National College of Ireland

Lambda Authorizer Benchmarking Tool with AWS SAM and Artillery Framework

MSc Research Project
Cloud Computing

Cornelius
Student ID: 21126747

School of Computing
National College of Ireland

Supervisor: Dr. Shivani Jaswal

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Cornelius
Student ID:	21126747
Programme:	Cloud Computing
Year:	2022
Module:	MSc Research Project
Supervisor:	Dr. Shivani Jaswal
Submission Due Date:	01/02/2023
Project Title:	Lambda Authorizer Benchmarking Tool with AWS SAM and Artillery Framework
Word Count:	6061
Page Count:	20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	26th January 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Lambda Authorizer Benchmarking Tool with AWS SAM and Artillery Framework

Cornelius
21126747

Abstract

Leading cloud provider Amazon Web Services (AWS) provides a security feature called Lambda Authorizer in their serverless service, AWS Lambda. This security feature processes the security token in the request header according to the custom code set by the developers. This security technology development has led to more developers leveraging serverless technology to develop serverless APIs with public access, which previously tended to be used for private scope. This study determines the performance and cost of a serverless function that implements the Lambda Authorizer. By knowing the benchmarking results, developers can maximize the performance parameters in realizing a secure and cost-effective serverless public API. The author developed a benchmarking tool based on the AWS Serverless Application Model (SAM) and Artillery framework to measure the performance of Lambda Authorizer-implemented serverless functions with three primary performance parameters: start-up conditions, programming language runtimes, and authorization types. Using this combination of parameters, the Lambda Authorizer Benchmarking Tool shows that Python is still more performant and cost-efficient than other runtimes. It also becomes the best choice to achieve the lowest response time when combined with the request authorizer during warm conditions. The exciting result is that Go performs better if the function code requires much memory since it starts faster and has better memory management than Python.

Keywords— AWS Lambda, AWS Serverless Application Model, Artillery framework, Benchmarking tool, Lambda Authorizer

Table of Content

1 Introduction

AWS EC2 is one of the company's first Infrastructure-as-a-Service (IaaS) products. IaaS enables computer resources as a service, allowing customers to lease computer resources, easily set up server environments, and access them through the internet. In detail, IaaS offers many benefits, including a reduction in workers because the infrastructure configuration process is automated. In this way, servers and resources are operated efficiently, development and production costs are reduced, and companies are able to compete quickly in the global market.

Platform-as-a-service (PaaS) is the next step in cloud computing. AWS Elastic Beanstalk is a well-known example. PaaS is a service abstraction layer built on top of IaaS to ease complex server installations by including the Operating System (OS) into the outsourced infrastructure and enables developers to deploy software, patch software, and monitor systems. Another comparable service is Containers-as-a-Service (CaaS). CaaS is a cloud-based service architecture that hosts and orchestrates containerized workloads within the cluster. A well-known application example for CaaS is Docker, a container-based virtual machine that promoted the notion of containerization among developers all over the world (Mohammed et al.; 2021).

Serverless computing is the most advanced breakthrough in the field of cloud computing. Serverless services divide into Backend-as-a-Service (BaaS) and Function-as-a-Service (FaaS). BaaS, such as AWS Amplify, aims to replace the back-end server developers that generally set up and manage themselves. In contrast, FaaS is more analogous to a computing service that allows developers to run code granularity at the function level. One of the most outstanding examples of FaaS is AWS Lambda, the most widely used FaaS nowadays. Furthermore, the term "serverless" does not indicate the lack of a server. In essence, serverless technology frees users from the burden of server administration (Shafiei et al.; 2022). Figure 1 depicts the distinction between traditional server and serverless applications.

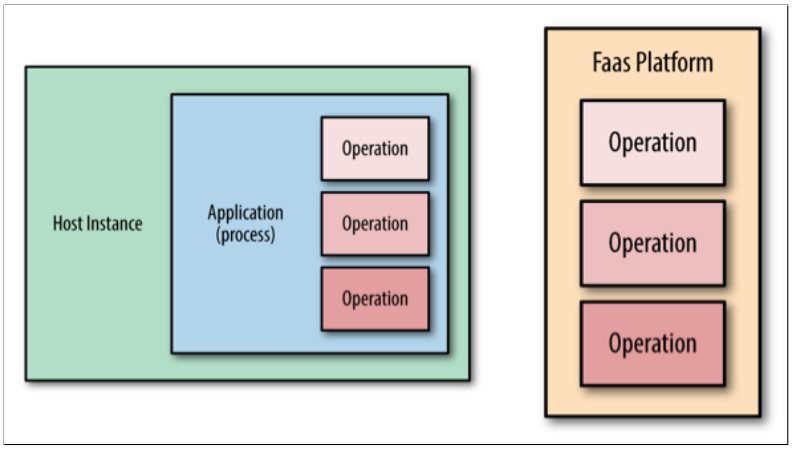

Figure 1: Traditional Server vs. FaaS Software Application (Roberts and Chapin; 2017)

The technology sector as a whole is embracing and developing serverless technologies. Serverless technology introduces a novel approach to creating and delivering applications, as well as the varying complexity of using the serverless ecosystem's components. As a result, each software company has come up with its own serverless application architectural design. Each architecture design, however, has one thing in common: the authorization and authentication procedure, which is carried out using either BaaS, Amazon Cognito, or a custom logic using Lambda Authorizer. On that premise, the author developed a benchmarking tool for serverless functions based on start-up circumstances, programming language runtimes, and authorization logic in order to improve the performance of the access-controlled AWS serverless API.

1.1 Research Question

The following research question must be answered to accomplish the thesis purpose.

How much influence do start-up conditions, programming language runtimes, and authorization types have on the cost and performance of executing the Lambda Authorizer-enabled AWS serverless API?

The above question seeks to ascertain which combination of primary performance parameters provides the optimum cost and performance for access-controlled AWS serverless API.

1.2 Motivation

In the world of cloud computing, the author believes his benchmarking tools can bring benefits in the following ways:

1. Researchers and developers can specify which performance parameters yield the most favorable cost-performance ratio when calling the Lambda Authorizer access-controlled AWS Serverless API.

2. Researchers and developers can use the Serverless Benchmarking Tool to evaluate different programming languages or access control mechanisms.

1.3 Ethics Consideration

This study does not include human subjects or private/public datasets, as per Table 1.

Table 1: Declaration of Ethics Consideration Table

This project involves human participants	Yes / **No**
The project makes use of secondary dataset(s) created by the researcher	Yes / **No**
The project makes use of public secondary dataset(s)	Yes / **No**
The project makes use of non-public secondary dataset(s)	Yes / **No**
Approval letter from non-public secondary dataset(s) owner received	Yes / **No**

1.4 Structure of Work

Following is an overview of the work's content. In Section 2, the author displays a literature overview of serverless computing, serverless security, and serverless benchmarking, as well as a table of comparative studies of key articles. Moreover, flowcharts and technical graphics are used in Section 3 to describe the research strategy and technology used to construct the Lambda Authorizer Benchmarking Tool. As shown in Section 4, the AWS SAM and Artillery Framework are used to build a high-level architectural Serverless Benchmarking Tool. Also, benchmarking of serverless function output is explained in Section 5, including configuring scenarios and how to evaluate the benchmark results. An in-depth analysis of the outcomes of each serverless function scenario is presented in Section 6. Eventually, as section 7 summarizes the study's findings and offers recommendations for future work, it discusses the study's contribution to the cloud computing community.

