



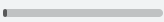


Detect, Avoid, and Troubleshoot Timeouts in AWS Lambda

blog.awsfundamentals.com/best-practices-to-avoid-and-troubleshoot-timeouts-in-aws-lambda

Tobias Schmidt



 Play this article

 0:00 / 7:51   

AWS Lambda is a popular serverless computing service that allows users to run code without the need to manage servers. However, like any other service, AWS Lambda has its own limitations and challenges.

One common that is very common is timeouts. This article will discuss best practices to avoid and troubleshoot them. We will cover what timeouts are, how to detect them, discover the reasons for timeouts, and provide best practices for minimizing, preventing, and dealing with timeouts.

Timeouts at AWS Lambda

DETECT, TROUBLESHOOT, AND MITIGATE



How Timeouts Occur ⚡

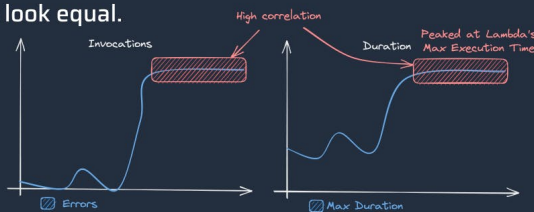
Lambda timeouts are errors that occur when a Lambda function exceeds its maximum allocated execution time and is **forcefully terminated by AWS**. This means the function will exit with an error code before it was able to finish its processing.

Reasons for Timeouts 🐛

The reason for function timeouts can be manifold. Mostly, it will be due to I/O operations (e.g. third-party calls or database queries) that didn't complete before the maximum function duration was reached.

Detecting Timeouts #2: Metrics 🔍

Another more simple way is checking CloudWatch's default metrics "Max Duration" & "Invocation Errors" to look equal.



Discovering Timeout Reasons 🕒

Use a logger middleware like middy to log context right before timeouts occur or make use of X-Ray.

Idempotency & Event-Driven Approach 📁

With idempotent functions & an event-driven approach, timeouts can simply be reprocessed automatically.

Implications of Timeouts 🔥

The immediate shutdown can result in incomplete or failed function execution, which can cause **data loss**, **service disruption**, **data corruption**, and other negative consequences.

Detecting Timeouts #1: CloudWatch Logs 🐼

Lambda reports timeouts to CloudWatch via a dedicated message: **Task timed out after <duration>**.

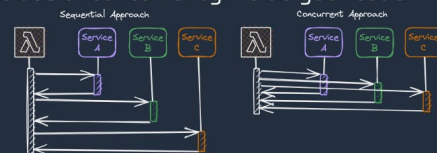
```
START [...]: $LATEST
[...] Task timed out after 3.01 seconds
END RequestId: 6f8d12ec-7d65-467c-a661-8309fd3c10b7
REPORT RequestId: 6f8d12ec-7d65-467c-a661-8309fd3c10b7
Duration: 3006.38 ms Billed Duration: 3000 ms [...]
```

We can also filter for this pattern & create our own metric that will report the count of those logs.



Minimizing, Preventing and Dealing with Timeouts 🛡️

- Decouple long-running tasks into async processes
- Minimize and optimize I/O operations
- Minimize external dependencies
- Make use of caching
- Make use of concurrency inside your code



Understanding Lambda Timeouts

Before we jump into the practical part of detecting and dealing with timeouts, we need to understand why they occur in the first place, and most importantly, what implications they come with.

How Timeouts Occur

Lambda timeouts are errors that occur when a Lambda function exceeds its maximum allocated execution time and is forcefully terminated by AWS.

This means the function will exit with an error code before it was able to finish its processing.

Lambda timeouts can't be compared to connection timeouts between two components, as this just interrupts the communication channel but not the processing. We didn't have to deal with the risk of an immediate stop of processing when working with containers, as they run all the time and do not have a maximum runtime.

Timeout Implications and Why They Should Be Avoided

The immediate shutdown of the function and process interruption can result in incomplete or failed function execution, which can cause data loss, service disruption, data corruption, and other negative consequences.

Those facts lead to the consequence to reduce, completely avoid, or mitigate their effects at all costs.

Reasons for Function Timeouts

The reason for function timeouts can be manifold. Maybe it's due to a design or architecture issue by running heavy workloads that couldn't compute in time.

Mostly, it will be due to I/O operations that didn't complete before the maximum function duration was reached. Those operations can be for example writes to a database or communications with third-party services that can have outages themselves or very slow responses due to high loads.

Detecting Timeouts

For finding the source of timeouts, we have two powerful onboard resources at AWS which are coming for free.

