

Latency Analysis of Serverless Computing

Summer Internship Report

By

Siddhartha Reddy Keesara

May 8, 2019 - June 20, 2019

Under the mentorship of

Dr. Soumya K. Ghoush

Dr. Sourav Kanti Addya



Department of Computer Science and Engineering

Indian Institute of Technology Kharagpur

Kharagpur-721302, India

June, 2019

1. INTRODUCTION

Amazon Web Services (AWS) is a platform of web services offering solutions for computing, storing, and networking, at different layers of abstraction. You can use these services to host web sites, run enterprise applications, and mine tremendous amounts of data. The term web service means services can be controlled via a web interface. The web interface can be used by machines or by humans via a graphical user interface. The most prominent services are EC2, which offers virtual servers, and S3, which offers storage capacity. Services on AWS work well together; you can use them to replicate your existing on-premises setup or design a new setup from scratch. Services are charged for on a pay-per-use pricing model.

As an AWS customer, you can choose among different data centers. AWS data centers are distributed in the United States, Europe, Asia, and South America. For example, you can start a virtual server in Japan in the same way you can start a virtual server in Ireland. This enables you to serve customers worldwide with a global infrastructure.

1.1 WHAT IS CLOUD COMPUTING?

Cloud computing is an internet-based computing service in which large groups of remote servers are networked to allow centralized data storage, and online access to computer services or resources.

Using cloud computing, organizations can use shared computing and storage resources rather than building, operating, and improving infrastructure on their own.

Cloud computing is a model that enables the following features.

- Users can provision and release resources on-demand.
- Resources can be scaled up or down automatically, depending on the load.
- Resources are accessible over a network with proper security.
- Cloud service providers can enable a pay-as-you-go model, where customers are charged based on the type of resources and per usage.

1.2 CLOUD COMPUTING DEPLOYMENT MODELS

Private cloud services are delivered from a business's data center to internal users. This model offers the versatility and convenience of the cloud, while preserving the management, control and security common to local data centers. Internal users may or may not be billed for services through IT chargeback. Common private cloud technologies and vendors include VMware and OpenStack.

Public cloud is a third-party cloud service provider delivers the cloud service over the internet. Public cloud services are sold on demand, typically by the minute or hour, though long-term commitments are available for many services. Customers only pay for the CPU cycles, storage or bandwidth they consume. Leading public cloud service providers include Amazon Web Services (AWS), Microsoft Azure, IBM and Google Cloud Platform.

Hybrid cloud is a combination of public cloud services and an on-premises private cloud, with orchestration and automation between the two. Companies can run mission-critical workloads or sensitive applications on the private cloud and use the public cloud to handle workload bursts or spikes in demand. The goal of a

hybrid cloud is to create a unified, automated, scalable environment that takes advantage of all that a public cloud infrastructure can provide, while still maintaining control over mission-critical data.

1.3 CLOUD SERVICE MODELS

There are three types of service models in cloud – IaaS, PaaS, and SaaS.

IaaS

IaaS stands for **Infrastructure as a Service**. It provides users with the capability to provision processing, storage, and network connectivity on demand. Using this service model, the customers can develop their own applications on these resources.

PaaS

PaaS stands for **Platform as a Service**. Here, the service provider provides various services like databases, queues, workflow engines, e-mails, etc. to their customers. The customer can then use these components for building their own applications. The services, availability of resources and data backup are handled by the service provider that helps the customers to focus more on their application's functionality.

SaaS

SaaS stands for **Software as a Service**. As the name suggests, here the third-party providers provide end-user applications to their customers with some administrative capability at the application level, such as the ability to create and manage their users. Also some level of customizability is possible such as the customers can use their own corporate logos, colors, etc.

1.4 WHAT IS SERVERLESS COMPUTING?

Serverless most often refers to serverless applications. Serverless applications are the ones that don't require you to provision or manage any servers. You can focus on your core product and business logic instead of responsibilities like operating system (OS) access control, OS patching, provisioning, right-sizing, scaling, and availability. By building your application on a serverless platform, the platform manages these responsibilities for you.

For service or platform to be considered serverless, it should provide the following capabilities:

No server management – You don't have to provision or maintain any servers. There is no software or runtime to install, maintain, or administer.

Flexible scaling – You can scale your application automatically or by adjusting its capacity through toggling the units of consumption (for example, throughput, memory) rather than units of individual servers.

High availability – Serverless applications have built-in availability and fault tolerance. You don't need to architect for these capabilities because the services running the application provide them by default.

No idle capacity – You don't have to pay for idle capacity. There is no need to pre-provision or over-provision capacity for things like compute and storage. There is no charge when your code isn't running.

The AWS Cloud provides many different services that can be components of a serverless application. These include capabilities for:

Compute – AWS Lambda

APIs – Amazon API Gateway

Storage – Amazon Simple Storage Service (Amazon S3)

Databases –Amazon DynamoDB

Interprocess messaging – Amazon Simple Notification Service (Amazon SNS) and Amazon Simple Queue Service (Amazon SQS)

Orchestration – AWS Step Functions and Amazon CloudWatch Events

Analytics – Amazon Kinesis

1.5.1 PROS OF SERVERLESS COMPUTING

Cheaper than the traditional cloud - FaaS allows you to pay a fraction of the price per request. If you're a startup, you can build an MVP nearly for free and ease into the market without dealing with huge bills for minimum traffic.

Scalable - Applications get high availability and auto-scalability without additional effort from the developer. This can significantly reduce development time and consequently costs.

Lower human resources costs - Just as you don't have to spend hundreds or thousands of dollars on hardware, you can stop paying engineers for maintaining it.

Ability to focus on user experience - Abstraction from servers allows companies to dedicate more time and resources to developing and improving customer-facing features.

1.5.2 CONS OF SERVERLESS COMPUTING

Vendor lock-in - When you give a vendor the reins to control your operations, you have to play by their rules. It's also not easy to port your application to Azure if you've already set it up on Lambda. The same concern refers to coding languages: Right now, only Node.js and Python developers have the freedom to choose between existing serverless options.

Learning curve - Despite the comprehensive documentation and community resources, you may soon find out that the learning curve for FaaS tools is pretty steep. Also, to painlessly migrate to serverless, you might want to split your monolith into microservices, another complicated task to tackle. That's why it's preferable to get help from professionals experienced in serverless tools.

Unsuitable for long term tasks - Lambda gives you five minutes to execute a task and if it takes longer, you'll have to call another function. Serverless is great for short real-time or near-real-time processes like

sending out emails. But long duration operations such as uploading video files would require additional FaaS functions or be better with “server-ful” architecture.

1.7 USE CASES OF SERVERLESS COMPUTING

Currently, most of the technology adopters are startups who seek for a possibility to scale painlessly and lower the entrance barrier. Serverless is also a perfect approach for applications that don’t run continuously but rather have quiet periods and peaks of traffic

Internet of Things applications - The real-time response nature of the serverless approach works great for IoT use cases. Motion activated cameras that we’ve already mentioned, along with applications that react to changes in weather, temperature, or health conditions are perfect for the serverless paradigm that won’t allow your services to sit idle 24/7.

Virtual assistants and chatbots - People using chats expect immediate responses which is why serverless data processing can be faster. As your application grows from one hundred to several thousand users, your processing time should also stay the same which is automated with FaaS.

Image-rich applications - To maintain great user experience, developers have to provide multiple versions of the same images for different screen sizes – from desktops, tablets and smartphones. This significantly decreases loading time. However, the tooling from AWS and Google will automatically optimize your images for any needs, making it a perfect solution for image-heavy applications.

Agile and Continuous Integration pipelines - The idea of running the code only when a certain event is triggered is perfectly in line with Agile or Continuous Integration principles. Separating your codebase into functions also helps with bug fixing and shipping updates. Serverless is an overall friendly way for maximum automation and rapid deployment processes.

2. SERVICES USED IN AWS MANAGEMENT CONSOLE

The AWS Management Console is a web application that comprises and refers to a broad collection of service consoles for managing Amazon Web Services. In this chapter, we will see the list of various services which are used to build this application from Amazon Web Services Management Console. The AWS Management Console provides multiple ways for navigating to individual service consoles.

We will learn about the following services in this chapter:

- AWS Lambda
- API Gateway
- AWS Identity and Access Management (IAM)
- Amazon Simple Storage Service(S3)
- DynamoDB
- CloudWatch

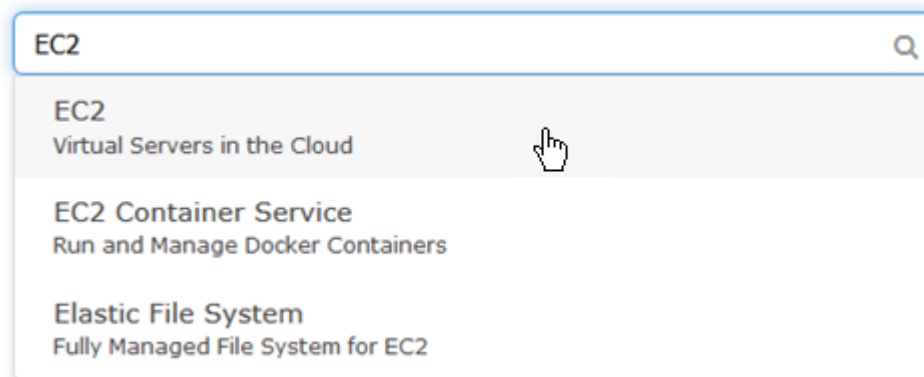
Getting Started with a Service

The AWS Management Console provides multiple ways for navigating to individual service consoles.

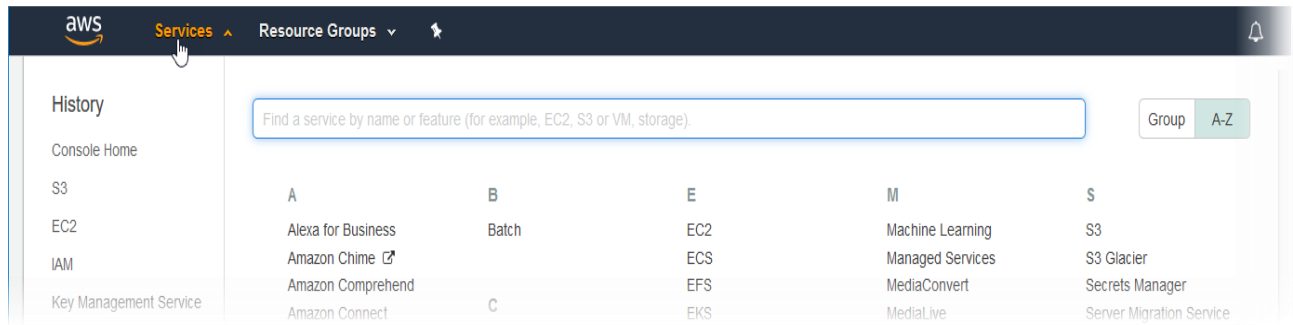
To open a console for a service

Do one of the following:

- Enter the name of the service in the search box. Then choose the service that you want from the list of search results.