2 Related Work

An understanding of serverless itself is necessary for researching and evaluating their performance. The following section overviews a brief description of existing research projects. A significant part of this section explains what is needed to conduct this project and helpful information from trustworthy sources to support the author's research.

2.1 Serverless Computing

In software architecture, serverless computing is a dream come true. By using serverless technology, developers can accelerate and simplify the development process while ensuring scalability, security, and performance are handled. The concept of serverless is widely misunderstood by most people, who believe it does not require a server. It is essential to understand that serverless still uses servers. It is only that the internal infrastructure is mainly concealed from the consumer. A serverless approach saves developers time by focusing on function development rather than configuring infrastructure (Sbarski and Kroonenburg; 2017, Chapter 1).

2.1.1 Serverless Services

Functional components can be separated granularly with the help of serverless technology. They use as few resources and costs as possible, owing to their advantages. For instance, when deploying functions as serverless services, developers do not need to pay for the creation and setup of the functions. The pricing system differs in serverless services, where developers have to pay based on the memory usage and the runtime duration of used functions. A single-purpose APIs and web services can be leveraged to easily create serverless APIs, which allow developers to construct loosely-linked, scalable, and efficient structures. Ultimately, developers should concentrate on creating code rather than worrying about how the server infrastructure works when using serverless (Sbarski and Kroonenburg; 2017, Chapter 3). Serverless services apply to a wide range of business applications, including enterprise applications, mobile applications, data processing, scheduling, and embedded applications. This technology fuses the processes of containerization and virtualization into a unified architecture. Even though serverless is still considered new to developers, they can choose the best strategy to implement the system into a serverless by understanding the various technologies that work around it (Rajan; 2018).

2.1.2 Serverless Cloud Platforms

Three of the most popular serverless cloud providers are AWS, Microsoft Azure, and Google Cloud Programming (GCP). According to their findings, AWS is the platform with the best performance across all testing situations, whereas Microsoft Azure has the most significant variance in working behavior and GCP has unpredictable performance (Copik et al.; 2021, Section 8). To objectively evaluate the performance of reference applications on cloud platforms, the workloads generated from the benchmark experiments must be realistic and use the same workload model and data volume as the input data (Deng; 2022, Section 3). However, the author does not have to create a fair workload because the focus of the research in this work is benchmarking serverless function that implemented Lambda Authorizer. This feature is only available in the AWS environment. In addition, a fully secured serverless application can be created with the help of AWS Lambda, which makes sense for enterprises already committed to using AWS as their public cloud provider. In order to allow fast access to internal resources, AWS integration is necessary, which includes storage, databases, and streaming. It automatically scales the bandwidth and processing power needed for each function based on the amount of memory specified by the developers (Patterson; 2019, Chapter 4).

2.1.3 Serverless Application Builder

AWS provides a unique AWS SAM framework to manage serverless resources programmatically or through Command-Line Interface (CLI). This framework can provision and perform serverless operations with multiple programming runtimes in AWS Lambda. AWS SAM is built on top of AWS CloudFormation as an extension, primarily used to develop structural components of AWS services because of their outstanding performance when used in the AWS environment. AWS CloudFormation, on the other hand, has a level of difficulty in learning and understanding it. Thus AWS SAM is built with a script syntax that is more human-readable and simpler to understand. The configuration files adopt popular formats such as JSON and YAML, which have various vital properties in serverless management, ranging from global, security, and event sources. Since the implementation of AWS SAM is open-source, the community may contribute to improving its functionality, and the degree of popularity of AWS SAM has continued to rise to compete with competing for serverless frameworks (Grumuldis; 2019).

2.1.4 Serverless Performance Tester

The Artillery and JMeter framework is one of the most versatile and famous testing frameworks. They both can generate the function performance result test offline. However, JMeter is not specifically designed to calculate the performance in serverless environments (Abbas et al.; 2017). On the other hand, the Artillery framework is built purposely for testing serverless functions. A similarity can be seen between the Artillery and JMeter framework in terms of what it aims to achieve. Libraries in this module measure serverless test data scenario performance. The maximum number of concurrent tests the Artillery framework can run depends on its cloud environment computing capabilities and network. The Lambda functions are executed using the Artillery package, which can increase throughput for developers. Due to this, the application can handle a more significant number of tests simultaneously. Additionally, Artillery allows developers to deploy custom methods for delivering header data and query string parameters in cases where the serverless API requires them (Andell; 2020).

2.2 Serverless Security

Consider what would happen if unauthorized users gained access to the AWS environment. Inadvertently publishing AWS credentials to the public GitHub project is one of the most prevalent examples. Hackers can quickly access the developer's AWS resources by searching the public repository. Events such as these may interrupt the serverless function working process since service constraints apply at the region level. All available throughput in the area can be exhausted by one team or service, and this causes all other processes to slow down. Every change made in a non-production environment may affect users in production since all environments share the same AWS account. This subsection explains a few examples of how developers can restrict access to AWS Lambda serverless APIs (Sbarski and Kroonenburg; 2017, Chapter 11).

2.2.1 Serverless Authorization Types

AWS Lambda has many security features, from IAM Permissions and Amazon Cognito to Custom Authorizer. The below part explains more about the custom authorizer because it is directly related to the main scenario of this research project. Moreover, AWS Lambda has changed the name of the Custom Authorizer feature to Lambda Authorizer. As of the most recent update, Lamba Authorizer allows the developers to create custom access controls for serverless APIs. API caller identity is determined using bearer token schemes like OAuth or SAML or request parameters from Lambda Authorizer. Following is an explanation of the two types of Lambda Authorizer and how they work:

1. **A TOKEN authorizer** identifies callers using bearer tokens, such as JSON Web Tokens (JWT) and OAuth tokens. Bearer Tokens are not meant to have any meaning for users. A typical implementation of token authorizer is GitHub's authorization process.
2. **A REQUEST authorizer** analyzes headers, query string arguments, state variables, and context variables to determine the caller's identity. Additionally, WebSocket APIs use request parameters and do not support other types of authorization.