Analyzing Logs for "Task timed out"

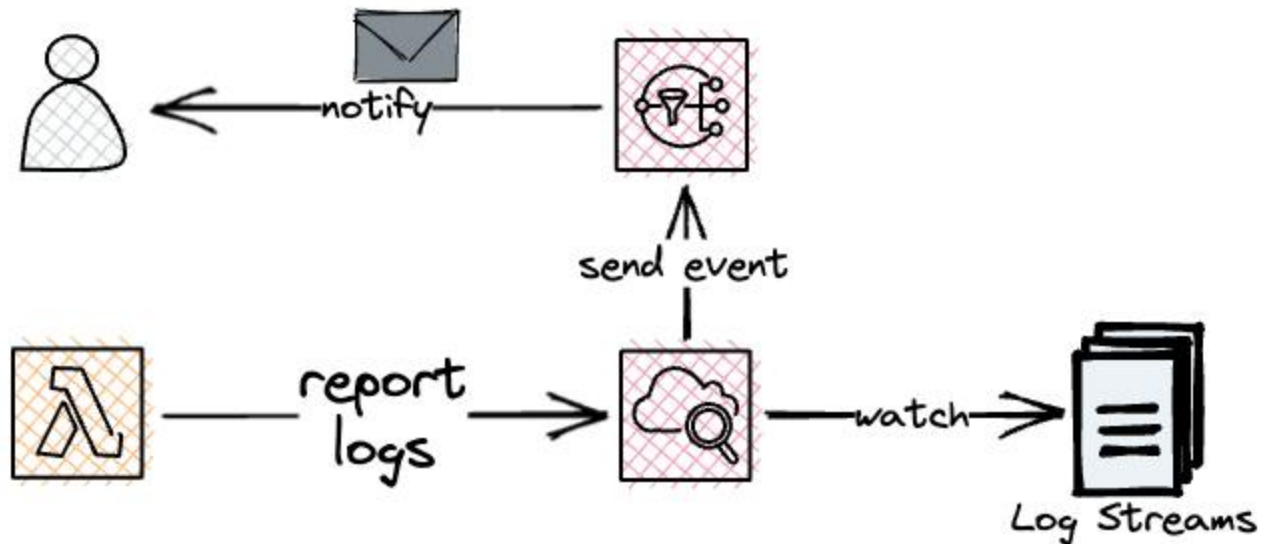
Lambda reports each of its invocations with a dedicated timeout metric that will end up in CloudWatch: `Task timed out after <duration> ms.`

```
START [...]: $LATEST
[...] Task timed out after 3.01 seconds
END RequestId: 6f0d12ec-7d65-467c-a661-8309fd3c10b7
REPORT RequestId: 6f0d12ec-7d65-467c-a661-8309fd3c10b7
Duration: 3006.38 ms Billed Duration: 3000 ms [...]
```

```
START [...]: $LATEST
[...] Task timed out after 3.01 seconds
END RequestId: 4392f94a-ae0e-e921-51d0-cb6a952bcae7
REPORT RequestId: 4392f94a-ae0e-e921-51d0-cb6a952bcae7
Duration: 3004.28 ms Billed Duration: 3000 ms [...]
```

```
START [...]: $LATEST
[...] Task timed out after 3.01 seconds
END RequestId: da5c7fb1-89ca-045b-a241-2f13136676d0
REPORT RequestId: a20350a3-216d-fd97-41f7-65ad52e0197a
Duration: 3007.11 ms Billed Duration: 3000 ms [...]
```

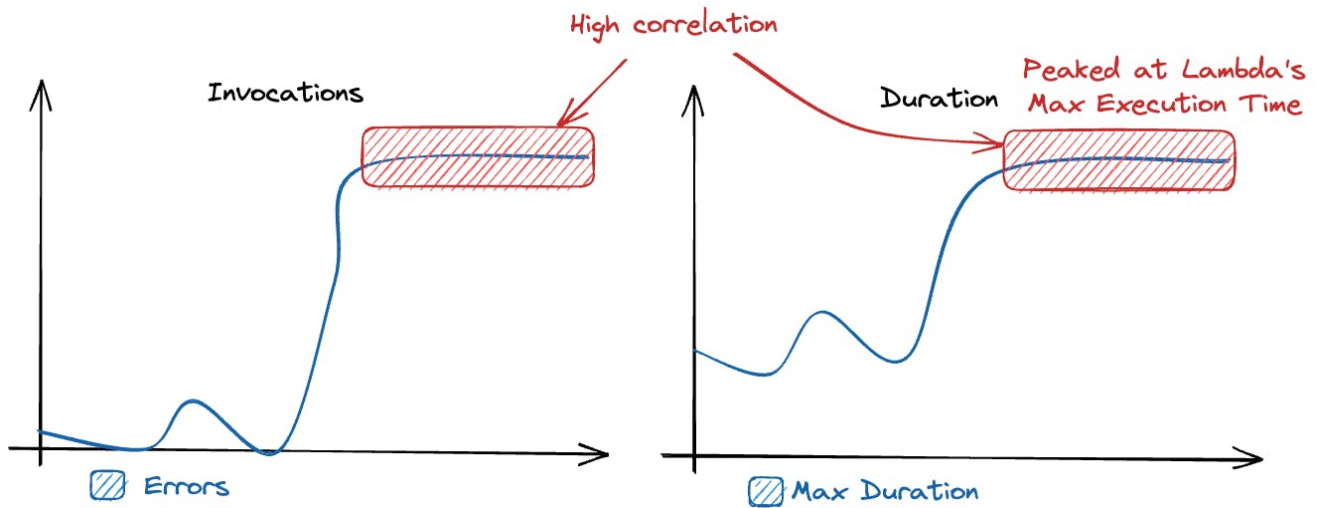
With CloudWatch, we can also filter for this pattern and create our own CloudWatch metric that will report the count of those logs.



