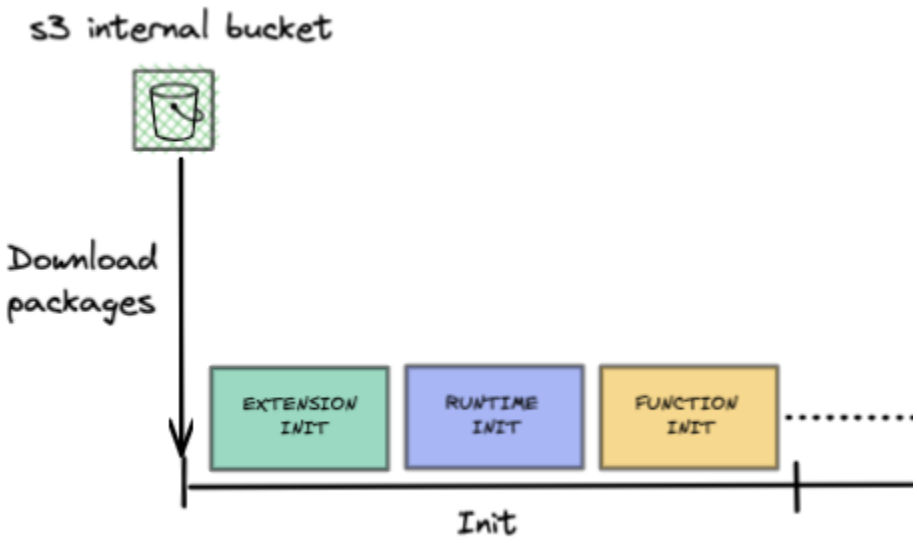
(https://docs.aws.amazon.com/lambda/latest/dg/runtime-support-policy.html). The runtime comes with the AWS-SDK already installed.

Function Init

The Function Init phase is in charge of running static code in the runtime environment. This phase allows for the code to be initialized and readied to be called by the service.
Function Init is by far the most important part of the Init phase. We will see later during this blog how to leverage Function Init logic to improve costs and performances.

Source Download of Init Phase

All resources (extensions layers, runtime package, source code) are stored in an internal S3 bucket. During the Init phase, the system pulls resources from this S3 bucket.



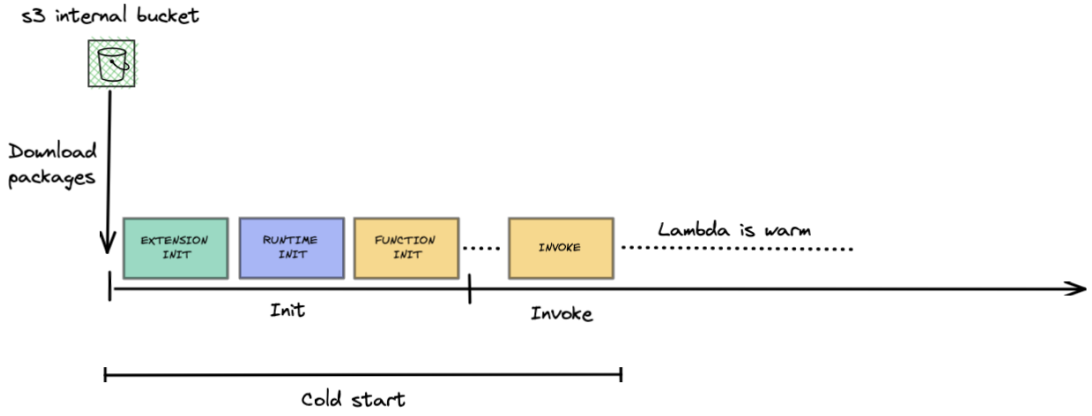
Package download

Invoke

In this section, we will refer to the concept of Warm and Cold Starts. After the Invoke phase, the Lambda environment is kept alive until the AWS Lambda service decides to kill it. When the service decides to kill it, the Shutdown Phase is triggered. A Lambda function that is waiting to be shut down is qualified as Warm.

