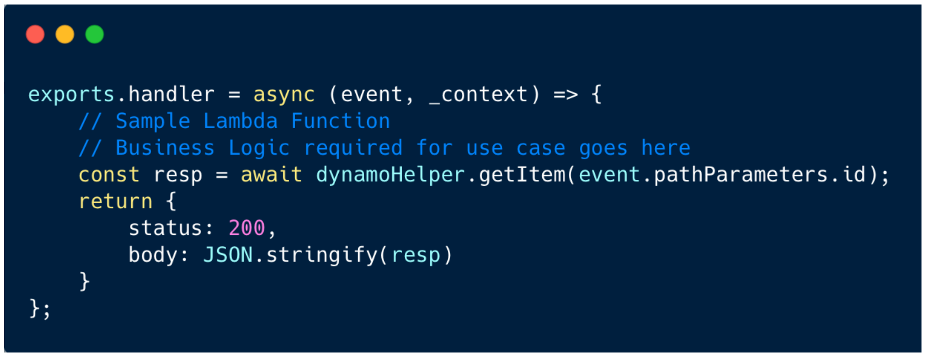
Introduction to Lambda Functions

AWS Lambda is an event-driven compute service introduced in 2014. It provisions compute resources on-demand to execute code in response to events. Since AWS manages the end-to-end provisioning of resources for Lambda, no underlying server infrastructure is visible to developers. Therefore, it is widely known as a fully managed Serverless compute solution.

A Lambda function can be written in a variety of programming languages. The following example shows a Lambda function written using NodeJS looks like.



```
exports.handler = async (event, _context) => {
    // Sample Lambda Function
    // Business Logic required for use case goes here
    const resp = await dynamoHelper.getItem(event.pathParameters.id);
    return {
        status: 200,
        body: JSON.stringify(resp)
    }
};
```

T-Mobile adopted a serverless first policy to develop its mission-critical platforms. They utilized the event-driven nature of AWS Lambda to create triggers for their database, S3 events, enabling them to scale up with demand effectively. Furthermore, their development times increased by 90% thus, ensuring faster release times as they do not have to maintain infrastructure.

Advantages using Lambda Functions

As you can see, there are several advantages of using Lambda that play a significant role in its popularity. Let's look at some of them in detail.

Resource Provisioning

AWS provisions the resources for Lambda on-demand, allowing it to scale well for varying workloads. In addition, since AWS takes care of the infrastructure to execute the function, developers only need to focus on writing the code that leads to faster development times, increasing productivity.

Costs

Secondly, AWS bills Lambda functions for every millisecond it is executed and the number of requests served by AWS Lambda.

Cross-Service Integrations

Additionally, Lambda functions seamlessly integrate with other AWS Services. For instance:

1. You can use the AWS API Gateway connected to Lambda to develop an end-to-end Serverless API. These Lambdas' can communicate with databases to fetch data for client requests.

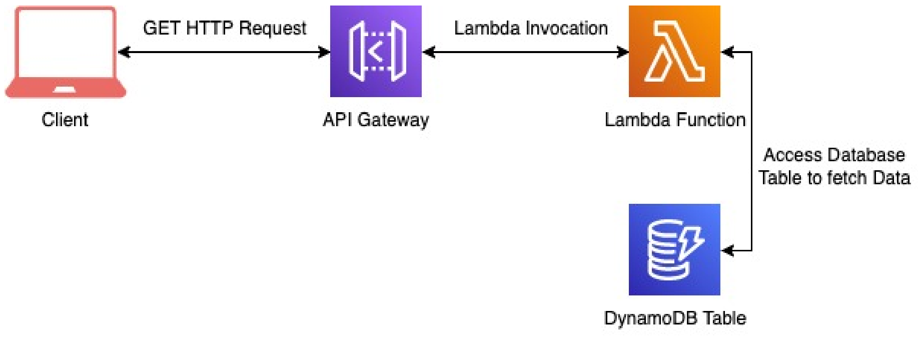


Figure 01 - A Lambda function integrated with the API Gateway.