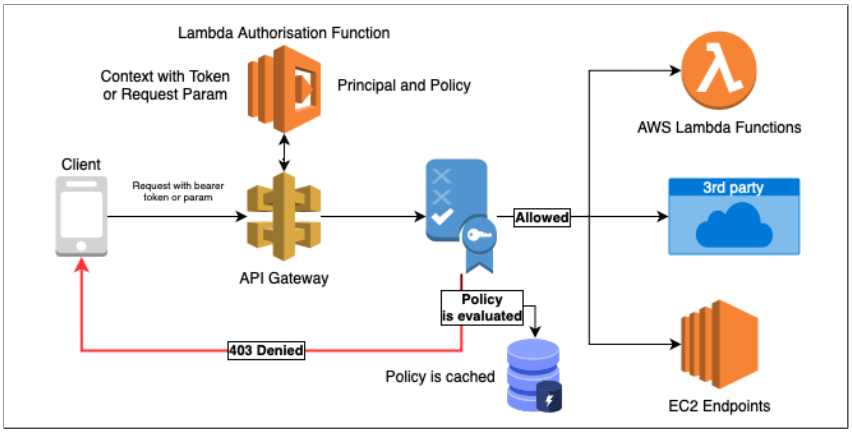

Figure 2: Lambda Authorizer Workflow (*Use API Gateway Lambda authorizers*; 2022)

Lambda Authorizer performs the authentication process between the client and serverless resource, as illustrated in Figure 2. First, clients request API Gateway APIs using bearer tokens or request parameters. In order to determine whether a Lambda Authorizer is used, the API Gateway checks the method configuration and determines whether the Lambda function can be called or not. It authenticates the caller by obtaining an access token from an OAuth provider or a SAML assertion from a SAML provider. Based on the value in the request parameter, the system creates an IAM policy to obtain login information. If the checking process is successful, the Lambda function sends an object containing the IAM policy and principal identity. If access is denied, the API gateway evaluates the policy and sends HTTP status codes such as 403 ACCESS DENIED. As soon as access is accepted, the API Gateway executes the method. Additionally, the API Gateway caches policy data, so Lambda Authorizer does not have to be called multiple times when the caching system is enabled (Calles; 2020, Chapter 4).

2.3 Serverless Benchmarking

By extracting research on serverless benchmarking tools, the author found a number of existing projects on serverless benchmarking tools, such as the Serverless Benchmarking Suite that used HyperFlow engine to support many clouds (Malawski et al.; 2017), the Serverless Performance Framework, which was built from the Serverless Framework (Jackson and Clynch; 2018), the Serverless Application Analytics Framework, which was used to measure the performance of serverless data processing pipelines (Cordingly et al.; 2020), and the PanOpticon benchmarking tool to support custom serverless functions with Python runtimes in AWS and GCP environment (Somu et al.; 2020). This includes the most comprehensive benchmarking tool, Serverless Benchmark Suite, which was used to assess the performance of serverless data processing pipelines. It also supports several serverless cloud platforms (Copik et al.; 2021), as well as the latest research titled Serverless Benchmarkers that uses fair workloads to test the performance of different serverless cloud platforms (Deng; 2022). This subsection will explain the factors that influence a serverless function's performance.

2.3.1 Start-up Conditions

When an initialization start occurs, the latency of the requests is also called cold latency, and the actual latency of the serverless function execution process is called warm latency. There are only two situations in which the cold start penalty occurs precisely. The first situation occurs when AWS Lambda must create new environments to handle the influx of incoming requests. The second is calling a serverless function that has not been called for a long time. Generally, cold starts make up fewer than 0.5 percent of incoming requests for serverless functions, but infrequently functions' traffic spikes are not affected as much by cold starts. Moreover, AWS Lambda's timeout setting applies to the entire request delay, so requests with cold start-ups may also experience timeouts (Sbarski and Kroonenburg; 2017, Chapter 10).

2.3.2 Programming Language Runtimes

Given the cost transparency provided by serverless pricing models, understanding the cost impact of a function's language is critical. For example, C# programming language has a modest advantage in minimizing the number of cold starts because of the maturity of support for distributed tracing in Microsoft Azure Functions. From what can be seen in Figure 3, calling a serverless function is not straightforward. It must go through several layers, such as the internal compute substrate, execution environment, and language runtime. The execution environment and the function code are instantiated on demand for each request. When the first request arrives, AWS Lambda runs the code within the function handler after the environment is formed. As soon as the handler logic is complete, AWS Lambda considers the function complete. AWS Lambda does not destroy the execution environment; instead, it saves it. When a subsequent request happens, and a cached execution environment is available, AWS Lambda handles the request using that execution environment. AWS Lambda will construct the new execution environment if a cached execution environment is unavailable.

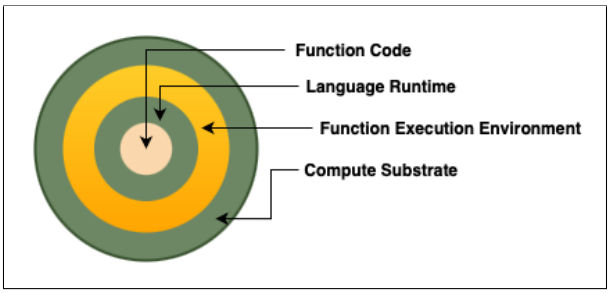

Figure 3: Serverless Function Layers Execution (Sbarski and Kroonenburg; 2017)

2.3.3 Serverless Underlyings

Many limitations must be overcome for serverless technology to be effective, including one arising from the dark side of the cloud platform. When developing non-serverless applications, the developer has total control over the software and hardware stack. However, in serverless application development, almost all hardware and software aspects become opaque and invisible from the developer's point of view. Because of this, serverless environments are less flexible than traditional ones. The following are some of the missing aspects in a serverless environment, the absence of total power and hardware underlying management, including the least control for complex security features. These limitations can sometimes cause benchmark testing results to be inconclusive and different from time to time due to many unexpected and mysterious things from the developer's perspective (Shahrad et al.; 2019) (Kelly et al.; 2020).

2.4 Comparative Analysis of Related Works

In Table 2, related works have been compared based on their framework, scenario, advantages, and limitations.

Table 2: Comparison of Related Works Table

Reference	Framework	Scenario	Advantages	Limitation
This Research	Lambda Authorizer Benchmarking Tool (LABT) constructed using AWS SAM and Artillery Framework.	A serverless function that implements Custom Lambda Authorizer.	Analyse the performance of a serverless function with access control.	
(Deng; 2022)	Serverless Benchmarker (SB).	Thumbnail generator, model training, and video processing.	Fair benchmark comparison to broad cloud serverless provider.	Mitigating construct validity depends on cloud platform features.
(Copik et al.; 2021)	Serverless Benchmark Suite (SeBS).	General serverless application.	Support multiple cloud platforms.	Support only specific serverless scenarios.
(Cordingly et al.; 2020)	Serverless Application Analytics Framework (SAAF).	Transform-Load-Query serverless application.	Support Data Processing Pipeline Scenario.	Data Storage is limited to S3.
(Somu et al.; 2020)	PanOpticon (PO) uses Serverless Framework and JMeter.	Custom simple function.	Easy configuration using a dedicated configuration file.	Python runtimes are the only ones supported.
(Jackson and Clynch; 2018)	Serverless Performance Framework (SPF) made using Serverless Framework.	Empty function.	As benchmarking runs without third parties involved, they produce accurate results.	The empty function scenario is plain compared to the actual serverless application.
(Back and Andrikopoulos; 2018)	Microbenchmark (MB) built on top Serverless Framework.	Basic function.	Wide range of cloud provider support.	Limited input and output performance parameters.
(Malawski et al.; 2017)	Serverless Benchmarking Suite (SBS) produced using Serverless and HyperFlow Engine.	Custom simple function.	Multi-cloud provider support.	Output parameters are only CPU and RAM usage.