Based on this metric in turn, we can create an alarm that will notify us via a preferred communication channel if a certain threshold is breached.

Checking CloudWatch's "Max Duration" Metric

Another more simple way is checking CloudWatch's default metrics that are collected for AWS Lambda.



The two metrics we need to look at are:

- Error Count - the number of unsuccessful invocations (not returning a success code) of Lambda functions.
- Maximum Duration - the maximum amount of time, in milliseconds, that a function invocation took to complete before it returned a response or time out.

If both lines correlate to a high degree and the maximum duration flatlines at some point, we can be sure that the errors are related to the function regularly reaching its maximum duration.

Discovering the Reasons for Timeouts

We need to identify the reasons for timeouts before we can take action to mitigate or avoid them.

Using a Logger Middleware

The issue with timeouts: there's no native option by AWS to simply run additional code right before the function times out. So we can't add context to the logs to know why the timeout occurred in the first place. We have no way of seeing the stack trace.

A middleware like middy can be used to do exactly that. We can add a custom routine that is executed right before the timeout will trigger so we can provide the needed context of our current process.


```
import middy from '@middy/core'

const lambdaHandler = (event, context, {signal}) => {
  signal.onabort = () => {
    // cancel events
  }
  // ...
}

export const handler = middy(lambdaHandler, {
  timeoutEarlyInMillis: 50,
  timeoutEarlyResponse: () => {
    return {
      statusCode: 408
    }
  }
})
```

This is a convenient way of finding the sources of our timeouts, as we can find out which operations happened before and at which step we are in our process.

A logger middleware will also help to add context and logs to all the important events like third-party HTTP calls so we immediately see time-consuming I/O operations in our logs.

Using X-Ray

We can also make use of X-Ray by using the corresponding SDK. With this, we can wrap HTTP calls and calls to other AWS services so X-Ray will take care of providing us with the execution context and the distributed tracing across services.

We'll see how long I/O operations take, even in an intuitive console interface, so we immediately know why a certain process has failed.

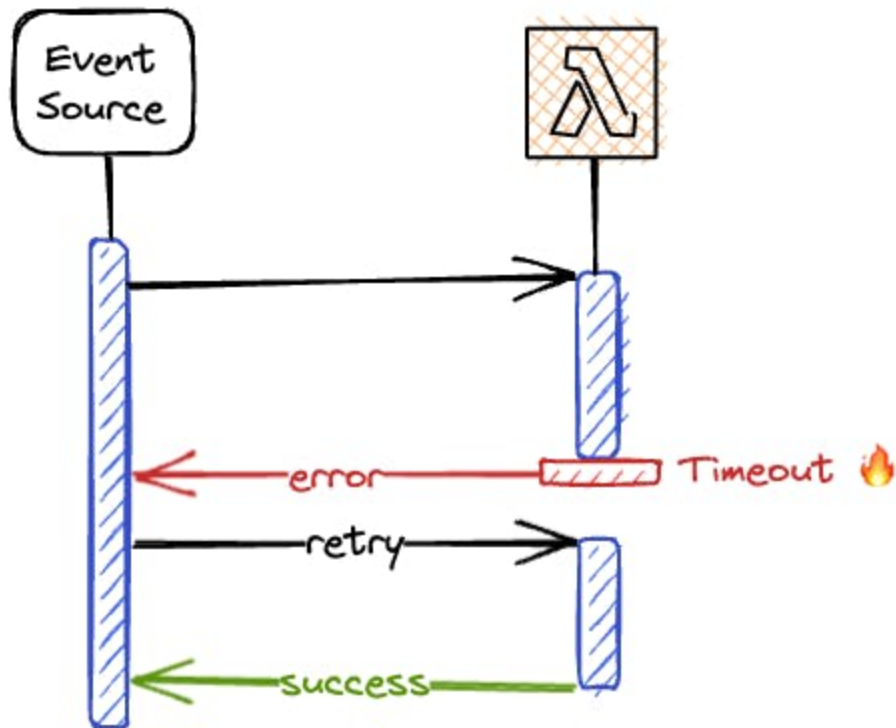
Best Practices for Minimizing, Preventing and Dealing with Timeouts

There are multiple ways of either reducing the number of timeouts or mitigating their negative effects.