- Choose one of your recently visited services under the search box.
- Choose **Services** to open a full list of services. On the upper right of the page, choose **Group** to see the services listed by category or choose **A–Z** to see an alphabetical listing. Then choose the service that you want.



2.1 AWS LAMBDA

Lambda can be described as a type of serverless Function-as-a-Service (FaaS). FaaS is one approach to building event-driven computing systems. It relies on functions as the unit of deployment and execution. Serverless FaaS is a type of FaaS where no virtual machines or containers are present in the programming model and where the vendor provides provision-free scalability and built-in reliability. Figure 2.1 shows the relationship among event-driven computing, FaaS, and serverless FaaS.

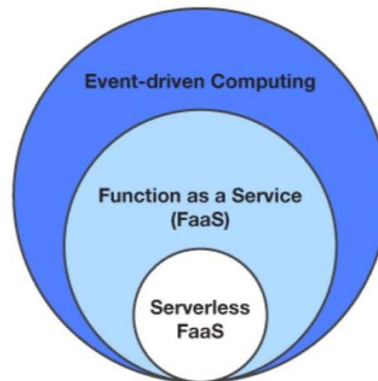


Figure 2.1: The relationship among event-driven computing, FaaS, and serverlessFaaS

With Lambda, you can run code for virtually any type of application or backend service. Lambda runs and scales your code with high availability.

Each Lambda function you create contains the code you want to execute, the configuration that defines how your code is executed and, optionally, one or more event sources that detect events and invoke your function as they occur.

You don't need to write any code to integrate an event source with your Lambda function, manage any of the infrastructure that detects events and delivers them to your function, or manage scaling your Lambda function to match the number of events that are delivered. You can focus on your application logic and configure the event sources that cause your logic to run. Your Lambda function runs within a (simplified) architecture that looks like the one shown in Figure 2.2



Figure 2.2: Simplified architecture of a running Lambda function

2.2 API GATEWAY

Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing REST and WebSocket APIs at any scale. API developers can create APIs that access AWS or other web services as well as data stored in the AWS Cloud. As an API Gateway API developer, you can create APIs for use in your own client applications (apps). Or you can make your APIs available to third-party app developers.

API Gateway creates REST APIs that:

- Are HTTP-based.
- Adhere to the REST protocol, which enables stateless client-server communication.
- Implement standard HTTP methods such as GET, POST, PUT, PATCH and DELETE.

API Gateway creates WebSocket APIs that:

- Adhere to the WebSocket protocol, which enables stateful, full-duplex communication between client and server.
- Route incoming messages and based on message content.

The following diagram shows API Gateway architecture.

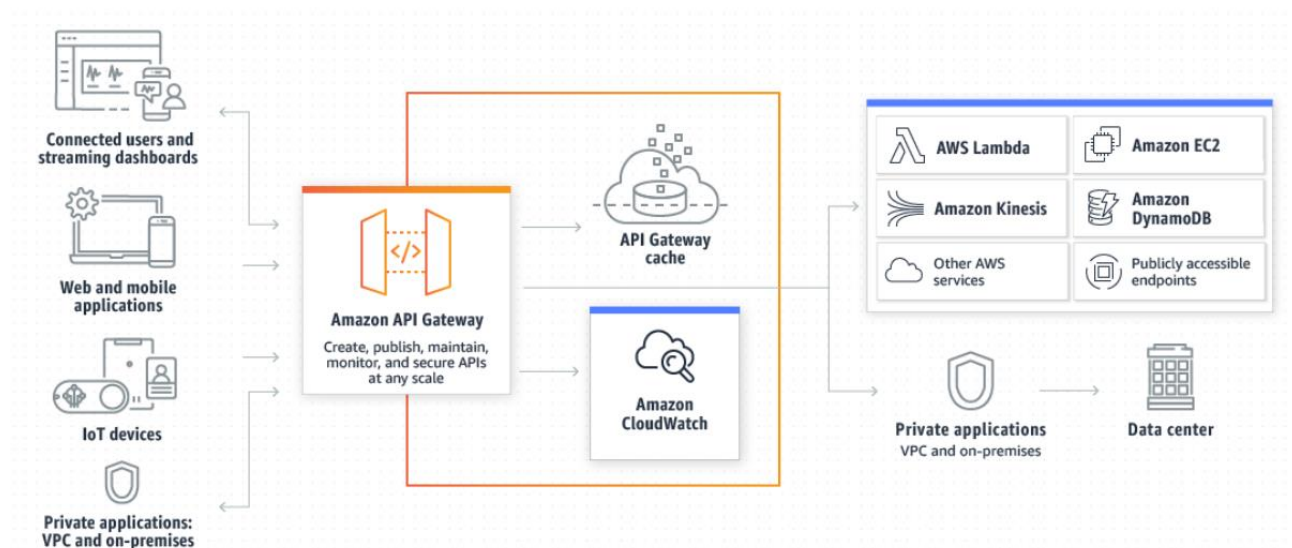


Figure 2.3: Architecture of API Gateway

This diagram illustrates how the APIs you build in Amazon API Gateway provide you or your developer customers with an integrated and consistent developer experience for building AWS serverless applications. API Gateway handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, including traffic management, authorization and access control, monitoring, and API version management. API Gateway acts as a "front door" for applications to access data, business logic, or functionality from your backend services, such as workloads running on Amazon Elastic Compute Cloud (Amazon EC2), code running on AWS Lambda, any web application, or real-time communication applications.

2.3 AWS IDENTITY AND ACCESS MANAGEMENT (IAM)

AWS Identity and Access Management (IAM) is a web service that helps you securely control access to AWS resources. You use IAM to control who is authenticated (signed in) and authorized (has permissions) to use resources.

When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account root user is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

2.4 AMAZON SIMPLE STORAGE SERVICE(S3)

Amazon Simple Storage Service is storage for the Internet. It is designed to make web-scale computing easier for developers.

Amazon S3 has a simple web services interface that you can use to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, fast, inexpensive data storage infrastructure that Amazon uses to run its own global network of websites. The service aims to maximize benefits of scale and to pass those benefits on to developers.

Amazon S3 is intentionally built with a minimal feature set that focuses on simplicity and robustness. Following are some of the advantages of the Amazon S3 service:

Create Buckets – Create and name a bucket that stores data. Buckets are the fundamental container in Amazon S3 for data storage.

Store data in Buckets – Store an infinite amount of data in a bucket. Upload as many objects as you like into an Amazon S3 bucket. Each object can contain up to 5 TB of data. Each object is stored and retrieved using a unique developer-assigned key.

Download data – Download your data or enable others to do so. Download your data any time you like or allow others to do the same.

Permissions – Grant or deny access to others who want to upload or download data into your Amazon S3 bucket. Grant upload and download permissions to three types of users. Authentication Mechanisms can help keep data secure from unauthorized access.

Standard interfaces – Use standards-based REST and SOAP interfaces designed to work with any Internet-development toolkit.

2.5 DYNAMO DB

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. DynamoDB lets you offload the administrative burden of operating and scaling a distributed database, so that you don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling. Also, DynamoDB offers encryption at rest, which eliminates the operational burden and complexity involved in protecting sensitive data.

With DynamoDB, you can create database tables that can store and retrieve any amount of data, and serve any level of request traffic. You can scale up or scale down your tables' throughput capacity without downtime or performance degradation, and use the AWS Management Console to monitor resource utilization and performance metrics.

Amazon DynamoDB provides on-demand backup capability. It allows you to create full backups of your tables for long-term retention and archival for regulatory compliance needs.

2.6 CLOUDWATCH

Amazon CloudWatch monitors your Amazon Web Services (AWS) resources and the applications you run on AWS in real time. You can use CloudWatch to collect and track metrics, which are variables you can measure for your resources and applications.

The CloudWatch home page automatically displays metrics about every AWS service you use. You can additionally create custom dashboards to display metrics about your custom applications, and display custom collections of metrics that you choose.

You can create alarms which watch metrics and send notifications or automatically make changes to the resources you are monitoring when a threshold is breached. For example, you can monitor the CPUUsage and disk reads and writes of your Amazon EC2 instances and then use this data to determine whether you should launch additional instances to handle increased load. You can also use this data to stop under-used instances to save money.

With CloudWatch, you gain system-wide visibility into resource utilization, application performance, and operational health.

3. ANALYSIS OF APPLICATION

3.1 API TO GET CUSTOMER DETAILS

We will start building our application by creating a Lambda Function for a specific region which is integrated to an API Gateway which is deployed and the data inserted will be saved in DynamoDB and the logs are accessed from CloudWatch for various AWS regions.

3.1.1 Creating a new GET API

In this step, you create a simple GET API in the API Gateway console and attach your Lambda function to it as a backend.

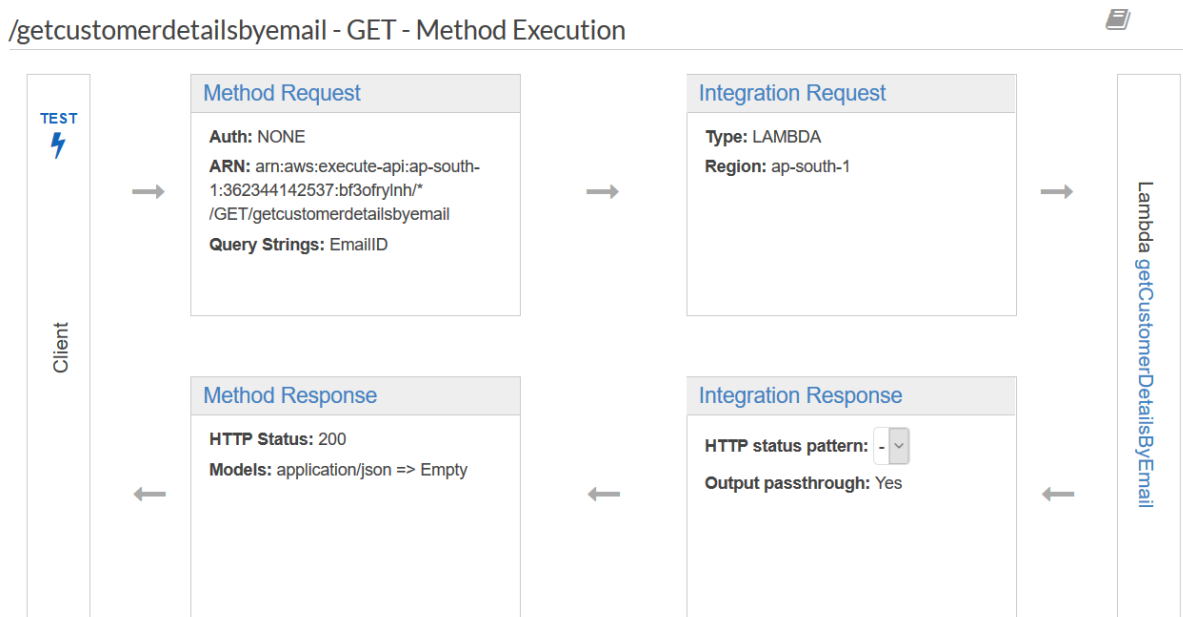
1. From the Services menu, choose API Gateway to go to the API Gateway console.
2. If this is your first time using API Gateway, you see a page that introduces you to the features of the service. Choose Get Started. When the Create Example API popup appears, choose OK.