3 Research Methodology

The use of appropriate methodologies is required when benchmarking serverless functions. With the correct approach, the configuration process will be easier to manage, and the outputs will be more accurate. This section presents a general overview of the benchmarking process for an access-controlled serverless function, from input until it generates the expected outcomes.

3.1 Process Overflow

Figure 4 shows the flow of benchmarking tool process in calculating performance for multiple secured serverless functions. Users must input a specific command to trigger the benchmarking process. In short, the benchmarking process undergoes several stages and validation to produce the desired output. Below is a more detailed description of each flowchart component:

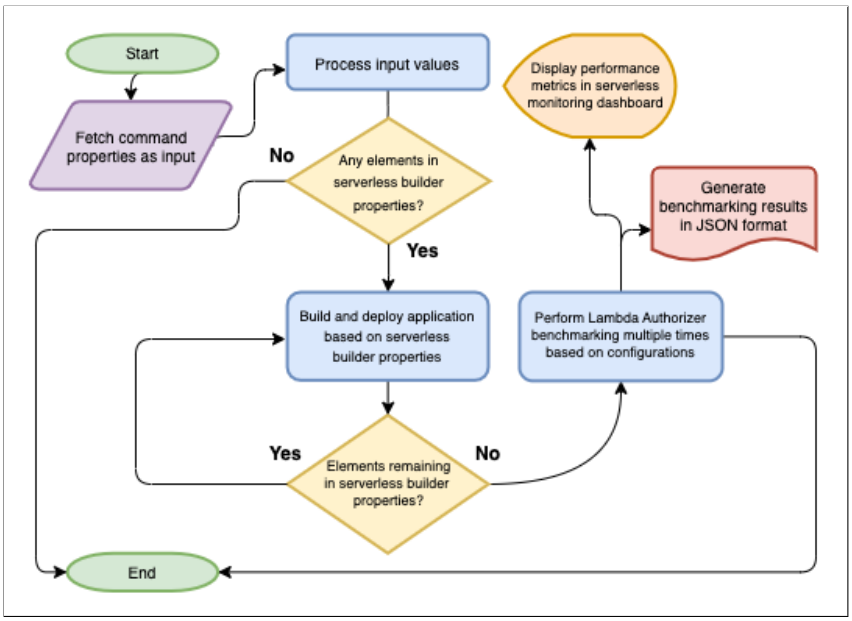

Figure 4: Process flowchart for General Serverless Benchmarking Tool

Using the benchmarking tool command line, users can start the serverless application builder and calculate serverless functions' performance simultaneously. Afterward, the system receives the user's input and processes the existing application builder properties. Keep in mind that the application builder properties must have a value. Otherwise, the system will stop. A Lambda Authorizer function is built and deployed based on the properties and linked with the desired function by the serverless application builder framework. Before moving on to the performance testing stage, a final check will be performed if any application building properties are still unexecuted. The following property will be run if one is found.

Subsequently, the system will perform serverless performance testing based on configuration files that the user already sets up. This process then determines which scenarios users wish to run and how many testing iterations are needed based on input in the beginning. As a result of benchmarking, the serverless monitoring dashboard displays each tested function's performance metrics. Also, users can retrieve benchmarking results in JSON format in the local output folder. Additionally, the users can repeat the testing process by providing different command parameters and settings for the serverless application builder to analyze different scenarios.

4 Design Specification

Cutting-edge technologies and techniques must be employed to develop a reliable and robust benchmarking tool for serverless computing. Accordingly, the author will briefly describe the high-level process for the Lambda Authorizer Benchmarking Tool using a sequence diagram. This section explains the performance metrics that the serverless benchmarking tool can gather.

4.1 Diagram

Sequence diagrams are used in this section to demonstrate the interaction between objects sequentially. The diagram is intended exclusively for developers. It has also been found that these diagrams can also be used to facilitate communication between technical and business department employees in some companies.

4.1.1 Sequence Diagram

Figure 5 illustrates the workflow of the Lambda Authorizer benchmarking process. This diagram was added by the author as a quick way to explain how the application's internal system works.

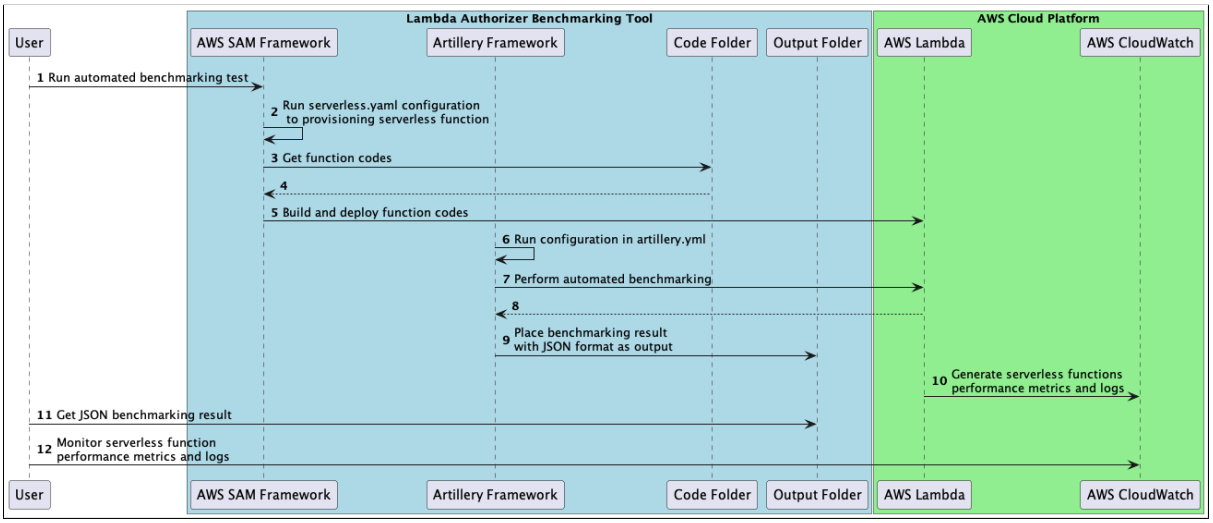

Figure 5: Lambda Authorizer Benchmarking Tool Sequence Diagram

The user instructs the Lambda Authorizer Benchmarking Tool to run an automated serverless benchmarking test. All serverless functions specified in the template.yaml are prepared to be pushed to the AWS Lambda service using the AWS SAM framework. AWS SAM reads the authorizer and application function code within the source folder and continues building the code using the configured runtime. If there is no compilation error, the system will deploy it. The system runs the performance test scenario by checking artillery.yml as a based configuration. It will generate multiple virtual users to call the request URL simultaneously for a particular duration. Through AWS Lambda, the Artillery framework converts the performance outputs into human-readable content in JSON format, which will be stored in the local folder. The AWS CloudWatch service monitors and writes logs of serverless function activities at the same time. Using the AWS CloudWatch dashboard, the users can visit a graphical representation of the performance activity by default. In addition, this service also provides users with remarkable capabilities to analyze logs interactively directly from within written logs using AWS CloudWatch Logs Insights (*Analyzing log data with CloudWatch Logs Insights*; 2022).