2. You can set up triggers in your database tables to perform actions when data is persisted or removed.

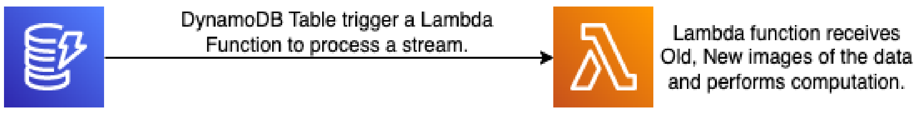

DynamoDB Table trigger a Lambda Function to process a stream.
Lambda function receives Old, New images of the data and performs computation.

Figure 02 - A Lambda function invoked by a DynamoDB table.

3. You can process data passed from SNS topics or Queues by setting up triggers to provide quick responses to requests.

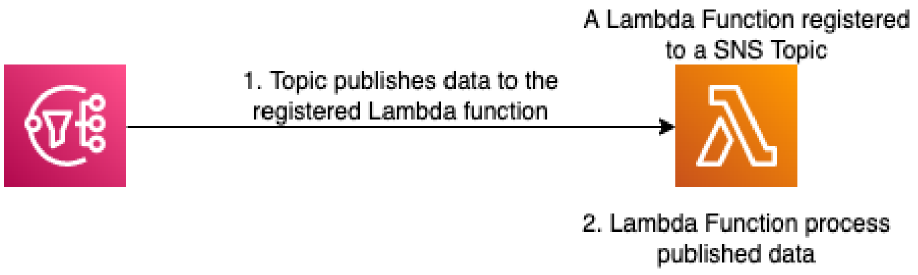

A Lambda Function registered to a SNS Topic
1. Topic publishes data to the registered Lambda function
2. Lambda Function process published data

Figure 03 - A SNS Topic publishing data to a Lambda function

In addition, there are many other integrations, including AWS Lambda S3 trigger, DynamoDB trigger, AppSync trigger.

However, along with all these benefits, moving to a Serverless stack using Lambda also brings several challenges. Therefore, it is crucial to create awareness to reduce their impact or avoid them altogether. So, let's look at these challenges in detail.

Addressing Latency in Lambda Functions

The main challenge of adopting Lambda is the latency it brings in typically with cold starts. Besides, there are other reasons as well.

Cold Starts

A Lambda function is initialized in an execution environment only when its invoked. Therefore it experiences delays during its first invocation. This is considered a cold start.

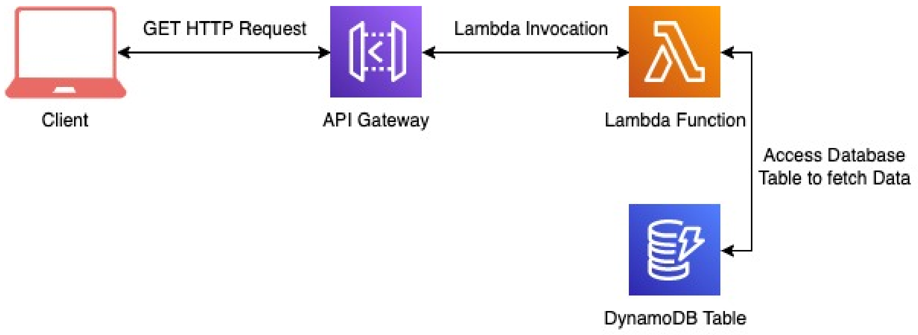


Figure 04 - A Lambda function used in an API Gateway

Figure 04 shows a client making an HTTP request to fetch items from a DynamoDB table via a Lambda function. When this request gets executed for the first time, AWS Lambda will:

1. Create a new execution container environment.
2. Download the Lambda code.
3. Initialize the function module.
4. Execute the function code passing the event.

Steps 1 to 3 happen only in the first request made to the Lambda container and can take more than 10 seconds to execute as the Lambda service must search for warm containers, create new ones when none are available, and finally, download/initialize the Lambda code before running it.

Additionally, this initialization time differs based on the initialization code size and runtime environment (e.g Node.js, C#, Python).

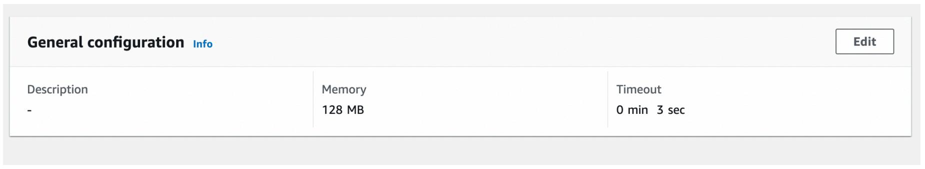
An important note is that a single worker can process only one event at a time. Although a single worker gets warm after its first invocation (for a period of approximately the next 5 minutes), the cold start could still occur for concurrent invocations. It happens since new workers get created for every concurrent event.