3. Under Choose the protocol, choose REST.
4. Under Create new API, choose New API.
5. Under Settings:
 - For API name, enter my-api.
 - If desired, enter a description in the Description field; otherwise, leave it empty.
 - Leave Endpoint Type set to Regional.
6. Choose Create API.

7. Under Resources, you'll see nothing but /. This is the root-level resource, which corresponds to the base path URL for your API. From the Actions drop down menu, choose Create Method.
8. Under the resource name (/), you'll see a dropdown menu. Choose GET and then choose the check mark icon to save your choice.
9. In the / – GET – Setup pane, for Integration type, choose Lambda Function.
10. Choose Use Lambda proxy integration.
11. For Lambda Region, choose the Region where you created your Lambda function.
12. In the Lambda Function field, type any character and then choose my-function.
13. When the Add Permission to Lambda Function popup appears, choose OK to grant API Gateway that permission.

Now you'll see a / – GET – Method Execution pane:



3.1.2 Deploying GET API

Once you complete the above steps, you've created an API, but you can't actually use it yet. This is because it needs to be deployed.

1. From the Actions dropdown menu, choose Deploy API.
2. From the Deployment stage dropdown menu, choose [New Stage].
3. For Stage name, enter prod.
4. Choose Deploy.

If you have cURL installed on your computer, you can test your API as follows:

1. Open a terminal window.
2. Copy the following cURL command and paste it into the terminal window, replacing [bf3ofrylnh](#) with your API's API ID and [ap-south-1](#) with the Region where your API is deployed.

Windows

```
curl -X GET "https://bf3ofrylnh.execute-api.ap-south-1.amazonaws.com/Dev"
```

This command returns the following output: "Hello from Lambda!"

3.2 LAMBDA FUNCTION TO GET CUSTOMER DETAILS

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.

3.2.1 Creating Lambda Function to Get Details

To create a Lambda function

1. Open the AWS Lambda console.
2. Choose Create a function.
3. For Function name, enter my-function.
4. Choose Create function.



```
1
2 const AWS = require('aws-sdk');
3 var docClient = new AWS.DynamoDB.DocumentClient();
4
5 var tableName = "CustomerDetails";
6
7 exports.handler = (event, context, callback) => {
8   console.log(event.EmailID)
9
10  var params = {
11    TableName : tableName,
12    Key:{
13      "EmailID" : event.EmailID
14    }
15  }
16
17  var start = new Date().getTime();
18  docClient.get(params, function(err,data){
19    var end = new Date().getTime();
20    var time = end - start;
21    callback(err, data +time)
22  })
23
24
25 };
26
```

3.2.2 Invoking Get Details Lambda Function

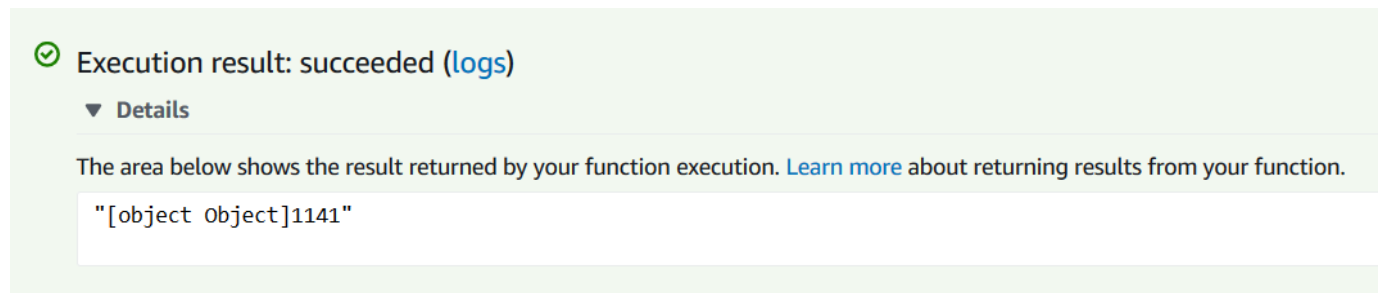
To invoke a function

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:

```
1 {  
2   "EmailID": "keesarasid@gmail.com"  
3 }
```

You can change key and values in the sample JSON, but don't change the event structure. If you do change any keys and values, you must update the sample code accordingly.

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.
4. AWS Lambda executes your function on your behalf. The handler in your Lambda function receives and then processes the sample event.
5. Upon successful execution, view results in the console.



3.3 API TO POST CUSTOMER DETAILS

We will start building our application by creating a Lambda Function for a specific region which is integrated to an API Gateway which is deployed and the data inserted will be saved in DynamoDB and the logs are accessed from CloudWatch for various AWS regions.

3.3.1 Creating a new POST API

In this step, you create a simple POST API in the API Gateway console and attach your Lambda function to it as a backend.

1. From the Services menu, choose API Gateway to go to the API Gateway console.
2. If this is your first time using API Gateway, you see a page that introduces you to the features of the service. Choose Get Started. When the Create Example API popup appears, choose OK.
3. Under Choose the protocol, choose REST.

4.Under Create new API, choose New API.

5.Under Settings:

- For API name, enter my-api.
- If desired, enter a description in the Description field; otherwise, leave it empty.
- Leave Endpoint Type set to Regional.

6.Choose Create API.

7.Under Resources, you'll see nothing but /. This is the root-level resource, which corresponds to the base path URL for your API.From the Actions drop down menu, choose Create Method.

8.Under the resource name (/), you'll see a dropdown menu. Choose POST and then choose the check mark icon to save your choice.

9.In the / – POST– Setup pane, for Integration type, choose Lambda Function.

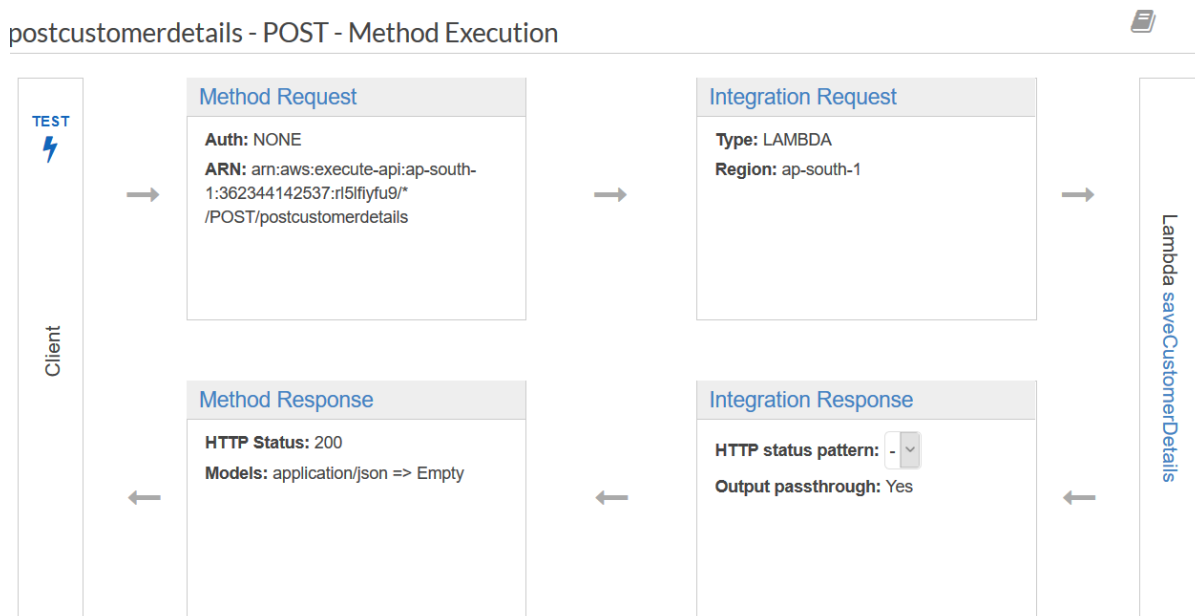
10.Choose Use Lambda proxy integration.

11.For Lambda Region, choose the Region where you created your Lambda function.

12.In the Lambda Function field, type any character and then choose my-function.

13.When the Add Permission to Lambda Function popup appears , choose OK to grant API Gateway that permission.

Now you'll see a / – POST– Method Execution pane:



3.3.2 Deploying POST API

Once you complete the above steps, you've created an API, but you can't actually use it yet. This is because it needs to be deployed.

1. From the Actions dropdown menu, choose Deploy API.
2. From the Deployment stage dropdown menu, choose [New Stage].
3. For Stage name, enter prod.
4. Choose Deploy.
5. In the prod Stage Editor, note the Invoke URL at the top. It should be in this format: (<https://rl5lfiyfu9.execute-api.ap-south-1.amazonaws.com/dev>). If you choose the Invoke URL.

If you have cURL installed on your computer, you can test your API as follows:

1. Open a terminal window.
2. Copy the following cURL command and paste it into the terminal window, replacing [rl5lfiyfu9](#) with your API's API ID and [ap-south-1](#) with the Region where your API is deployed.

Windows

```
curl -X POST "https://rl5lfiyfu9.execute-api.ap-south-1.amazonaws.com/dev"
```

This command returns the following output: "Hello from Lambda!"