Furthermore, there are three main components in the Lambda Authorizer Benchmarking Tool, namely AWS SAM and Artillery Framework, also AWS CloudWatch as monitoring services. The author chooses AWS SAM for the serverless application builder framework rather than popular frameworks like Serverless Framework because the benchmark scenario in this research work is to measure the performance of serverless functions that implements Lambda Authorizer. Since Amazon SAM is a native framework, it performs much better than rival frameworks. Meanwhile, the Artillery framework is used as a serverless performance tester framework because it is purposely engineered to measure the performance of scalable cloud functions (Ritzal; 2020). In contrast, popular performance testing frameworks like JMeter exist for functions in general. As a serverless monitoring system, AWS CloudWatch is chosen since it is suitable for this application. This service directly supports logging and monitoring activities natively within AWS as the cloud provider being tested. Another thing that needs to be added is that the author chooses the output results in JSON format because it has high flexibility and can support various processes regardless of programming language.

4.2 Performance Metrics

The results of different scenarios in benchmarking will be presented in a Table 3 structure. Based on four programming runtimes and two authorizer types, the performance output parameters recorded are the maximum duration for cold and warm starts, maximum memory use, response time, and performance cost. Using this table, the author will determine which programming runtime has won the most categories as the overall winner.

Table 3: Serverless Benchmarking Tool Performance Metrics Template

Runtime	Authorizer Type	Max Init Duration	Max Duration	Max Memory Used	Avg Response Time	Max Cost
Python 3.9	Request	?	?	?	?	?
Python 3.9	Token	?	?	?	?	?
Node 16.x	Request	?	?	?	?	?
Node 16.x	Token	?	?	?	?	?
Go 1.x	Request	?	?	?	?	?
Go 1.x	Token	?	?	?	?	?
Java 11	Request	?	?	?	?	?
Java 11	Token	?	?	?	?	?

In order to determine the performance cost, the author has to manually calculate it. With AWS Lambda, the service price is calculated based on the number of CPU cycles rather than the number of bytes (Farley; 2021, Chapter 2). The service counts all types of requests and their duration, whether they come from event managers, Amazon API Gateway, or AWS console directly (Ibrahimi; 2017). During the calculation, the time from the start of the function code to its end is considered. Minor duration rounding begins at 1 ms. Moreover, serverless functions are also priced based on the memory resources allocated to them. Due to memory allocation, function durations will be influenced, which affects cost as well. Memory can be allocated to functions between 128 MB and 10,240 MB in 1 MB increments. By default, each function has a timeout of 3 seconds, 128 MB of memory and 512 MB ephemeral storage. Table 4 gives a picture of the cost calculation of a serverless function using 1024 MB of memory.

Table 4: AWS Lambda Pricing Table (*AWS Lambda Pricing*; 2022)

Architecture	Duration	Requests
x86	$0.0000166667 for every GB-second	$0.20 per 1M requests
Arm	$0.0000133334 for every GB-second	$0.20 per 1M requests

5 Implementation

The implementation of the proposed design is presented in this section. Besides explaining the high-level architecture of the application and the type of scenarios are run, the author also describes how the performance cost calculations are made. Interested readers can find the source code for Lambda Authorizer Benchmarking Tool on **GitHub**.

5.1 Architecture Overview

Figure 6 displays the high-level architecture of the Lambda Authorizer Benchmarking Tool. This application is developed based on the sequence diagram in Figure 5. AWS SAM is used for removing existing function stack, building, and deploying function code according to template scripts. Furthermore, the application also utilizes the Artillery framework to run performance test scenarios with query string parameters for request authorization types and authorization headers for token authorization types. In addition, using the basic configuration provided by this benchmarking tool, the user can adjust the duration and rate of function calls. Performance testing results are split into two parts, those obtained from the Artillery framework and those queried directly from AWS CloudWatch Logs Insights. Performance results generated by Artillery have important outputs such as request rate and response time. This output has a JSON format that can be converted into HTML with a more human-friendly visual. Meanwhile, the output provided by AWS CloudWatch Log Insight only has a JSON format but has a complete output, such as cold and warm duration, as well as memory usage.

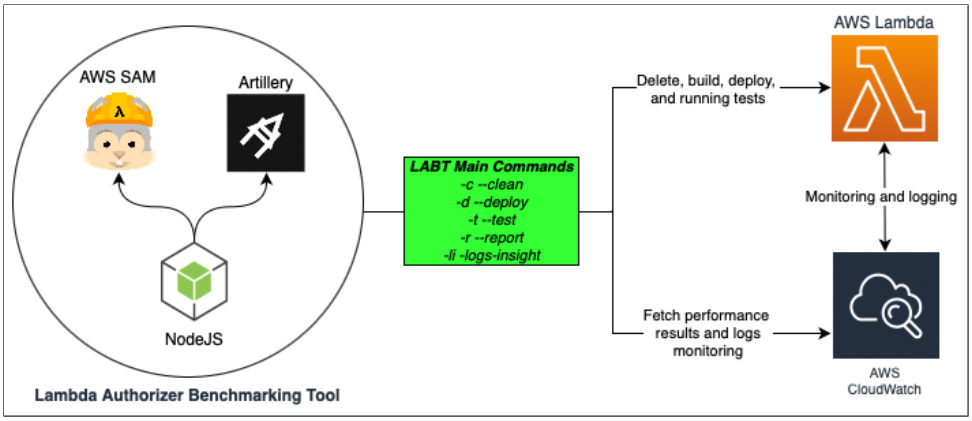

Figure 6: High-Level Architecture of Lambda Authorizer Benchmarking Tool

5.2 Performance Test Scenarios

The author runs and evaluates a total of eight benchmarking scenarios, a combination of four programming language runtimes and two authorization types. The programming language runtimes used are Python 3.9, NodeJS 16.x, Go 1.x and Java 11. According to AWS Lambda's documentation, Python and NodeJS have superior overall performance, while Go has a quick start, while Java has a long start but fast after initialization (*AWS Lambda Runtimes and performance*; 2022). Based on these documents, the authors chose the four programming runtimes as the subject tests in this research. The default specification for all tested serverless functions is x86 architecture, 128 MB memory, 512 MB ephemeral storage, and no cache. Further, each function is tested for ten seconds, which is comprised of 10 different virtual users invoking each function once every second, resulting in a total of 100 invocations.

5.3 Logs Insights Custom Queries

The Lambda Authorizer Benchmarking Tool's Logs Insights feature allows this application to generate seven pre-defined query results consisting of an overview query with a combination of two authorizer types and three main parameters (maximum initialization time, maximum duration, and maximum memory used). All these results are displayed in JSON format. As additional information, if the user wants to get other custom query output, the user can get the results of the query results manually, as shown in Figure 7.