As you can see, the main challenge with cold start is the adverse effect on application performance. For example, Fidel, an API for linking Bank Cards to applications, was heavily affected by cold starts when sending password reset emails. This caused delays for users trying to gain access to their accounts, potentially harming consumer trust in this application.

Therefore, cold starts must be addressed and fixed after analyzing your application access patterns.

Memory Allocation to Lambda Functions

Another cause for Lambda latency happens with its memory configuration. By default, AWS assigns 128MB of memory to your Lambda function. However, since many developers go with the default configuration, its impact isn't fully realized without further analyzing its impact on Lambda performance.



General configuration Info

Edit

Description	Memory	Timeout
-	128 MB	0 min 3 sec

Figure 05 - Default Memory Allocation to a Lambda Function

Besides, in Lambda, there is a direct proportionality between memory allocation and CPU allocation. Lower the memory, lower the CPU allocated to the Lambda function. Because of that, there is a clear impact on the code execution time depending on the nature of the code and the Lambda memory configuration.

For example, if we allocate 128MB of memory for a code that demands high CPU, network, or memory, it will:

- Impact the execution time and cold start.
- Incurring more cost since Lambda cost is tied with the execution time.
- Could even occasionally timeout resulting in costly retry operations.

Ultimately all these cause a delayed response time. This is why AWS recommends allocating 128MB of memory for Lambda functions that don't demand performance such as those route events to other services. In contrast, Lambda functions that communicate with Databases and Amazon S3 must undergo memory optimization.

Lambda Runtime Language

Some of you may wonder how the programming language in which we write the Lambda function affects its performance. Although AWS provides the flexibility of writing Lambda functions in various languages (Node.js, Java, Python, Go, Ruby, C#), some perform better than others. Besides, specific programming languages create latency during initialization and contribute to long cold starts, causing high latency.

How does a programming language create longer cold starts?

Java, C# requires AWS Lambda to bootstrap both the virtual environments and runtimes to execute the code. Additionally, it requires all compiled resources (including unused) to be imported to the function. This contributes to the cold start duration taking more time for the Lambda function to complete its work.

Therefore, developers that use Java, C#, or any VM-dependent, statically typed language to write Lambda functions unknowingly add overhead to their cold start durations.

Service Integrations with Lambda Functions

So far we have discussed the latencies that occur during the initialization of a Lambda function. Besides, Lambda functions can experience high latency during execution time as well. It occurs due to the delays caused by the end services integrating with the Lambda function.

For example, a Lambda function can integrate with:

1. DynamoDB to fetch or persist data.
2. Amazon S3 to retrieve or store an image asset.
3. Amazon EC2 to start or stop a Virtual Machine.

Let us look at a Lambda function that integrates with the API Gateway and DynamoDB to understand how the end services add delay.