3.4 LAMBDA FUNCTION TO POST CUSTOMER DETAILS

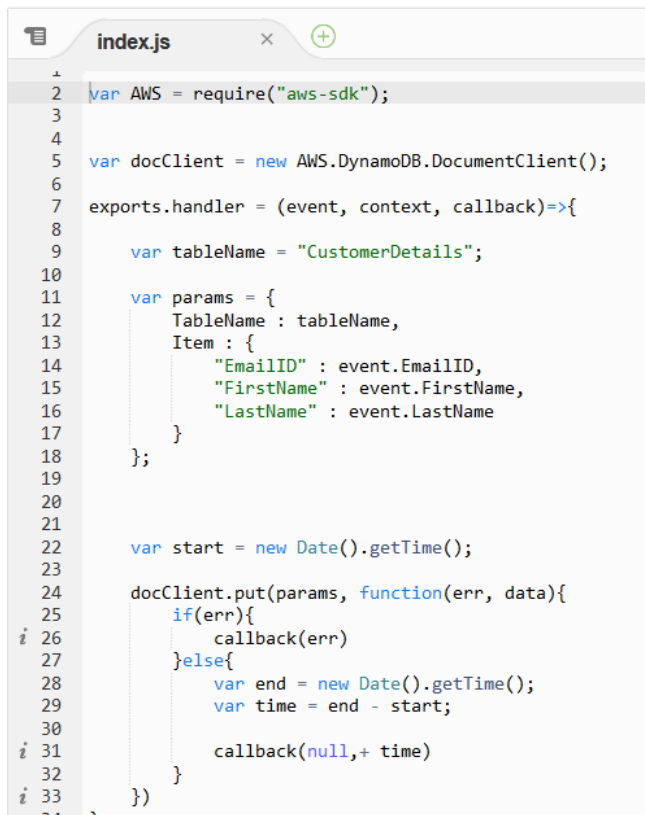
AWS Lambda executes the Lambda Function and returns results. You then verify execution results, including the logs that your Lambda Function created and various CloudWatch metrics.

3.4.1 Creating Lambda Function to Post Details

To create a Lambda function

1. Open the AWS Lambda console.
2. Choose Create a function.
3. For Function name, enter my-function.
4. Choose Create function.

Lambda creates a Node.js function and an execution role that grants the function permission to upload logs. Lambda assumes the execution role when you invoke your function, and uses it to create credentials for the AWS SDK and to read data from event sources.



```

1
2 var AWS = require("aws-sdk");
3
4
5 var docClient = new AWS.DynamoDB.DocumentClient();
6
7 exports.handler = (event, context, callback)=>{
8
9     var tableName = "CustomerDetails";
10
11     var params = {
12         TableName : tableName,
13         Item : {
14             "EmailID" : event.EmailID,
15             "FirstName" : event.FirstName,
16             "LastName" : event.LastName
17         }
18     };
19
20
21
22     var start = new Date().getTime();
23
24     docClient.put(params, function(err, data){
25         if(err){
26             callback(err)
27         }else{
28             var end = new Date().getTime();
29             var time = end - start;
30
31             callback(null, + time)
32         }
33     })
34 }

```

3.4.2 Invoking Post Details Lambda Function

To invoke a function

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:



```

1 {
2   "EmailID": "test@test.com",
3   "FirstName": "Test",
4   "LastName": "Post"
5 }

```

You can change key and values in the sample JSON, but don't change the event structure. If you do change any keys and values, you must update the sample code accordingly.

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.
4. AWS Lambda executes your function on your behalf. The handler in your Lambda function receives and then processes the sample event.
5. Upon successful execution, view results in the console.

3.5 SETTING UP A DYNAMO DB

1. Create a table by accessing the console at <https://console.aws.amazon.com/dynamodb>.
2. Then choose the “Create Table” option.



3. . In the Create Table screen, enter the table name within the table name field; enter the primary key (ID) within the partition key field; and enter “Number” for the data type.

The screenshot shows the 'Create Table' screen in the Amazon DynamoDB console. It includes a 'Table Name' field with a placeholder 'Table will be created in us-east-1 region'. Below this is the 'Primary Key' section, which states: 'DynamoDB is a schema-less database. You only need to tell us your primary key attribute(s)'. It offers two options for 'Primary Key Type': 'Hash and Range' (selected) and 'Hash'. Under 'Hash and Range', there are two fields: 'Hash Attribute Name' and 'Range Attribute Name'. Each field has radio buttons for 'String', 'Number', and 'Binary'. The 'Number' option is selected for both. A warning icon and text are present at the bottom: 'Choose a hash attribute that ensures that your workload is evenly distributed across hash keys. For example, "Customer ID" is a good hash key, while "Game ID" would be a bad choice if most of your traffic relates to a few popular games. Learn more about choosing your primary key'.

4. After entering all information, select Create.

3.5.1 Creating a Table in DynamoDB

In this step, you create a CustomerDetails table in Amazon DynamoDB. The table has the following details:

- Partition key —EmailID
- Sort key —null

To create a new Music table using the DynamoDB console:

1. Sign in to the AWS Management Console and open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose Dashboard.
3. On the right side of the console choose Create Table.
4. Enter the table details as follows:
 - a. For the table name, enter CustomerDetails.
 - b. For the partition key, enter EmailID.
 - c. Choose Add sort key.
 - d. Enter null as the sort key.
5. Choose Create to create the table.

Create DynamoDB table

DynamoDB is a schema-less database that only requires a table name and primary key uniquely identify items, partition the data, and sort data within each partition.

Table name* ⓘ

Primary key* **Partition key**

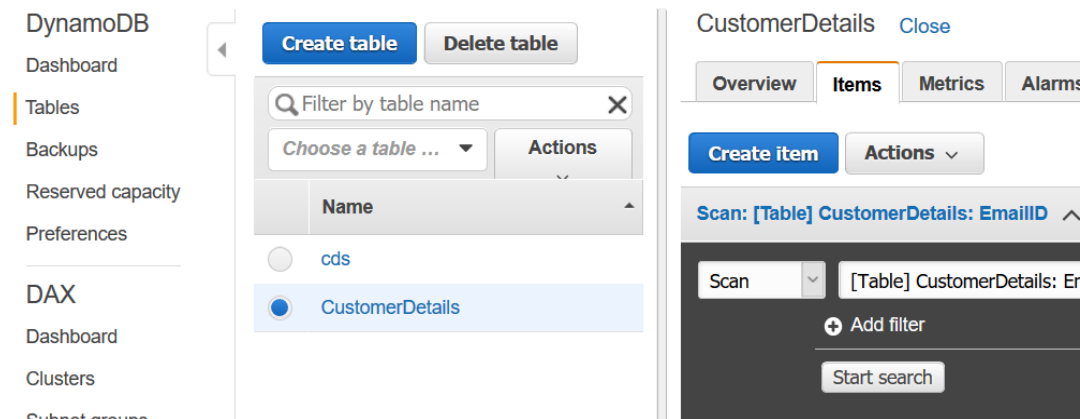
ⓘ

☐ Add sort key

3.5.2 Write Data to the Table

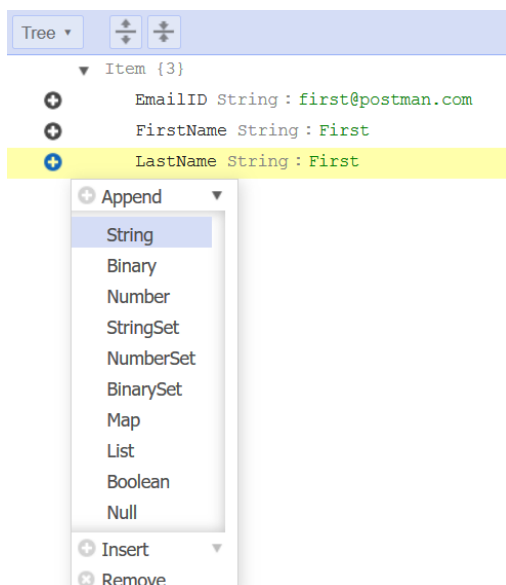
Follow these steps to write data to the CustomerDetails table using the DynamoDB console.

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose Tables.
3. In the table list, choose the CustomerDetails table.
4. Choose the Items tab for the CustomerDetails table.
5. On the Items tab, choose Create item.



6. Choose the plus sign (+) symbol next to EmailID.

7. Choose Append, and then choose String. Name the field FirstName.



8. Repeat this process to create a LastName of type String.

9. Choose Save.

3.5.3 Update Data of Table

You can use the DynamoDB console to update data in the Music table.

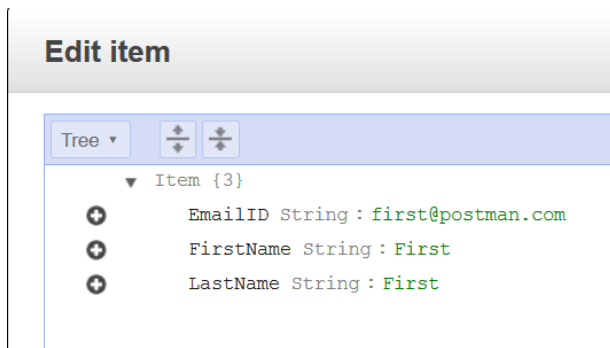
1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.

2. In the navigation pane on the left side of the console, choose Tables.

3. Choose the CustomerDetails table from the table list.

4. Choose the Items tab for the CustomerDetails table.

5. Update the FirstName value to Update FirstName, and then choose Save.



3.6 SETTING UP S3

Amazon S3 has a simple web services interface that you can use to store and retrieve any amount of data, at any time, from anywhere on the web.

3.6.1 Creating a S3 Bucket

First, you need to create an Amazon S3 bucket where you will store your objects.

1. Sign in to the preview version of the AWS Management Console.
2. Under Storage & Content Delivery, choose S3 to open the Amazon S3 console.
3. From the Amazon S3 console dashboard, choose Create Bucket.
4. In Create a Bucket, type a bucket name in Bucket Name.

5. Choose Create.

3.6.2 Upload Files to S3

Upload the data files to the new Amazon S3 bucket.

1. Click the name of the data folder.
2. In the Upload - Select Files wizard, click Add Files.







A file selection dialog box opens.

3. Select all of the files you downloaded and extracted, and then click Open.

3 Files **Size:** 145.9 KB **Target path:** custdetails

To upload a file larger than 80 GB, use the AWS CLI, AWS SDK, or Amazon S3 REST API. [Learn more](#)

[+ Add more files](#)

	index.html - 2.3 KB	
	jquery-3.1.1.min.js - 84.7 KB	
	knockout-3.4.2.js - 58.9 KB	

1. Click Start Upload.

3.8.3 Generate Policy to S3 Bucket

- 1) Select Policy Type

Step 1: Select Policy Type

A Policy is a container for permissions. The different types of policies you can create are an [IAM Policy](#), an [S3 Bucket Policy](#), an [SNS Topic Policy](#), a [VPC Endpoint Policy](#), and an [SQS Queue Policy](#).

Select Type of Policy S3 Bucket Policy

- 2) Add Statement(s)

Step 2: Add Statement(s)

A statement is the formal description of a single permission. See [a description of elements](#) that you can use in statements.

Effect ☒ Allow ☐ Deny

Principal

Use a comma to separate multiple values.

AWS Service Amazon S3 ☐ All Services (**)

Use multiple statements to add permissions for more than one service.

Actions -- Select Actions -- ☐ All Actions (**)

Amazon Resource Name (ARN)

ARN should follow the following format: arn:aws:s3:::<bucket_name>/<key_name>.
Use a comma to separate multiple values.