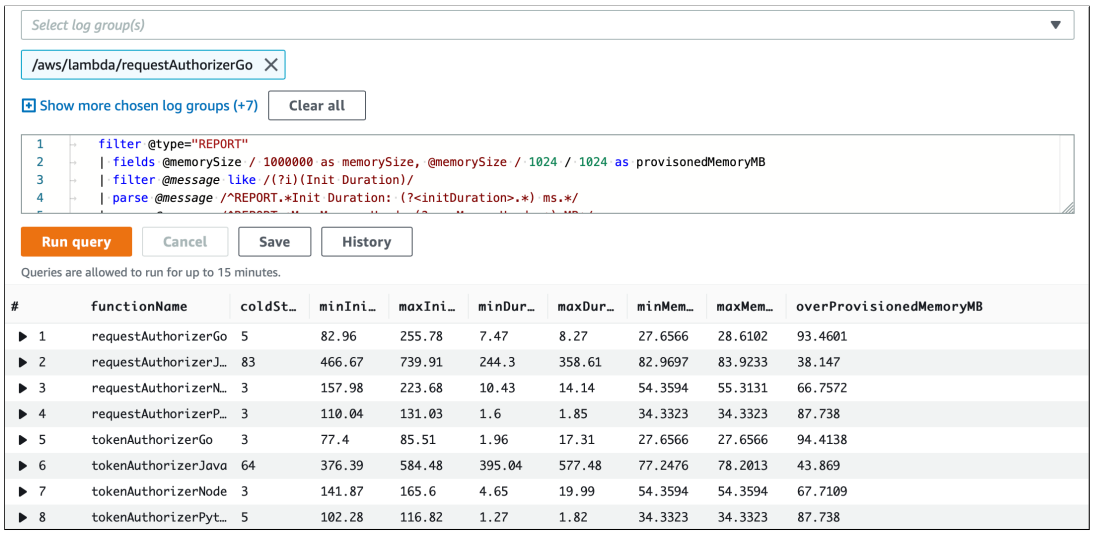

Figure 7: Logs Insight Manual Query for Overview Output

5.4 Performance Cost Calculation

The author calculates the function cost manually with the help of the AWS Calculator tool. This tool can estimate the price to be paid each month based on the service used and the resource parameters entered. For example, a function with three times cold starts of 1 million calls. The maximum cold start duration is 131.03 ms, which happened three times, while the maximum warm start duration is 1.85 ms. Then with balance calculation, the average maximum duration of a function is 5.7254 ms. It is necessary to round up the result to 6 milliseconds because AWS Lambda does not record duration in decimal form. Next, the result must be input into the AWS Calculator so that the below results will be displayed:

```
# Function duration 6 ms and 1,000,000 requests/month
Memory allocated: 128 MB x 0.0009765625 GB in a MB = 0.125 GB
Ephemeral storage allocated: 512 MB x 0.0009765625 GB in a MB = 0.5 GB

# Pricing calculations
1,000,000 requests x 6 ms x 0.001 ms to sec = 6,000.00 total compute (seconds)
0.125 GB x 6,000.00 seconds = 750.00 total compute (GB/s)
Tiered price for: 750.00 GB/s
750 GB/s x $0.0000166667 = $0.01
Total tier cost = $0.0125 (monthly compute charges)
1,000,000 requests x $0.0000002 = $0.20 (monthly request charges)
Billable ephemeral storage = 0.50 GB - 0.5 GB = 0.00 GB (no charge)
Lambda costs - Without Free Tier (monthly): $0.0125 + $0.20 = $0.21
```

6 Evaluation

This section is filled with five summary bar charts for each category of main output parameters in the section Performance Metrics. Subsequently, the author discusses the advantages and disadvantages of each programming language runtime based on benchmarking results.

6.1 Maximum Initialization Duration Query Result

Figure 8, the first bar chart in this section, shows the benchmarking results in the form of the maximum duration of initialization of a function at each runtime. This duration is known as cold latency. Cold latency usually occurs when a new function is created or one that has not been used for a long time starts to be activated. Typically, with the default settings including cache-enabled, cold starts only occur in less than 0.25 percent of the total requests.

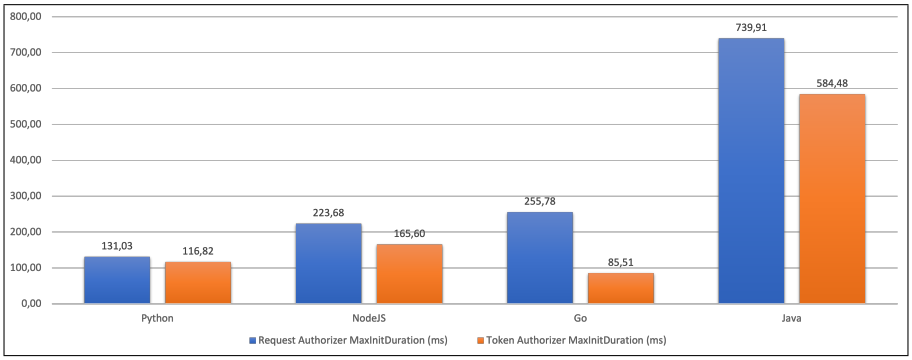

Figure 8: Authorizer Maximum Initialization Duration Bar Chart

6.2 Maximum Duration Query Result

The maximum duration bar chart illustrated in Figure 9 shows the speed in processing the Lambda Authorizer function with the Request and Token types. This maximum duration is also called warm duration, which means that the function is active and ready to wait for incoming requests. The author observes and analyzes a discrepancy between Java and other runtimes regarding the maximum time required. This lengthy process duration was observed only in 1 to 5 percent of total requests, according to other output results the author obtained.

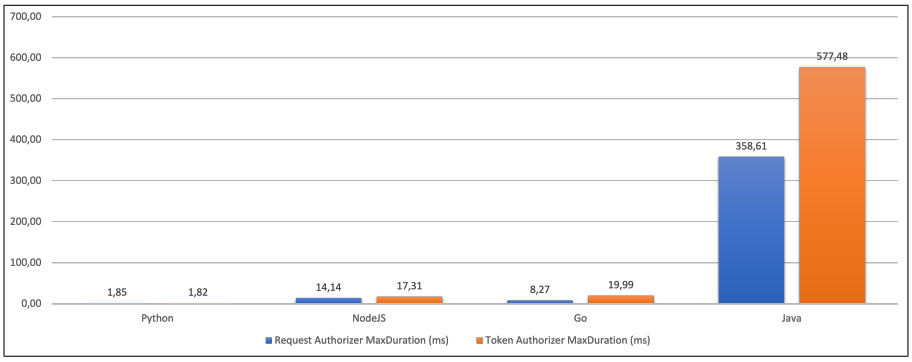

Figure 9: Authorizer Maximum Duration Bar Chart

6.3 Maximum Memory Used Query Result

The results of querying the efficiency of runtime in memory management are shown in the bar chart in Figure 10. Despite implementing Lambda Authorizer with the lowest memory specification (128 MB), there is still a lot of memory left over. So there should be no more excuses if the function that implements the access control feature brokes the amount of memory limit used. Interestingly, when comparing the memory efficiency of the Go runtime with that of the Java runtime, the gap is quite large, more than two times as large.