Cold Start

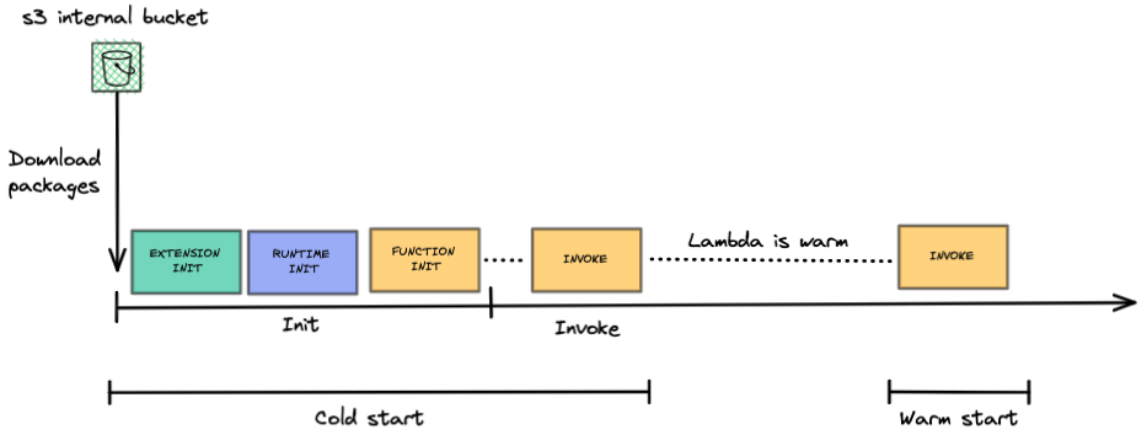
When the service request to invoke a Lambda and a new sandbox environment is spawned, it is called a Cold Start.



Cold Start of a Single Execution

Warm Start

Here's an example of Lambda executions one after another:

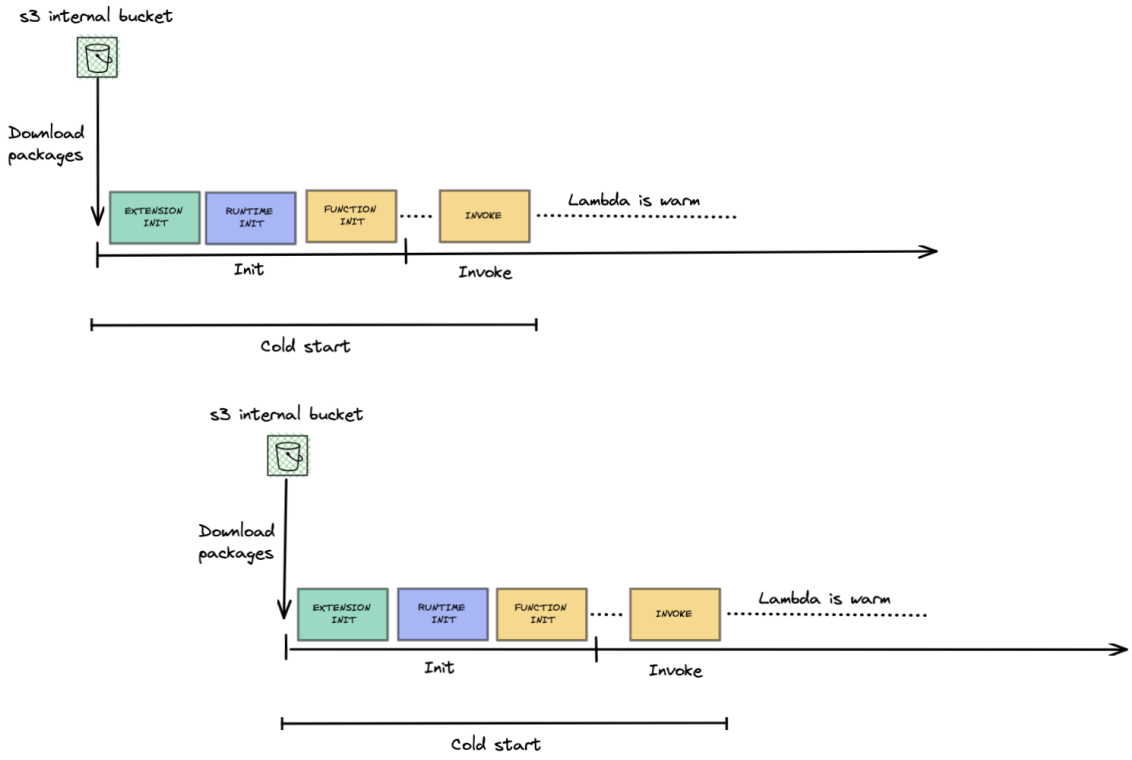


Warm Invocation

As seen in the diagram above, the second invocation does not go through the Init phase. This reduces the execution time and helps ensure that AWS only charges the user for the invocation time.

Concurrent Execution

When a Lambda function is triggered where there is no warm Lambda available, the service generates a new one, and this execution goes through a cold start.



Concurrent Executions

In this example, the service receives an invocation request while the first Lambda is in the Invoke phase i.e. it's not available to receive a new Invoke. The system then generates a brand new environment for the executions. Both these Lambdas will be kept warm after the invocations (again, until AWS decides to terminate them).

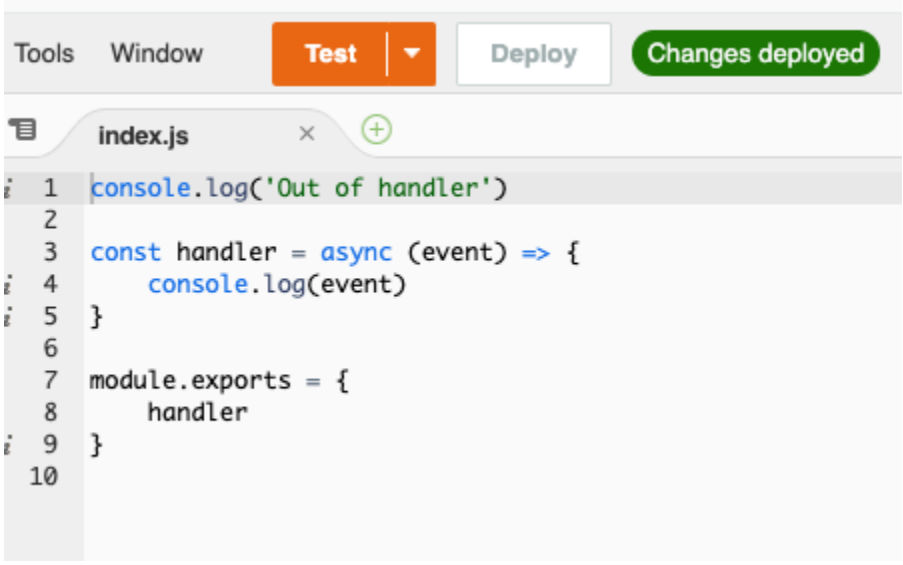
Using Amazon CloudWatch

User can observe Cold Starts on Amazon CloudWatch. They can see whether a new execution environment (container) was created for a specific Lambda function by checking the CloudWatch logs associated with that Lambda. In the associated logs, a new "Log Stream" group means that a new execution environment (Init phase) was created for the associated Lambda function.