[Add Conditions \(Optional\)](#)

[Add Statement](#)

- 3) Generate Policy

```

1  {
2    "Id": "Policy1508278368323",
3    "Version": "2012-10-17",
4    "Statement": [
5      {
6        "Sid": "stmt1508278366673",
7        "Action": [
8          "s3:GetObject"
9        ],
10       "Effect": "Allow",
11       "Resource": "arn:aws:s3:::codewithvijaycustomerdetails/*",
12       "Principal": "*"
13     }
14   ]
15 }
16

```

4. DESIGN / IMPLEMENTATION OF APPLICATION

The coming of Ajax was an important landmark in the history of Web 2.0. Ajax is a group of technologies that enable developers to build interactive, feature-rich web applications. Most of these technologies were available many years before Ajax itself. However, the advent of Ajax represents the transition of the web from static pages that need to be refreshed whenever data was exchanged to dynamic, responsive and interactive user interfaces.

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
  <meta charset="utf-8" />
  <script src="jquery-3.1.1.min.js"></script>
  <script src="knockout-3.4.2.js"></script>
  <script type="text/javascript">

    $(document).ready(function() {
      var customerViewModel = function() {
        var self = this;
        self.firstName = ko.observable("");
        self.lastName = ko.observable("");
        self.emailId = ko.observable("");
        self.searchKey = ko.observable("");
        self.getCustomerDetails = function () {
          $.ajax({
            url: ' https://o6qjc7lg49.execute-api.us-east-1.amazonaws.com/devlop/getcustomerdetailsbyemail',
            cache: false,
            type: 'GET',
            data: { "EmailID": self.searchKey() },
            success: function (data) {
              self.firstName(data.Item.FirstName)
              self.lastName(data.Item.LastName),
              self.emailId(data.Item.EmailID)
            }
          });
        }
      }

      var viewModel = new customerViewModel();
      ko.applyBindings(viewModel);
    });
  </script>
</head>

<body>
  <table>
    <tr>
      <td>Search Key(EmailID):</td>
      <td><input type="text" id="txtSearchKey" data-bind="value : searchKey"/></td>
    </tr>
  </table>

  <br />
  <table id="CustomerDetails">
    <thead>
      <tr>
        <td>First Name:</td>
        <td><label id="firstName" data-bind="text: firstName"/></td>
      </tr>
      <tr>
        <td>Last Name:</td>
        <td><label id="lastName" data-bind="text: lastName"/></td>
      </tr>
      <tr>
        <td>Email:</td>
        <td><label id="emailId" data-bind="text: emailId"/></td>
      </tr>
    </thead>
  </table>

  <br />
  <table>
    <tr>
      <td><input type="button" value="GetCustomerDetails" data-bind="click: $root.getCustomerDetails()"/></td>
    </tr>
  </table>
</body>
</html>
```

4.2 HTTP GET REQUEST

The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data.

4.2.1 GET : ap-south-1(Mumbai)

```
1 import requests
2 import statistics
3 num_req = 100
4 data = []
5 for i in range(num_req):
6     r = requests.get('https://bf3ofrylnh.execute-api.ap-south-1.amazonaws.com/Dev/getcustomerdetailsbyemail?EmailID=keesarasid@gmail.com')
7     print(r.text)
8     data.append(int(r.text.split(" ")[-1][: -1]))
9 print(data)
10 print(statistics.mean(data))
11 print(statistics.stdev(data))
```

4.2.2 GET : us-east-1 (N.Virginia)

```
1 import requests
2 import statistics
3 num_req = 100
4 data = []
5 for i in range(num_req):
6     r = requests.get('https://bf3ofrylnh.execute-api.ap-south-1.amazonaws.com/Dev/getcustomerdetailsbyemail?EmailID=keesarasid@gmail.com')
7     print(r.text)
8     data.append(int(r.text.split(" ")[-1][: -1]))
9 print(data)
10 print(statistics.mean(data))
11 print(statistics.stdev(data))
```

4.2.3 GET : eu-west-2 (London)

```
1 import requests
2 import statistics
3 num_req = 101
4 data = []
5 for i in range(num_req):
6     r = requests.get('https://51kkg4k9ya.execute-api.eu-west-2.amazonaws.com/develop/getcustdetails?EmailID=keesarasid@gmail.com')
7     print(r.text)
8     data.append(int(r.text.split(" ")[-1][: -1]))
9 print(data)
10 print(statistics.mean(data))
11 print(statistics.stdev(data))
```

4.2.4 GET : ap-northeast-2 (Seoul)

```
1 import requests
2 import statistics
3 num_req = 5
4 data = []
5 for i in range(num_req):
6     r = requests.get('https://4jwpcfmeu2.execute-api.ap-northeast-2.amazonaws.com/dev/getcustomerdetailsbyemail?EmailID=keesarasid@gmail.com')
7     print(r.text)
8     data.append(int(r.text.split(" ")[-1][: -1]))
9 print(data)
10 print(statistics.mean(data))
11 print(statistics.stdev(data))
```


4.3 HTTP POST REQUEST

A POST request is used to send data to the server, for example, customer information, file upload, etc. using HTML forms.

4.3.1 POST: ap-south-1(Mumbai)

```
1 import requests
2 import statistics
3 num_req = 101
4 data=[]
5 payload = {'EmailID': 'keesarasid@gmail.com', 'FirstName': 'First', 'LastName': 'Last'}
6 for i in range(num_req):
7     r = requests.post("https://r151fiyfu9.execute-api.ap-south-1.amazonaws.com/dev/postcustomerdetails", json=payload)
8     print(r.text)
9     data.append(int(r.text))
10 print(data)
11 print(statistics.mean(data))
12 print(statistics.stdev(data))
```

4.3.2 POST: us-east-1 (N.Virginia)

```
1 import requests
2 import statistics
3 num_req = 101
4 data=[]
5 payload = {'EmailID': 'keesarasid@gmail.com', 'FirstName': 'First', 'LastName': 'Last'}
6 for i in range(num_req):
7     r = requests.post("https://rb9lhmvjxk.execute-api.us-east-1.amazonaws.com/dev/postcustomerdetails", json=payload)
8     print(r.text)
9     data.append(int(r.text))
10 print(data)
11 print(statistics.mean(data))
12 print(statistics.stdev(data))
```

4.3.3 POST: eu-west-2 (London)

```
1 import requests
2 import statistics
3 num_req = 101
4 data=[]
5 payload = {'EmailID': 'keesarasid@gmail.com', 'FirstName': 'First', 'LastName': 'Last'}
6 for i in range(num_req):
7     r = requests.post("https://l77jrnpwge.execute-api.eu-west-2.amazonaws.com/devlop/postcustdetails", json=payload)
8     print(r.text)
9     data.append(int(r.text))
10 print(data)
11 print(statistics.mean(data))
12 print(statistics.stdev(data))
```

4.3.4 POST: ap-northeast-2 (Seoul)