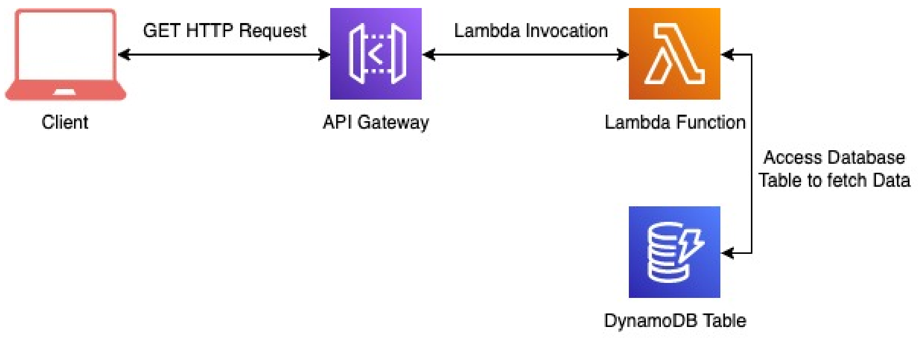
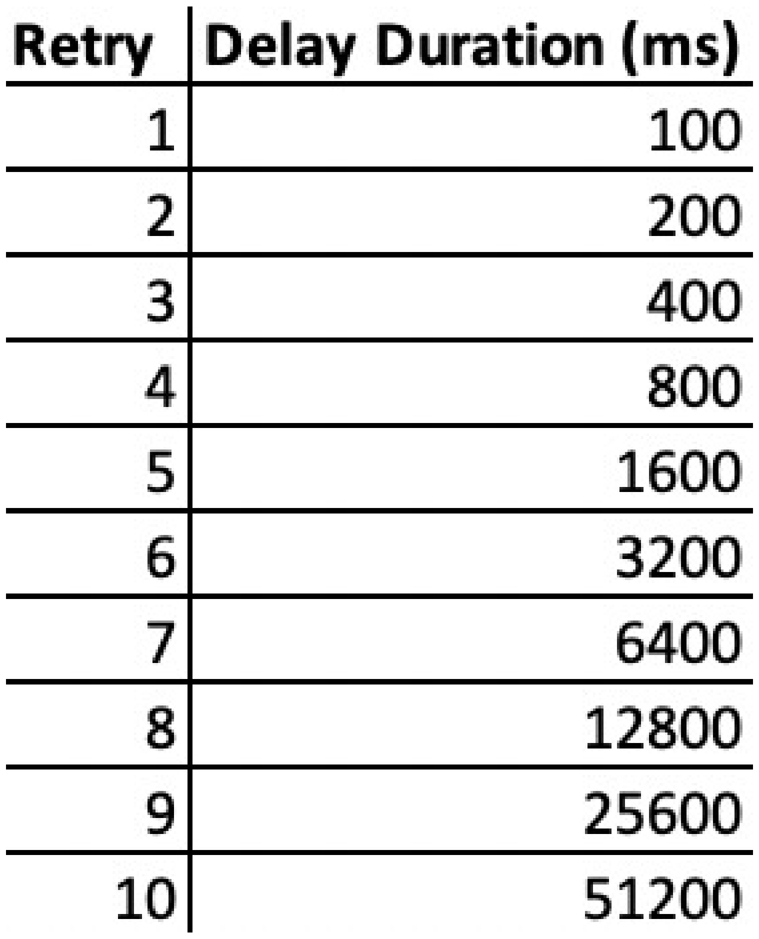


Figure 06 - Lambda integration with DynamoDB

The event sequence illustrated above shows an API Request invoking a Lambda function to fetch data from DynamoDB and return the data to the client.

The end services (such as DynamoDB) contribute to the overall execution time of the Lambda function. Here, the impact could vary from a few milliseconds to multiple depending on the query or scan operation we perform. Besides, there could be unexpected delays, if DynamoDB runs into errors. It happens since AWS SDK retries the DynamoDB request for a default period of 10 times using the exponential backoff algorithm with an initial delay at 50ms.



Retry	Delay Duration (ms)
1	100
2	200
3	400
4	800
5	1600
6	3200
7	6400
8	12800
9	25600
10	51200

Figure 07 - Exponential Delay Period

However, services like API Gateway usually add a fixed latency to the Lambda function which usually goes for several milliseconds. By default, a Lambda function is set to time out at 3 seconds and can have a maximum of 29 seconds when integrated with the API Gateway.

Therefore, it can cause the Lambda function to unnecessarily hang and create response delays of up to 4 seconds (in a warm invocation) or more than 15 seconds in cases of a cold invocation. These delays are noticeable if a user is waiting for the result and affect the overall user experience. For example, a recent page load benchmark by Google indicated that sites having page load times of up to 10 seconds have a bounce rate over 123%.

Accelerating Lambda Functions

Modern applications need to perform well. The ones that perform poorly have the risk of losing sales in drastic numbers.

Research conducted by Akamai Technologies in 2017 indicated that 100ms latency could cause a 7% loss in revenue. This rate was at 1% in 2009 when initially examined by Amazon. It shows that user experience is becoming critical in the coming years, and 100ms latency is not an option.

Therefore, it is crucial to fix latency issues in Lambda functions and improve the overall application performance.

As we have observed before, the cold start is one of the major contributors to poor Lambda functions. The good news is that there are five main ways to mitigate cold-start delays.