Examples

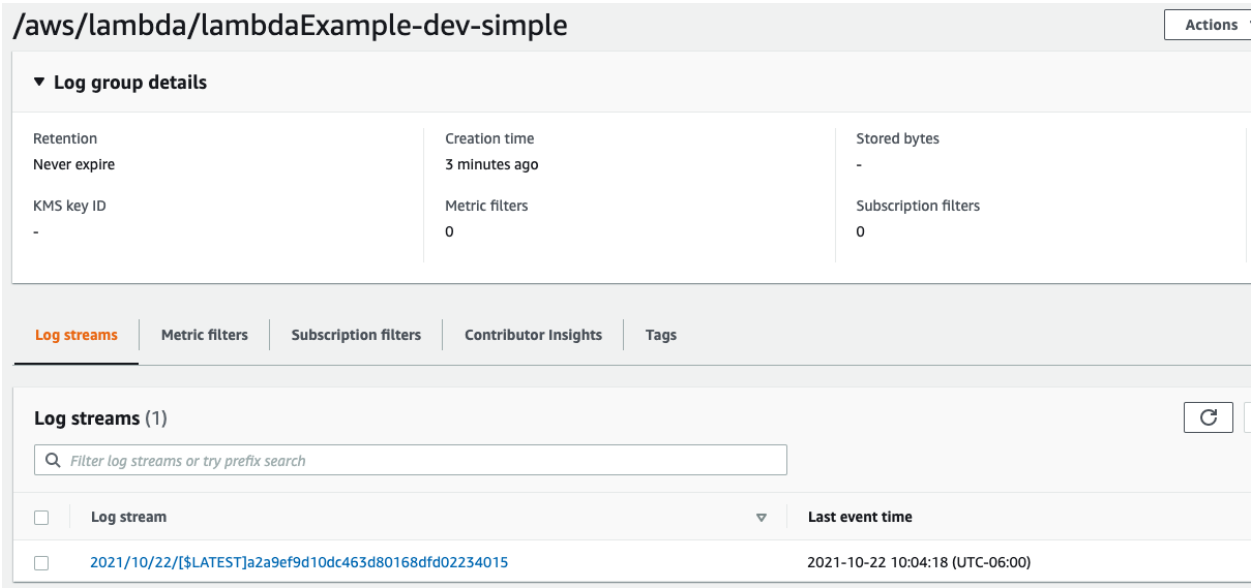
Warm Lambda

Let's consider the following example of a simple lambda function:



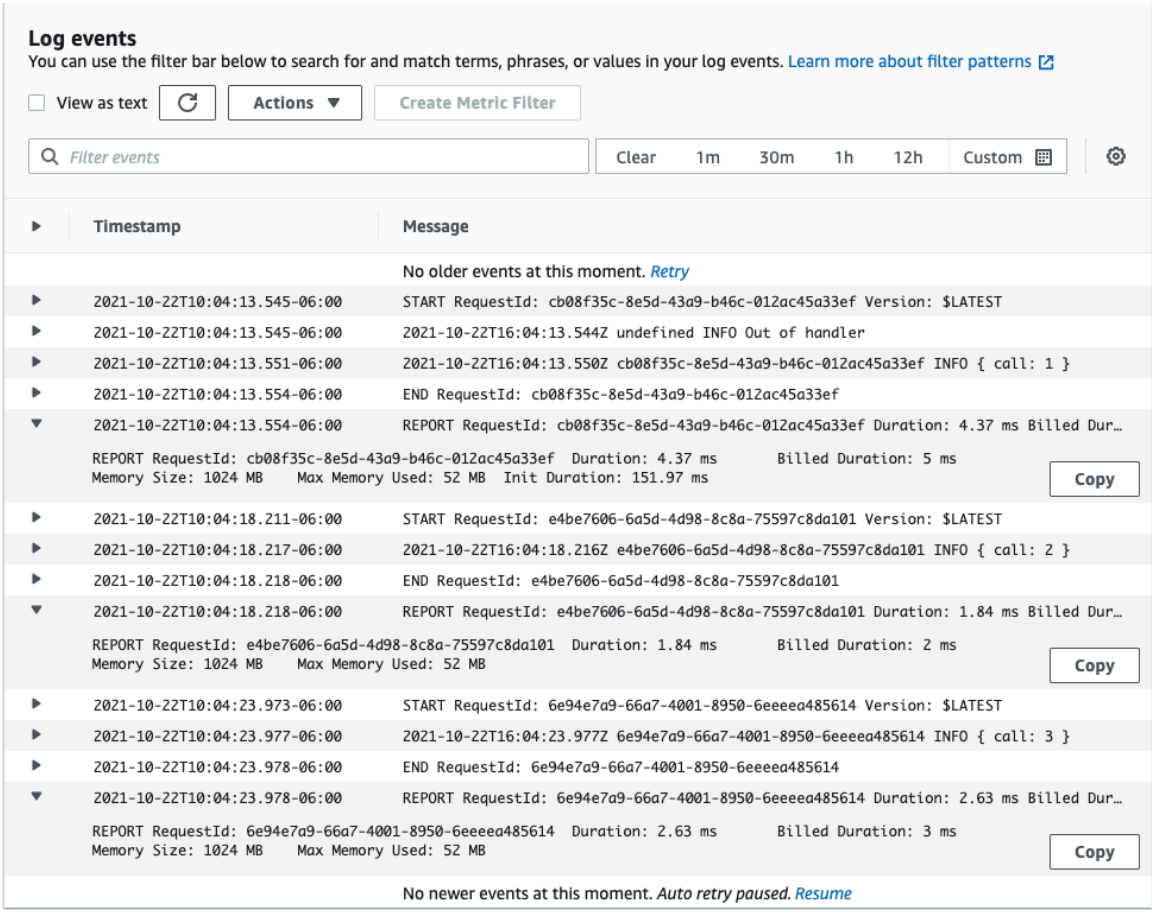
Basic Lambda Example

We have a basic handler that logs the event it receives. As seen in the code, a log has been added out of the handler to show the Init Phase. The function is then triggered 3 times, with different events, to show an execution log. As seen in the next screenshot, there is only a single Log Stream indicating that just one sandbox environment has been created for the 3 executions.



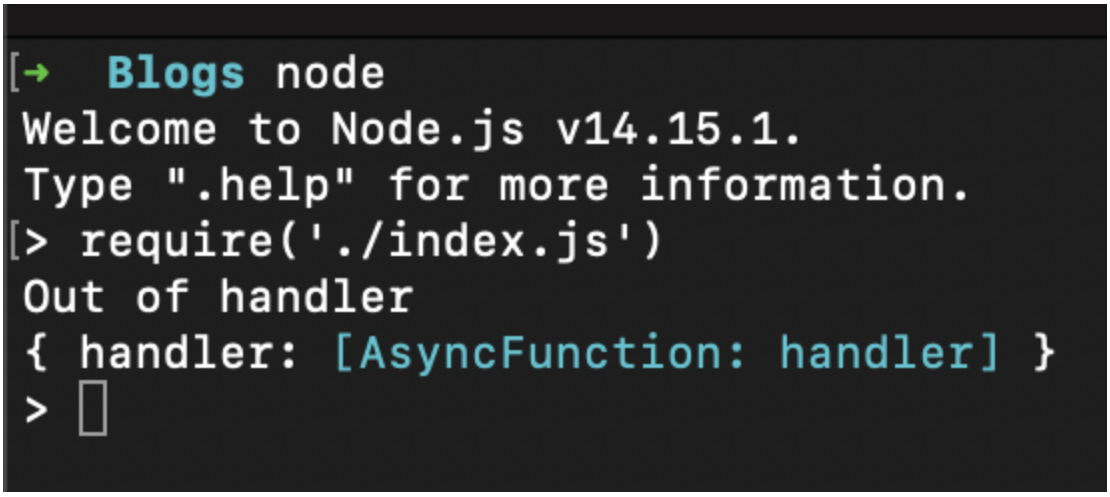
CloudWatch Console

By digging into the Log Stream, users can see their Lambda logs.
As seen in the screenshot below, there is only one “Out of handler” log, and 3 different { call: “number” } logs.



Log Events in Log Stream

This occurs because the system imports the file in nodejs. The same thing happens if the file is imported in a node process.



NodeJS Example

AWS Lambda service is doing the same thing. During the Function Init, it loads your handler file in his nodeJS environment. When the system is warm, it doesn't have to perform the Init phase.