```
1 import requests
2 import statistics
3 num_req = 100
4 data=[]
5 payload = {'EmailID': 'keesarasid@gmail.com', 'FirstName': 'First', 'LastName': 'Last'}
6 for i in range(num_req):
7     r = requests.post("https://ka906xst5m.execute-api.ap-northeast-2.amazonaws.com/dev/savecustomerdetails", json=payload)
8     print(r.text)
9     data.append(int(r.text))
10 print(data)
11 print(statistics.mean(data))
12 print(statistics.stdev(data))
```

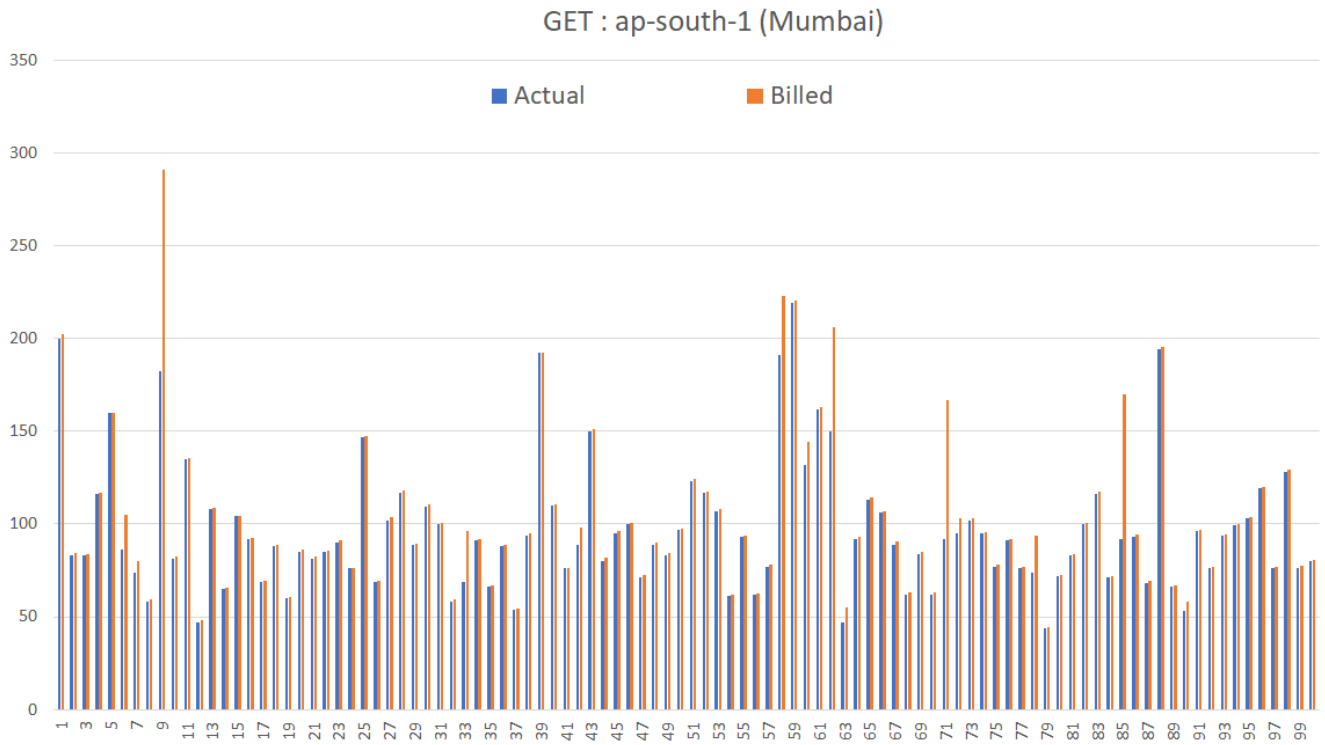


Fig. 1. Actual Time Taken (ms) vs Billed Time (ms) [GET : ap-south-1(Mumbai)]

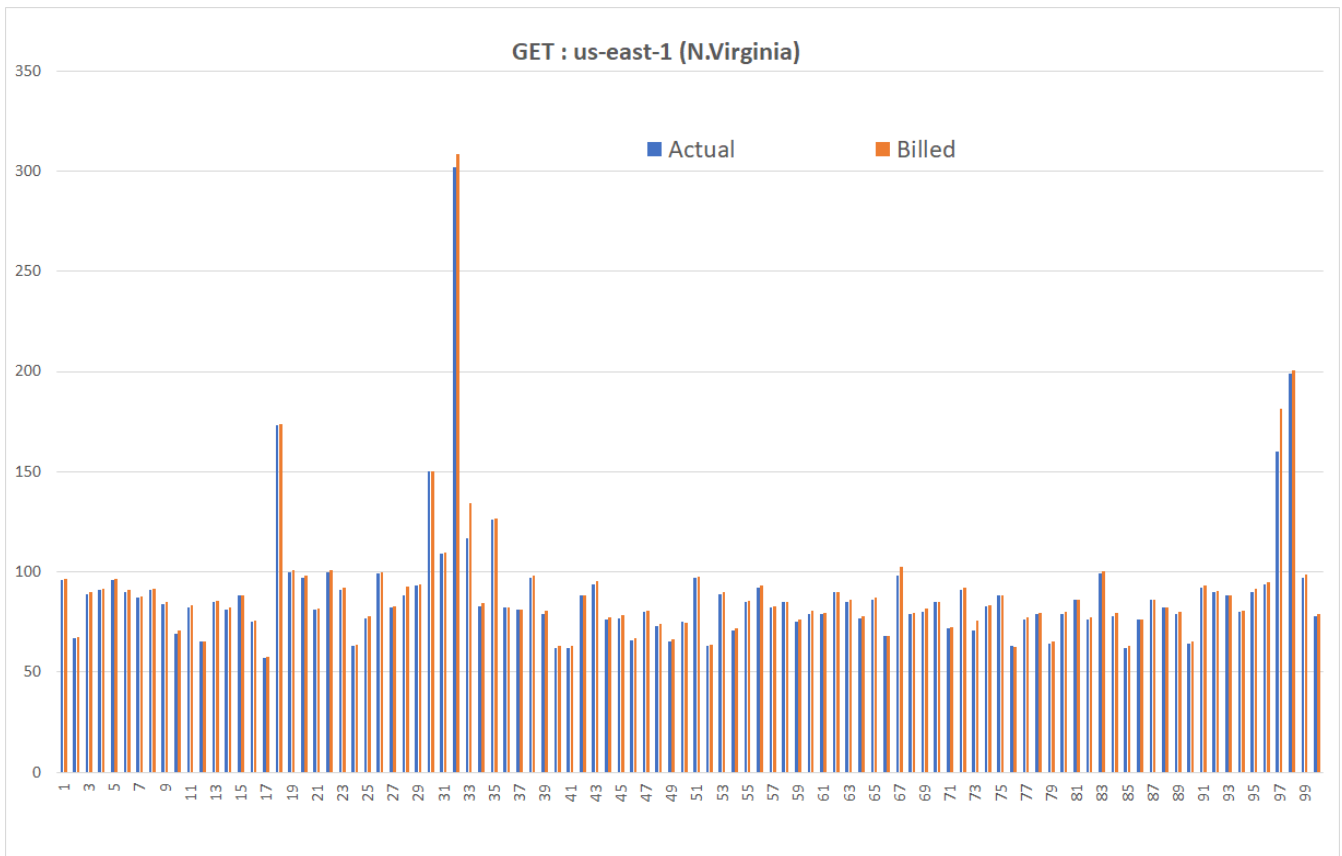


Fig.2. Actual Time Taken (ms) vs Billed Time (ms) [GET : us-east-1 (N.Virginia)]

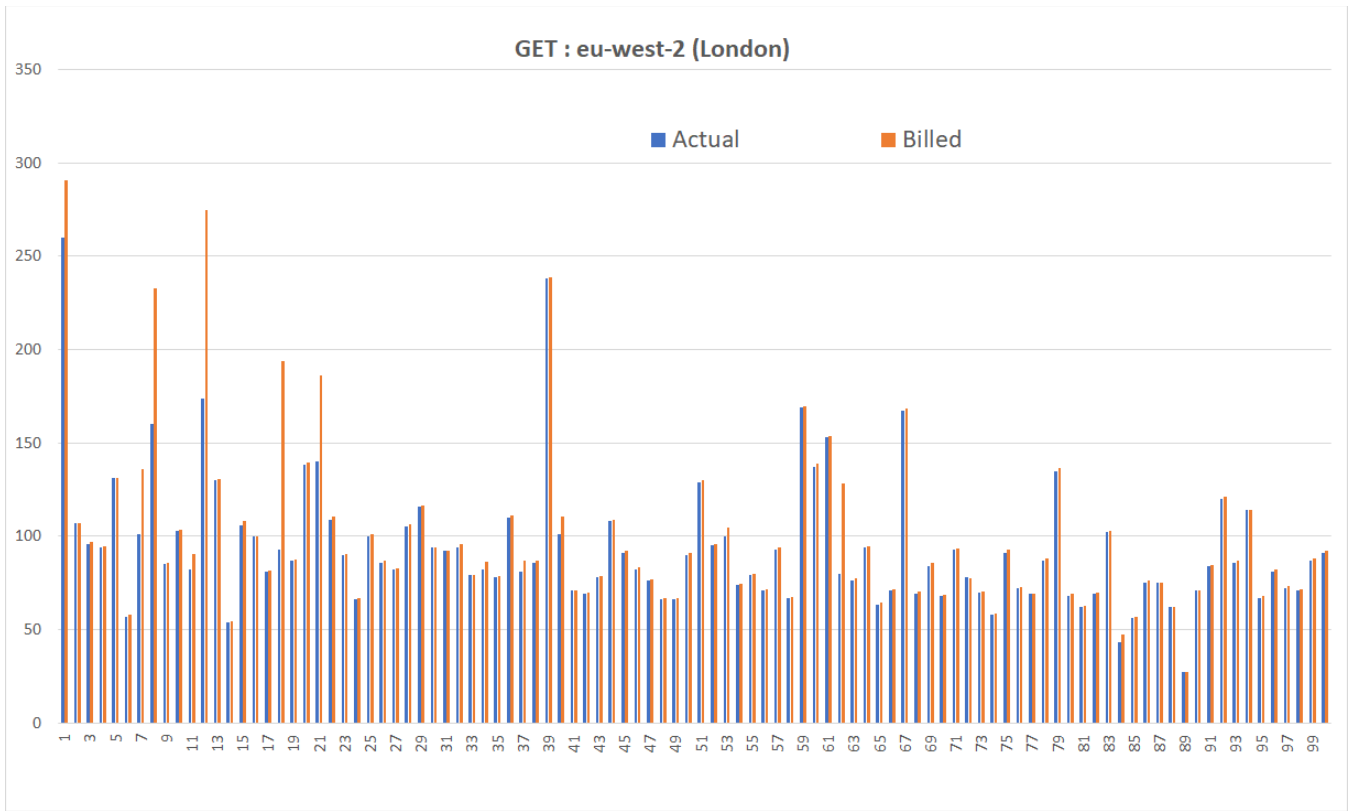


Fig. 3. Actual Time Taken (ms) vs Billed Time (ms) [GET : eu-west-2 (London)]

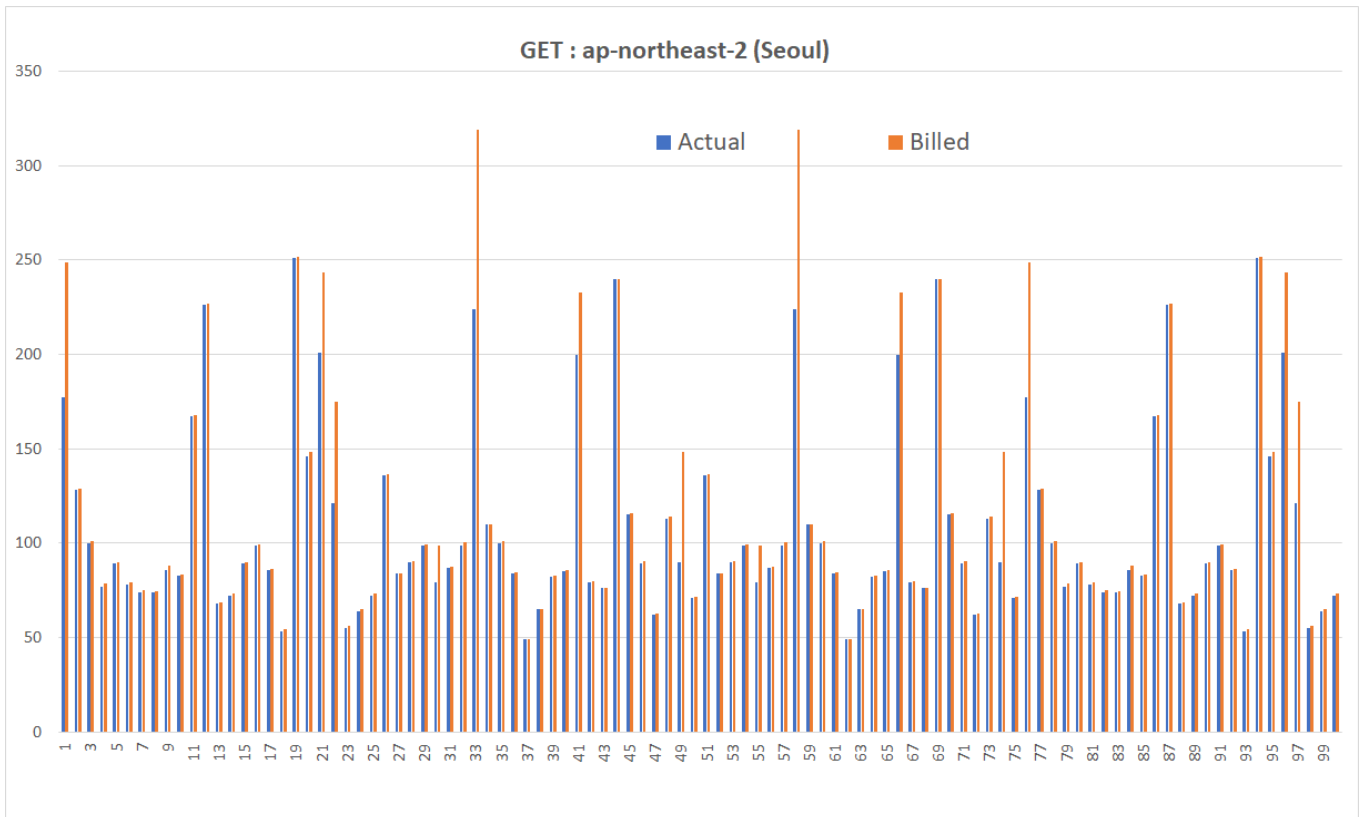


Fig. 4. Actual Time Taken (ms) vs Billed Time (ms) [GET : ap-northeast-2 (Seoul)]

X-AXIS	Total Number of Get Requests
Y-Axis	Time Take (ms)

Fig.1 shows the comparison of time taken for an Http Get request placed from Amazon Lambda Function which is in the region [ap-south-1(Mumbai)] to retrieve data from DynamoDB which located in the region [ap-south-1(Mumbai)].From the graph it is clearly evident that for a few requests there is a slight variation in the actual time taken to retrieve data from DynamoDB to that of duration which in shown in AWS Logs.For a total of 100 Get requests it clearly shows a slight difference of (5.37 ms) in mean and a difference of (6.88 ms) in standard deviation of actual time taken to that of time shown in AWS.

Fig.2 shows the comparison of time taken for an Http Get request placed from Amazon Lambda Function which is in the region [us-east-1 (N.Virginia)] to retrieve data from DynamoDB which located in the region [us-east-1 (N.Virginia)].From the graph it is clearly evident that for a few requests there is a slight variation in the actual time taken to retrieve data from DynamoDB to that of duration which in shown in AWS Logs.For a total of 100 Get requests it clearly shows a slight difference of (1.34 ms) in mean and a difference of (1.14 ms) in standard deviation of actual time taken to that of time shown in AWS.

Fig.3 shows the comparison of time taken for an Http Get request placed from Amazon Lambda Function which is in the region [eu-west-2 (London)] to retrieve data from DynamoDB which located in the region [eu-west-2 (London)].From the graph it is clearly evident that for a few requests there is a slight variation in the actual time taken to retrieve data from DynamoDB to that of duration which in shown in AW Logs.For a total of 100 Get requests it clearly shows a slight difference of (5.4 ms) in mean and a difference of (8.76 ms) in standard deviation of actual time taken to that of time shown in AWS.