1. Implementing function Warmers
2. Implementing Provisioned Concurrency
3. Optimizing the Worker Runtime Language and Memory Allocation
4. Using AWS Lambda Power Tuning

Implementing Function Warmers

The first approach we are going to discuss is using function warmers to reduce cold start time. Function warmers ping a set of Lambda functions over a given period using EventBridge Rules to keep the Lambda functions warm and their workers active. This approach increases the probability of using a warm container when we invoke the function using API Gateway or any other service, thus, improving overall performance.

Let's look at a step-by-step example of setting up a function warmer using Serverless Framework. To do that all you need is the plugin - Serverless WarmUp Plugin. It can be added to your project using the NPM command shown below.

```
npm i --save-dev serverless-plugin-warmup
```

The WarmUp plugin must be configured after adding it to your project by declaring it in the plugins array in the `plugins` array in the `serverless.yml` file, as shown below.

```
plugins:
  - serverless-plugin-warmup
```

After this, the warmer function must get permission to invoke selected Lambda functions. This can be done by providing an IAM Role Statement, as shown below.

```
provider:
name: aws
runtime: nodejs12.x
lambdaHashingVersion: 20201221
```

```
iamRoleStatements:
  - Effect: 'Allow'
    Action:- 'lambda:invokeFunction'
    Resource: "*"
```

Next you can configure the Lambda function to use the function warmer. This can be done by adding the property `warmer: true` in the Lambda function declaration in your `serverless.yml`.

```
functions:fetchAccountInformation:warmup: truehandler: handler.fetchAccountInformation
```

After this, the Lambda function must be provided with a condition to break execution if triggered by the function warmer.

```
"use strict";

module.exports.fetchAccountInformation = async (event, context, callback) => {
   if (event.source === "serverless-plugin-warmup") {
      console.log("Triggered by Function Warmer To Keep Environment   Warm");
      callback(null, "Warming Lambda Up");
   }
   return {
     statusCode: 200,
     body: JSON.stringify({
       message: "Go Serverless v2.0! Your function executed successfully!",
     }),
   };
};
```

After the function gets deployed, the function warmer will be triggered every 5 to 6 minutes, allowing the selected Lambdas to get invoked to keep them warm. It ultimately increases performance by reducing latency.

However, AWS does not recommend function warmers in production workloads as the function can experience cold starts when Lambda functions scale up with traffic increase.

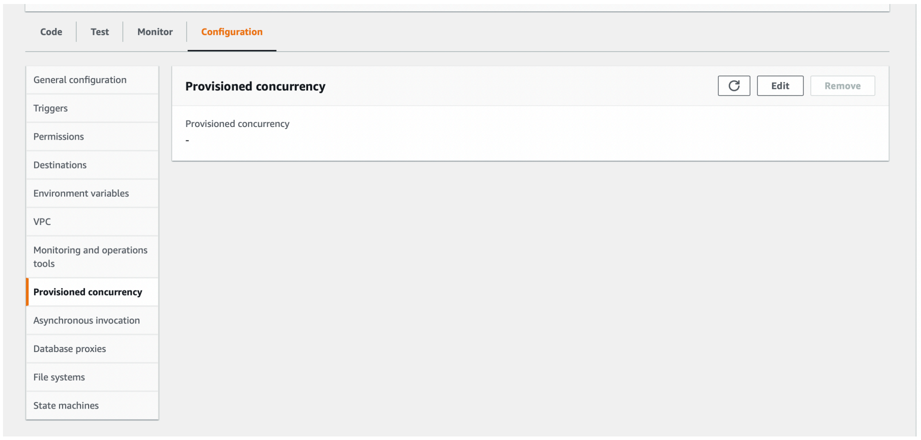
Implementing Provisioned Concurrency

AWS introduced Provisioned Concurrency to obtain consistent warm invocations with low latency in production workloads. It provides a way of preparing workers before receiving traffic, hence, provisioned. Furthermore, it downloads the function code, runs the initialization code, and keeps the worker on standby in a warmed state, ensuring that the API Gateway can respond quickly with low latency.

To enable Provisioned Concurrency:

1. Go to your Lambda function in the AWS Console.
2. Create a new version by publishing the function.
3. Navigate to Configuration.