The different durations for each Lambda invocation extracted from the 'Log Events in Log Stream' screenshot are presented in the table below:

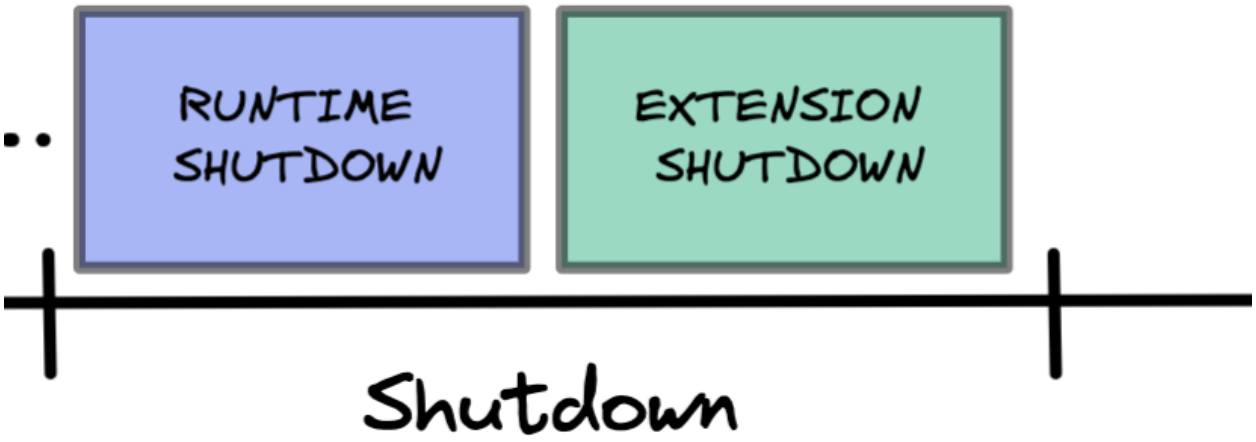
Invocation number	Init duration (ms)	Duration (ms)	Billed duration (ms)
1	151.47	4.37	5
2	0	1.84	2
3	0	2.63	3

As seen in the table above, only one of the Lambda invocations received an Init duration. This happened because it was a cold start. For the second and third call, there was no initialization phase to perform.

The duration of the first call is twice as long as the two others because **Function Init** and **Execution Init** are billed to the customer, whereas runtime init is not billed. The first invocation's total duration is 155.84 ms (151.47 ms for the initialization and 4.37 ms for the execution). Since the environment was warm for the second and third invocations, their total durations were only 1.84 ms and 2.63 ms.

Shutdown

Once the Lambda function is invoked and the Lambda Service decides to kill the environment, the Shutdown phase is triggered. The Lambda Service sends events to the Runtime and Extension processes to trigger the Shutdown Phase. If the Lambda service does not receive a response 2 seconds after the events are sent, it terminates the processes via a SIGKILL signal.



Shutdown Sub Phases

Since the Extension process listens to events from the Runtime process, the Extension Shutdown only happens after the Runtime Shutdown.

AWS Lambda, The Heart of Serverless Architecture

It is extremely useful for developers to understand the inner workings of a service like AWS Lambda that is often considered the centerpiece of business logic for solutions deployed on the cloud. This article elaborates on the internal behavior of AWS Lambda and provides readers with the insights they need to design better performing and cost-effective Serverless applications. By improving Init Phase design to reduce Cold Start costs, readers will reduce AWS Lambda-related cloud costs for their Serverless Applications.

For questions, feel free to contact the author via LinkedIn @jeremycare or email jeremy@trackit.io

About TrackIt

TrackIt is an Amazon Web Services Advanced Consulting Partner specializing in cloud management, consulting, and software development solutions based in Venice, CA.

TrackIt specializes in Modern Software Development, DevOps, Infrastructure-As-Code, Serverless, CI/CD, and Containerization with specialized expertise in Media & Entertainment workflows, High-Performance Computing environments, and data storage.

TrackIt's forté is cutting-edge software design with deep expertise in containerization, serverless architectures, and innovative pipeline development. The TrackIt team can help you architect, design, build and deploy a customized solution tailored to your exact requirements.

In addition to providing cloud management, consulting, and modern software development services, TrackIt also provides an open-source AWS cost management tool that allows users to optimize their costs and resources on AWS.