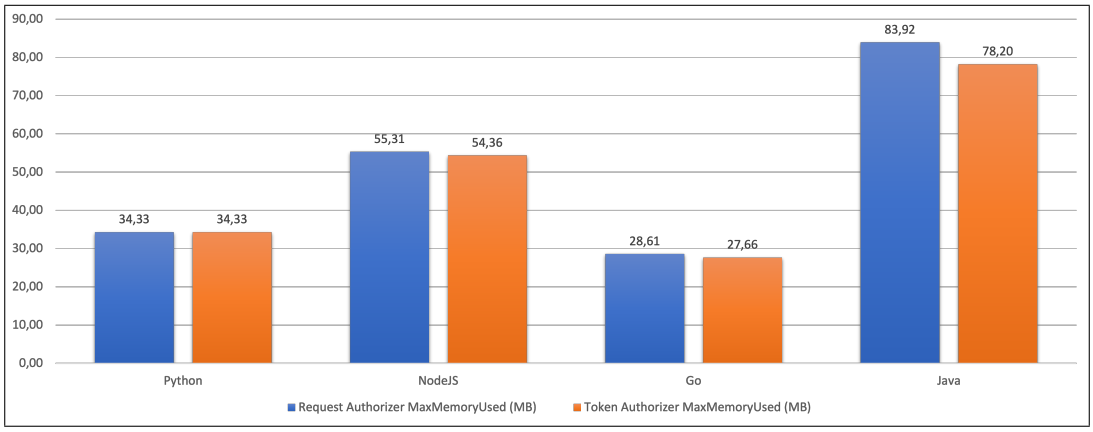

Figure 10: Authorizer Maximum Memory Used Bar Chart

6.4 Response Time Calculation Result

There is a difference between response time and duration. Duration means the period of time required to initiate and execute Lambda Authorizer functions. Meanwhile, this output parameter is an overall time calculation starting from when the user requests the function, enters the Lambda Authorizer function, steps further to the actual function, and returns it as a response to the user. The Artillery framework generates the data displayed in Figure 11. The bar chart shows here by average response time after 100 invocations. It is interesting to note that the Java runtime actually has relatively good performance. Because after the averaging process, in token authorizer scenarios, the results are the same as the performance of the NodeJS runtime.

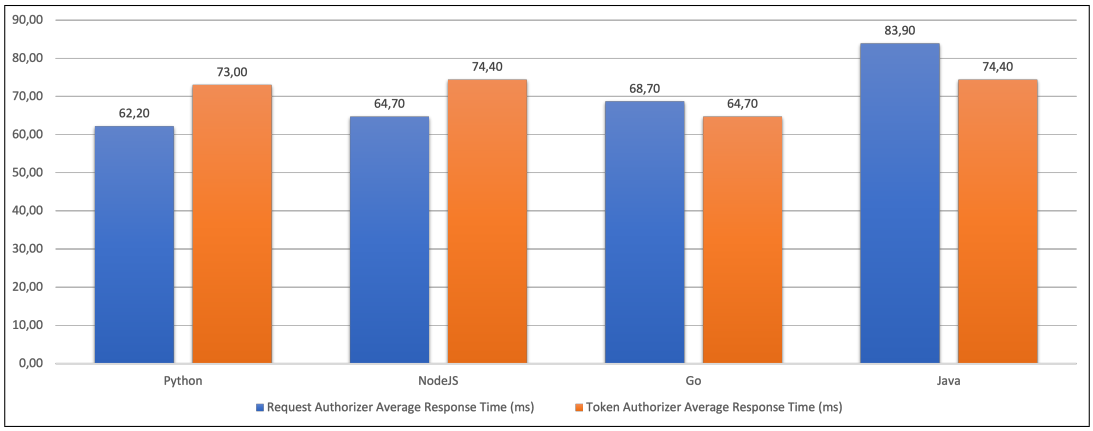

Figure 11: Authorizer Average Response Time Bar Chart

6.5 Performance Cost Calculation Result

Finding the best performance cost without reducing the security of a function is one of the objectives of this research work. The performance cost of a scenario is generally affected by the speed at which functions are executed and by the amount of memory they consume. The value in Figure 12 is calculated manually based on the maximum init and function duration according to the proportion of cold and warm conditions. The results of these calculations are fed into forecasting calculations for AWS Lambda in AWS Calculator. This chart shows that Python is the runtime with the best cost performance. However, according to Figure 10, Go runs more efficiently in terms of memory usage. When running scenarios that drain memory capacity, Go will have the upper hand in the battle for best performance cost.

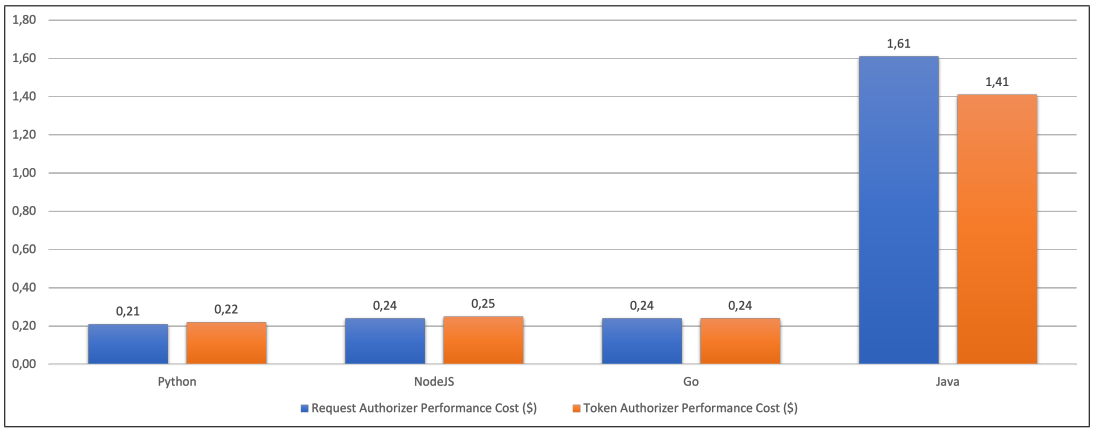

Figure 12: Authorizer Performance Cost Bar Chart

6.6 Discussion

Table 5 is a copy of the performance matrix of Table 3, filled with the benchmarking results from previous subsections. The author gives a blue highlight to the columns with the best results from each category and a red highlight which means the worst result in each category. The programming language runtime will be sorted based on how many highlights it gets. Highlights in blue score plus 1, highlights in red score minus 1, and columns without highlights score 0 points. After calculation, here is the order of the best programming language runtimes: Python 3.9, Go 1.x, NodeJS 16.x, and Java 11. Moreover, rows with the most blues represent the best scenario. There are two scenarios. First is a combination of Python runtime with a request authorizer in warm conditions. The other one combines Go runtime with a token authorizer in cold conditions. However, the cold duration is only a tiny part of the total request, and the Go's warm duration in the second scenario is ten times slower than in the first scenario. So the author decides that the first result is taken as the best scenario.