4. Click “Edit” On Provisioned Concurrency.

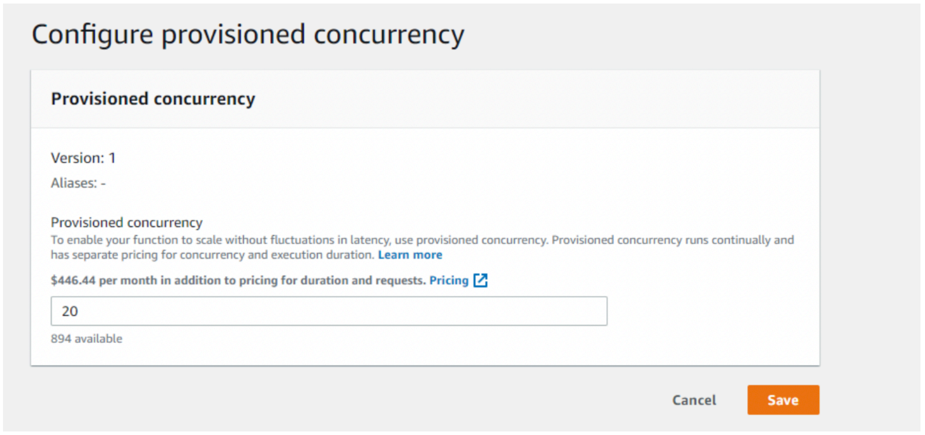


Provisioned concurrency

Provisioned concurrency
-

Figure 08 - Viewing Provisioned Concurrency Setup Panel

5. Select the required workers (concurrency) to keep warm.



Configure provisioned concurrency

Provisioned concurrency

Version: 1

Aliases: -

Provisioned concurrency

To enable your function to scale without fluctuations in latency, use provisioned concurrency. Provisioned concurrency runs continually and has separate pricing for concurrency and execution duration. Learn more

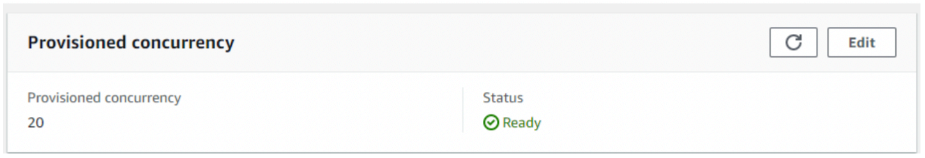
$446.44 per month in addition to pricing for duration and requests. Pricing

20

894 available

Figure 09 - Setting Required Workers for Lambda

6. Click ***Save,*** and the effect will take place in one minute.



Provisioned concurrency

Provisioned concurrency: 20

Status: Ready

Figure 10 - Configured Provisioned Concurrency

Note: As we added Provisioned Concurrency to a new version, the invoker must call the newly published version of the function to use a warmed-up instance. Provisioned Concurrency cannot be applied to the $LATEST version.

During execution, if the active concurrent workers exceed 20 (set value), the Lambda function accepts the 21st and subsequent concurrent invocations as an on-demand basis where workers get allocated with cold starts.

More importantly, there is a significant difference in Lambda performance after implementing Provisioned Concurrency, as shown below.