Fig.4 shows the comparison of time taken for an Http Get request placed from Amazon Lambda Function which is in the region [ap-northeast-2 (Seoul)] to retrieve data from DynamoDB which located in the region [ap-northeast-2 (Seoul)].From the graph it is clearly evident that for a few requests there is a slight variation in the actual time taken to retrieve data from DynamoDB to that of duration which in shown in AWS Logs.For a total of 100 Get requests it clearly shows a slight difference of (8.18 ms) in mean and a difference of (11.54 ms) in standard deviation of actual time taken to that of time shown in AWS.

TABLE I : Comparison of Mean and Std Dev for 100 HTTP GET Requests

Region	Actual Mean	Billed Mean	Actual Std Dev	Billed Std Dev
ap-south-1(Mumbai)	96.44 ms	101.8 ms	34.89 ms	41.780 ms
us-east-1 (N.Virginia)	88.48 ms	89.82 ms	30.32 ms	31.46 ms
eu-west-2 (London)	93.4 ms	98.8 ms	34.89 ms	43.65 ms
ap-northeast-2 (Seoul)	106.88 ms	115.06 ms	50.76 ms	62.3 ms

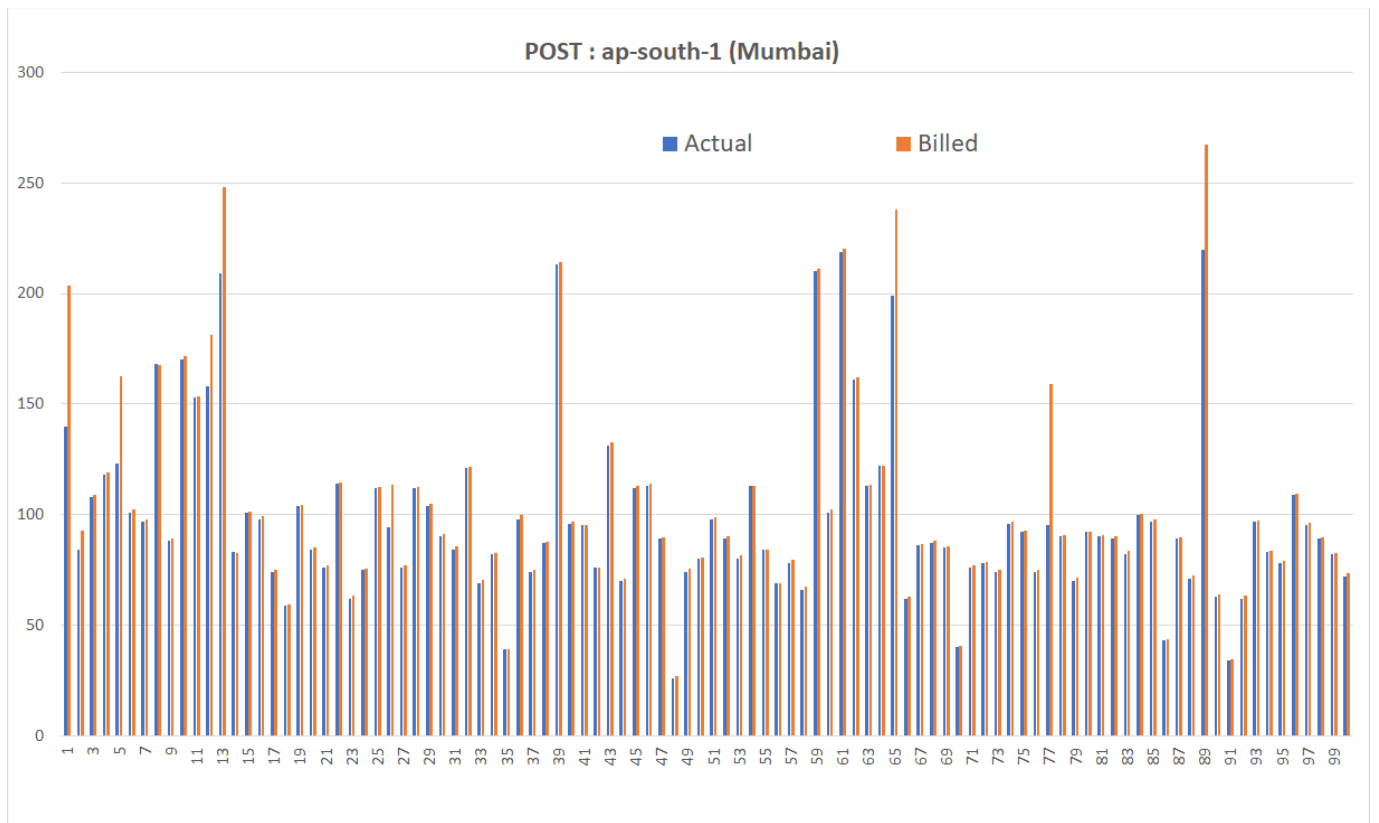


Fig. 5. Actual Time Taken (ms) vs Billed Time (ms) [POST : ap-south-1(Mumbai)]

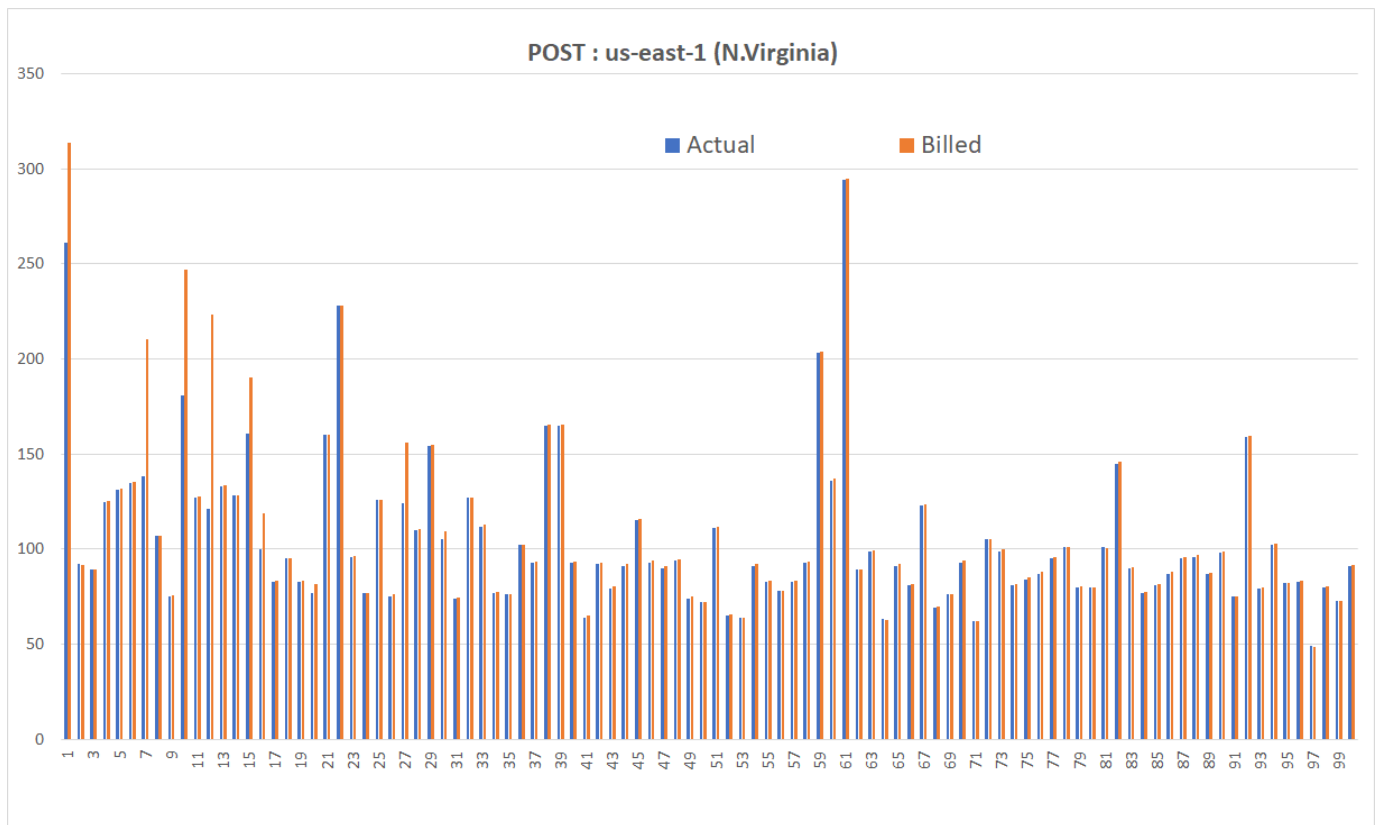


Fig. 6. Actual Time Taken (ms) vs Billed Time (ms) [POST : us-east-1 (N.Virginia)]

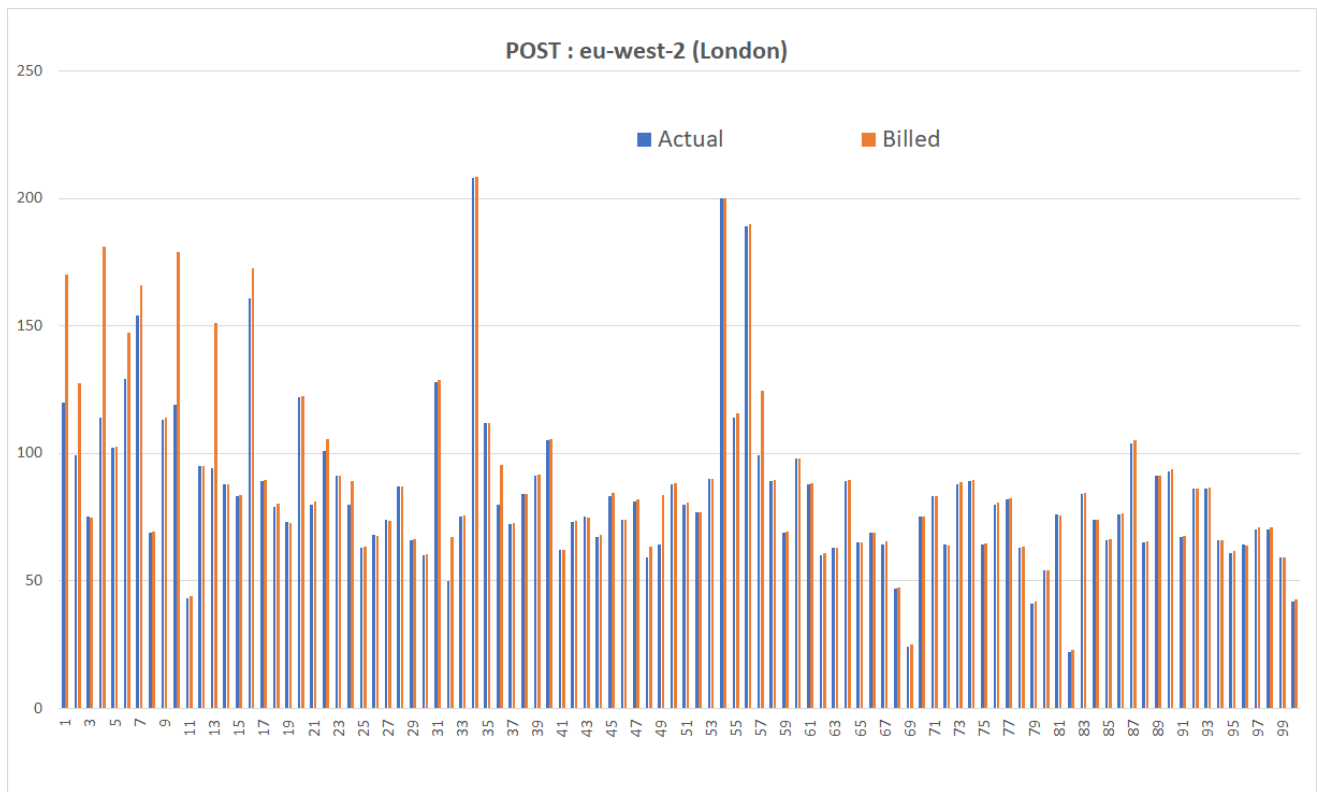


Fig. 7. Actual Time Taken (ms) vs Billed Time (ms) [POST : eu-west-2 (London)]

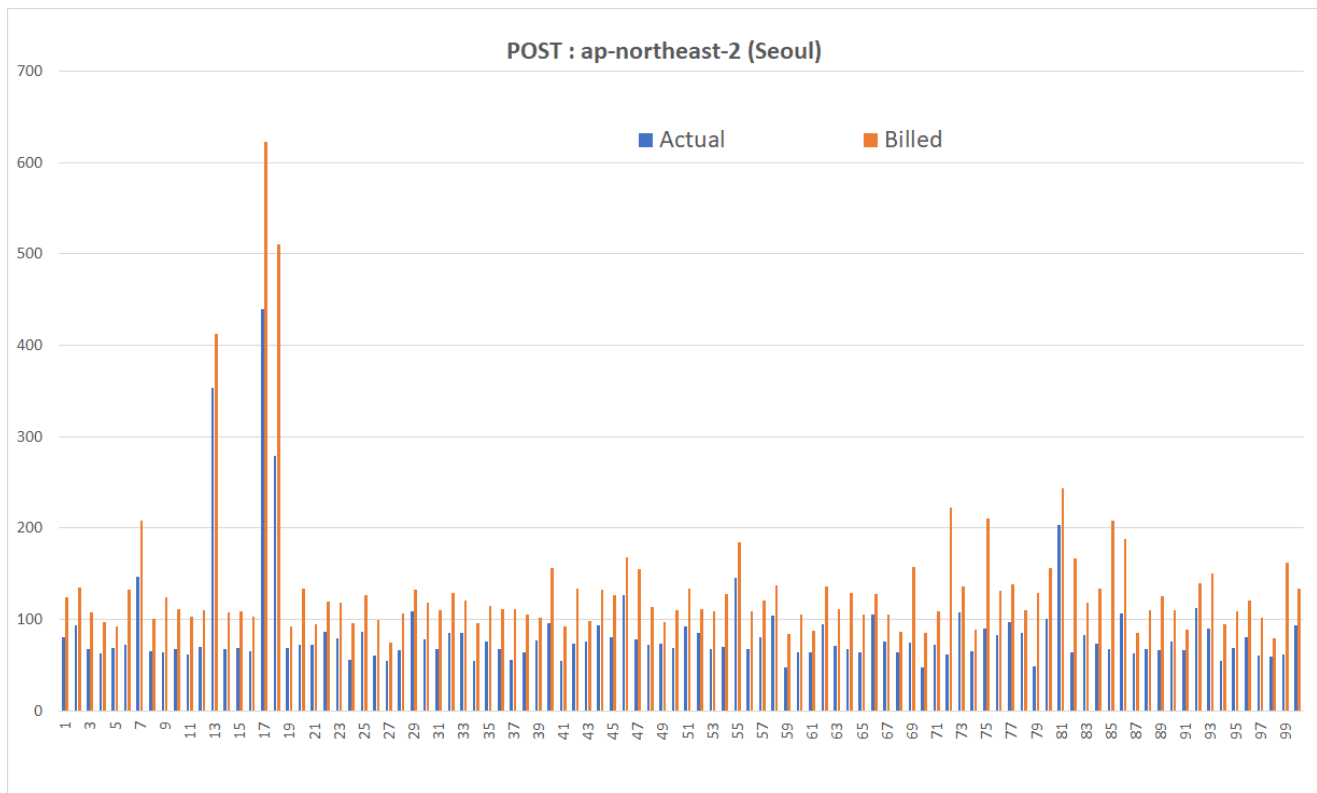


Fig. 8. Actual Time Taken (ms) vs Billed Time (ms) [POST : ap-northeast-2 (Seoul)]

X-AXIS	Total Number of Post Requests
Y-Axis	Time Take (ms)

Fig.5 shows the comparison of time taken for an Http Post request placed from Amazon Lambda Function which is in the region [ap-south-1(Mumbai)] to update data in DynamoDB which located in the region [ap-south-1(Mumbai)].From the graph it is clearly evident that for a few requests there is a slight variation in the actual time taken to update data in DynamoDB to that of duration which in shown in AWS Logs.For a total of 100 Post requests it clearly shows a slight difference of (4.21 ms) in mean and a difference of (6.19 ms) in standard deviation of actual time taken to that of time shown in AWS.

Fig.6 shows the comparison of time taken for an Http Post request placed from Amazon Lambda Function which is in the region [us-east-1 (N.Virginia)] to update data in DynamoDB which located in the region [us-east-1 (N.Virginia)].From the graph it is clearly evident that for a few requests there is a slight variation in the actual time taken to update data in DynamoDB to that of duration which in shown in AWS Logs.For a total of 100 Post requests it clearly shows a slight difference of (4.3 ms) in mean and a difference of (7.22 ms) in standard deviation of actual time taken to that of time shown in AWS.