Ensuring Function Idempotency

This is maybe the most important consideration: making our functions idempotent.

In AWS Lambda, idempotency refers to the property of a function where invoking the same function multiple times with the **same input parameters** produces the **same result** as invoking it once.



This means we can safely retry in the cause of errors without unintended side effects, such as duplicate records, errors, or incomplete processing.

Using an Event-Driven Approach

The combination of an event-driven approach and idempotent functions will allow for building highly-reliable architectures, as timeouts can be automatically handled safely without side effects.

With Lambda destinations, we can forward failed events to an SQS error queue and automatically retry those events later on. Generally, we can queue events and Lambda can take on processing automatically. Only successfully processed events will then be removed from the queue.

Adjusting the Function Configuration

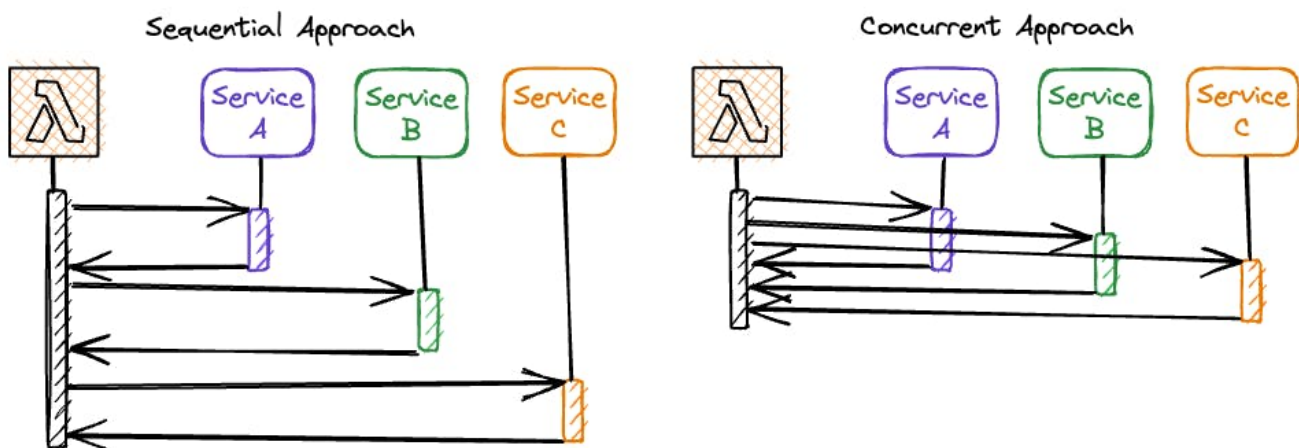
The most obvious first step for long-running operations that are regularly timing out: we can adjust our timeout settings to Lambda has more time to finish the process.

This timeout can be increased up to 15 minutes. Be aware that customer-facing functions shouldn't have long execution durations as each 100ms that are added until the response is returned will negatively impact satisfaction. Long maximum duration should only be considered for background, asynchronous operations.

Optimizing our Function Code

There are several ways to optimize function code to reduce timeouts in AWS Lambda:

1. **Decouple long-running tasks into async processes** - only process immediately necessary operations that are required to return a result. Send long-running tasks into a queue so another function can safely take over those operations.
2. **Minimize and optimize I/O operations** - Minimize the number of input/output operations, such as database queries or file systems access, that the function performs to reduce latency. Maybe there are long-running database queries that can be improved. Make sure to never use inefficient operations like scan for DynamoDB, but proper queries.
3. **Minimize external dependencies** - Avoid using external dependencies or APIs that may introduce additional latency or failure points.
4. **Use caching**: Use caching to reduce the need for repetitive computations and improve performance. Be aware that caching in multi-tenant, distributed systems can cause other side effects.
5. **Make use of concurrency inside your code** - if you're doing multiple third-party calls, try to submit them concurrently instead of waiting for responses sequentially. This can drastically reduce execution times.



By implementing these optimization techniques, developers can improve the performance and reliability of their Lambda functions, reduce the risk of timeouts, and optimize resource usage in their applications.

Conclusion

In conclusion, timeouts can cause serious negative consequences such as data loss, service disruption, and data corruption. It is important to detect and troubleshoot timeouts in AWS Lambda to minimize their effects.

Developers can use various techniques such as ensuring function idempotency, using an event-driven approach, adjusting function configuration, and optimizing function code to reduce timeouts and improve the performance and reliability of their Lambda functions. By following these best practices, users can optimize resource usage in their applications and ensure a seamless experience for their customers.

Published on