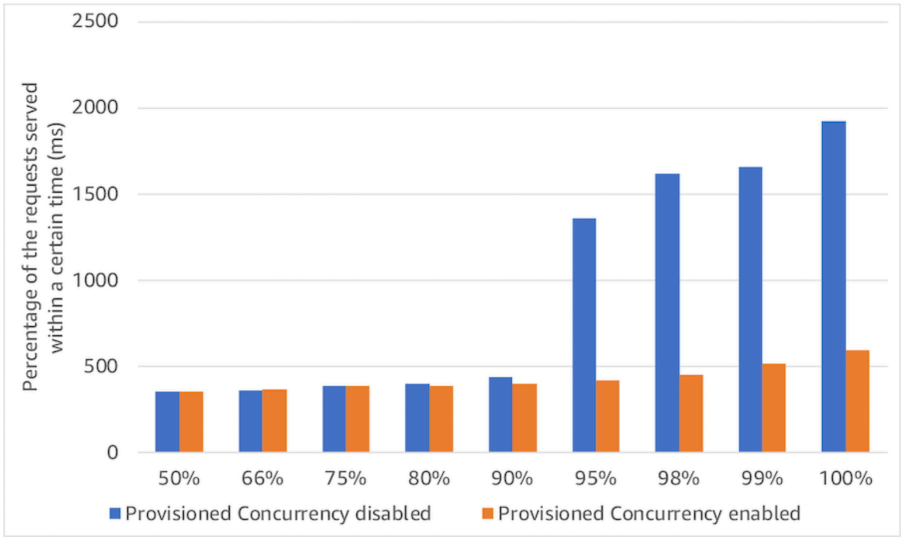


Figure 11 - Performance Comparison of Provisioned Concurrency

Figure 11 shows the results of Two Lambda functions (with/without Provisioned Concurrency) and how it handles 10,000 requests through the API Gateway. The Lambda without Provisioned Concurrency creates high latency as workers need to initialize due to growing requests. The Lambda with Provisioned Concurrency provides responses under 500ms.

Therefore, AWS recommends using Provisioned Concurrency over function warmers to obtain low latency at a constant rate.

Optimizing the Worker Runtime Language and Memory Allocation

The memory allocated to the worker and the runtime language of the worker can be optimized to bring low latency.

Firstly, runtime languages such as Java and C# create high cold start times as VMs must get spun up. However, it can be solved with Provisioned Concurrency. But, this increases the cost for the Lambda function. Therefore, it is essential to write Lambda functions in an optimal language.

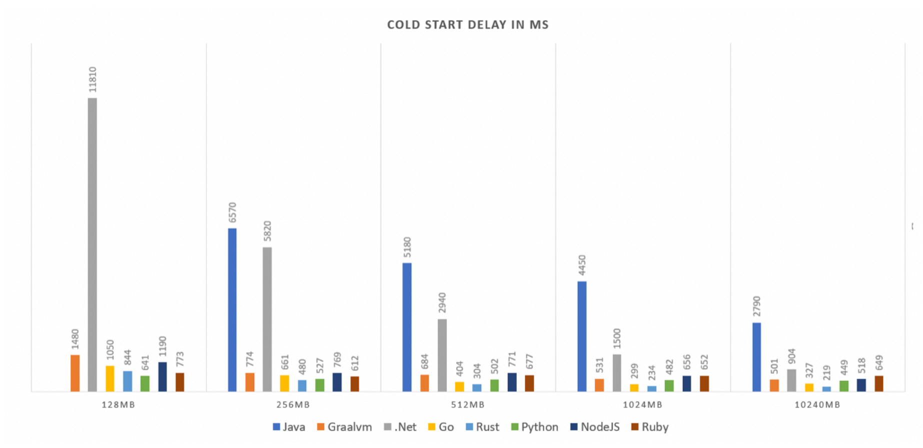


Figure 12 - Lambda cold start delay for each language (Provided by Aleksandr Filichkin)

Figure 11 illustrates cold start delays for Lambda functions written in all supported Languages across different memory types. For example, it shows that with 10GB of memory, there is a 3-second cold delay in Java and a 1-second delay with .NET. Therefore, some of your Lambda functions will not even start with 128MB of memory on .NET and Java and may timeout.

However, Rust, Node.JS, Python perform well and have minimal cold start delays ($<$ 1 second). Therefore, write your Lambda functions in these three languages to achieve less latency.

Secondly, the memory allocated for a Lambda function must be optimized. This is solely based on your use case. If your computation is CPU intensive, more memory will help increase your performance.

Therefore, a recommendation accepted by the Serverless community is to perform a trial and error method to optimize the memory allocation effectively.

The best way to accomplish this is by using the AWS Lambda Power Tuning tool. It accepts a Lambda function ARN. It invokes the function on various memory configurations from 128MB to 10GB, analyses the output logs, and suggests optimal configurations to achieve less latency at a lower cost.

End Service Optimization

Fixing cold starts can still create high latency in the execution time. It is caused due to end services that integrate with the Lambda function. Therefore, they must get optimized to provide faster execution times, thus, reducing latency.

When a Lambda function integrates with an end service, ensure to configure the following.

Timeouts on End Services

Amazon Cognito has a default timeout of 5 seconds with three retries for Lambdas. This can cause significant performance issues.

For example: when executing DynamoDB operations, it may exceed 5 seconds, and as database queries are asynchronous, the retry will not stop the previous database query. Therefore, the database query gets invoked more than once.