Fig.7 shows the comparison of time taken for an Http Post request placed from Amazon Lambda Function which is in the region [eu-west-2 (London)] to update data in DynamoDB which located in the region [eu-west-2 (London)].From the graph it is clearly evident that for a few requests there is a slight variation in the actual time taken to update data in DynamoDB to that of duration which in shown in AWS Logs.For a total of 100 Post requests it clearly shows a slight difference of (4.33 ms) in mean and a difference of (4.95 ms) in standard deviation of actual time taken to that of time shown in AWS.

Fig.8 shows the comparison of time taken for an Http Post request placed from Amazon Lambda Function which is in the region [ap-northeast-2 (Seoul)] to update data in DynamoDB which located in the region [ap-northeast-2 (Seoul)].From the graph it is clearly evident that for a few requests there is a slight variation in the actual time taken to update data in DynamoDB to that of duration which in shown in AWS Logs. For a total of 100 Post requests it clearly shows a huge difference of (48.76 ms) in mean and a difference of (21.66 ms) in standard deviation of actual time taken to that of time shown in AWS.

TABLE II : Comparison of Mean and Std Dev for 100 HTTP POST Requests

Region	Actual Mean	Billed Mean	Actual Std Dev	Billed Std Dev
ap-south-1(Mumbai)	98.08 ms	102.29 ms	38.58 ms	44.77 ms
us-east-1 (N.Virginia)	104.34 ms	108.64 ms	39.8 ms	47.03 ms
eu-west-2 (London)	84.01 ms	88.34 ms	30.58 ms	35.53 ms
ap-northeast-2 (Seoul)	85.85 ms	134.6 ms	54.01 ms	75.67 ms

REFERENCES

- [1] Ana Klimovic and Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Christos Kozyrakis
Pocket: Elastic Ephemeral Storage for Serverless Analytics. *In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. ISBN 978-1-931971-47-8
- [2] Adam Eivy : Be Wary of the Economics of “Serverless” Cloud Computing. *In IEEE CLOUD COMPUTING (MARCH/APRIL 2017)*.
- [3] Claudio Cicconetti, Marco Conti, Andrea Passarella : An Architectural Framework for Serverless Edge Computing: Design and Emulation Tools. *In 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*.
- [4] Erwin van Eyk, Lucian Toader, Sacheendra Talluri, Laurens Versluis, Alexandru Uță, Alexandru Iosup :
Serverless Is More: From PaaS to Present Cloud Computing. *www.computer.org/internet (2018)*.
- [5] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt : SAND: Towards High-Performance Serverless Computing. *In 2018 USENIX Annual Technical Conference*.
- [6] AWS Product Documentation. <https://d1.awsstatic.com/whitepapers/aws-overview.pdf>.