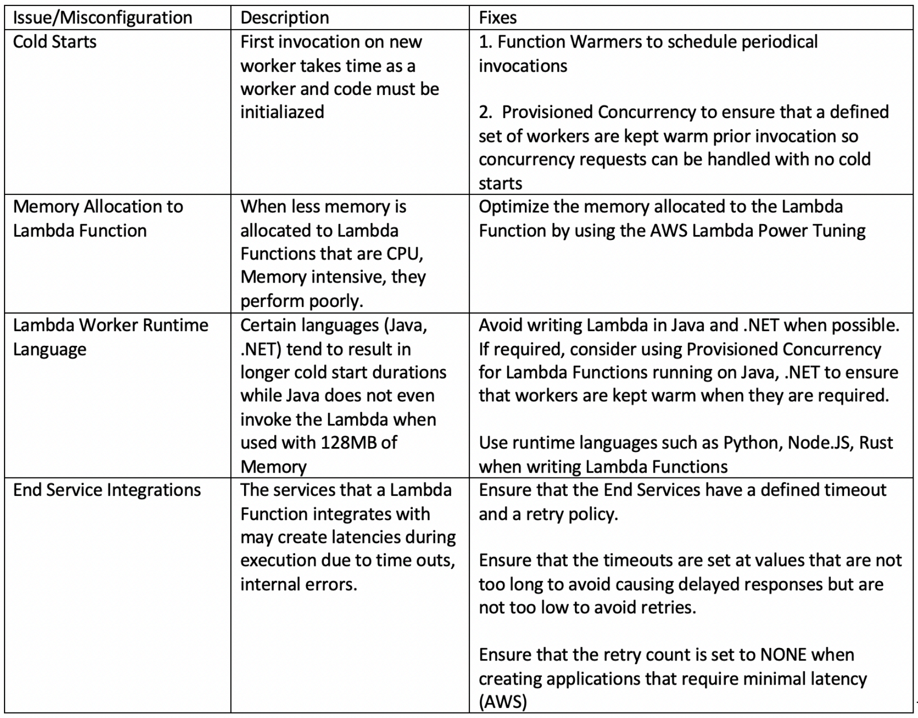
You may face this issue when using Cognito Lambda triggers. For instance, if you have multiple DynamoDB queries to store the user information in the database inside the Cognito user confirmation Lambda trigger, it's likely to exceed the 5-second mark and fall back to retries. To fix it, you have to increase the default timeout of the Lambda function associated with Cognito

Retry limit

Configuring the retry limit is helpful when we use DynamoDB with AWS SDK. DynamoDB has a retry rate of 10 times with exponential backoff by default. Therefore, a Lambda could get delayed up to a noticeable latency of 5 seconds because of the exponential backoff. Thus, for applications that require minimal execution latency, use no retries for Lambda functions.

Summary

A Lambda function takes away the complexity of maintaining servers allowing us to focus on writing the code for the function. However, a Lambda function has room to generate high latencies and delayed responses due to misconfigurations. These misconfigurations and fixes are summarized below.



Issue/Misconfiguration	Description	Fixes
Cold Starts	First invocation on new worker takes time as a worker and code must be initialiazed	1. Function Warmers to schedule periodical invocations 2. Provisioned Concurrency to ensure that a defined set of workers are kept warm prior invocation so concurrency requests can be handled with no cold starts
Memory Allocation to Lambda Function	When less memory is allocated to Lambda Functions that are CPU, Memory intensive, they perform poorly.	Optimize the memory allocated to the Lambda Function by using the AWS Lambda Power Tuning
Lambda Worker Runtime Language	Certain languages (Java, .NET) tend to result in longer cold start durations while Java does not even invoke the Lambda when used with 128MB of Memory	Avoid writing Lambda in Java and .NET when possible. If required, consider using Provisioned Concurrency for Lambda Functions running on Java, .NET to ensure that workers are kept warm when they are required. Use runtime languages such as Python, Node.JS, Rust when writing Lambda Functions
End Service Integrations	The services that a Lambda Function integrates with may create latencies during execution due to time outs, internal errors.	Ensure that the End Services have a defined timeout and a retry policy. Ensure that the timeouts are set at values that are not too long to avoid causing delayed responses but are not too low to avoid retries. Ensure that the retry count is set to NONE when creating applications that require minimal latency (AWS)

Figure 13 - Recap of the content covered in this e-book

These tips will help you decide on the best solutions to implement, ensuring faster Lambda functions with low latency in the future.