Table 5: Serverless Benchmarking Tool Performance Metrics Result

Runtime	Authorizer Type	Max Init Duration	Max Duration	Max Memory Used	Avg Response Time	Max Cost
Python 3.9	Request	131.03 ms	1.85 ms	34.33 MB	62.2 ms	$0.21
Python 3.9	Token	116.82 ms	1.82 ms	34.33 MB	73 ms	$0.22
Node 16.x	Request	223.68 ms	14.14 ms	55.31 MB	64.7 ms	$0.24
Node 16.x	Token	165.6 ms	17.31 ms	54.36 MB	74.4 ms	$0.25
Go 1.x	Request	255.78 ms	8.27 ms	28.61 MB	68.7 ms	$0.24
Go 1.x	Token	85.51 ms	19.99 ms	27.66 MB	64.7 ms	$0.24
Java 11	Request	739.91 ms	358.61 ms	83.92 MB	83.9 ms	$1.61
Java 11	Token	584.48 ms	577.48 ms	78.20 MB	74.4 ms	$1.41

Python is the overall winner of this research's benchmarking tests. This runtime gets ranked one for function duration, response time, and performance cost. Based on the author's observation, the Python runtime is swift because cold conditions only occur three times out of 100 invocations, apart from a fast start-up time. Apart from that, what is impressive is that the duration of running the function in warm conditions is at least four times faster than other runtimes. Then with everything fast, performance costs can be reduced, so this runtime combined with a request authorizer is the best choice if developers aim to create serverless applications with a limited budget but strong performance.

Go ranks second in this benchmarking test research. This runtime attracts the writer's attention because, according to the benchmark output, this runtime can beat Python in terms of cold start and memory efficiency. The difference is quite plenty. The cold start speed on the token authorizer scenario is approximately 30 milliseconds or 27 percent faster than Python. Meanwhile, one of the most fantastic things about this runtime is its memory management, which only requires 20 percent less memory than its closest competitor, Python again. For this reason, the authors suggest that if developers want to develop responsive or real-time serverless applications with the best memory usage, then use Go.

NodeJS is firmly in third place. The performance of the runtime individual component variables is entirely satisfactory. It is just that the average response time in the token authorizer scenario has the same results as the Java runtime, which is 74 ms, which is weird. Because the scope of this research is limited, this anomaly needs to be investigated further in other research. Developers can still use this runtime if they are familiar with it. According to the author, this runtime is the most developer-friendly, easy to develop, has no constraints, and has many communities that provide open-source libraries.

Java is located at the bottom of the benchmarking test in this research. All red highlights are located in this runtime, especially in the request authorizer scenario. This runtime has the most prolonged start-up duration, lengthy function execution, highest memory usage, high response time, and a more expensive function per request cost. However, if the readers look more closely at the response time section, the results obtained in the token authorizer scenario are only 13 percent slower than rank 1, Go. That means the maximum function duration hit at the Java runtime is rare and mainly has a much lower duration. Java is the programming language of millions of people and has proven reliable and secure on various production systems. For this reason, the authors understand that there are developers who want to use Java in developing serverless applications. The author recommends enabling caches to minimize dire performance activities and high costs for a serverless API created with Java runtime.

7 Conclusion and Future Work

In conclusion, as seen from the evaluation results, Python runtime is the best option for the lowest response time scenario in combination with the request authorizer type and warm condition. The Python runtime is very powerful also economical, especially when running functions during warm condition. Compared to its closest competitor, Go, this runtime is four times faster. However, this achievement could have been flawless. Go, which ranks second, is very close to Python. It also has advantages over Python, such as a 27 percent quicker cold start-up time and 20 percent more efficient memory usage. Go may have the upper hand if the scenario has higher memory usage. In third place is the NodeJS runtime, which performs well and is stable in individual component testing. However, when viewed from the authorizer token scenario average response time, the results obtained are the same as Java results. The last position is the Java runtime, which has the most unsatisfactory results in almost every benchmarking scenario. Despite this, Java's overall average response time on token authorizer is still on par with NodeJS in some scenarios and only 13 percent slower than the first-place Go.

With the creation of the Lambda Authorizer Benchmarking Tool, understanding the influence between start-up conditions, programming language runtimes, and authorization types in Lambda Authorizer-enabled serverless function or API becomes more accessible and accurate. However, this application still needs improvement, and many parts can be improved. One of them is the addition of command options so that users can easily set up a new language runtime or other access control features in the AWS SAM builder properties. Then a higher level of complexity can also be added to benchmarking scenarios, such as scenarios accessed via databases, message queues, and event managers, along with a greater variety of output formats and parameters. This application can be commercialized with more diverse features, such as CI/CD integration, multiple accounts access, a friendly user interface, web accessibility, and a proper support system. It is also possible to develop this benchmarking tool for other cloud platforms. The only requirement is to choose a framework that supports both automated serverless application building and cross-cloud platforms, such as the Serverless, Spring Cloud Function, or Terraform. Alternatively, there is also an option of developing natively for each cloud provider.

References

Abbas, R., Sultan, Z. and Bhatti, S. N. (2017). Comparative analysis of automated load testing tools: Apache jmeter, microsoft visual studio (tfs), loadrunner, siege, *2017 international conference on communication technologies (comtech)*, IEEE, pp. 39–44.

Analyzing log data with CloudWatch Logs Insights (2022).
URL: *https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/AnalyzingLogData.html*

Andell, O. (2020). Architectural implications of serverless and function-as-a-service.

AWS Lambda Pricing (2022).
URL: *https://aws.amazon.com/lambda/pricing/*

AWS Lambda Runtimes and performance (2022).
URL: *https://docs.aws.amazon.com/lambda/latest/operatorguide/runtimes-performance.html*

Back, T. and Andrikopoulos, V. (2018). Using a microbenchmark to compare function as a service solutions, *European Conference on Service-Oriented and Cloud Computing*, Springer, pp. 146–160.

Calles, M. A. (2020). Authentication and authorization, *Serverless Security*, Springer, pp. 229–256.

Copik, M., Kwasniewski, G., Besta, M., Podstawski, M. and Hoefler, T. (2021). Sebs: A serverless benchmark suite for function-as-a-service computing, *Proceedings of the 22nd International Middleware Conference*, pp. 64–78.

Cordingly, R., Yu, H., Hoang, V., Perez, D., Foster, D., Sadeghi, Z., Hatchett, R. and Lloyd, W. J. (2020). Implications of programming language selection for serverless data processing pipelines, *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, IEEE, pp. 704–711.

Deng, R. (2022). Benchmarking of serverless application performance across cloud providers: An in-depth understanding of reasons for differences.

Farley, D. (2021). *Modern Software Engineering: Doing What Works to Build Better Software Faster*, Addison-Wesley Professional.

Grumuldis, A. (2019). Evaluation of "serverless" application programming model: How and when to start serverles.

Ibrahimi, A. (2017). Cloud computing: Pricing model, *International Journal of Advanced Computer Science and Applications* **8**(6).

Jackson, D. and Clynch, G. (2018). An investigation of the impact of language runtime on the performance and cost of serverless functions, *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, IEEE, pp. 154–160.

Kelly, D., Glavin, F. and Barrett, E. (2020). Serverless computing: Behind the scenes of major platforms, *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, IEEE, pp. 304–312.

Malawski, M., Figiela, K., Gajek, A. and Zima, A. (2017). Benchmarking heterogeneous cloud functions, *European Conference on Parallel Processing*, Springer, pp. 415–426.

Mohammed, C. M., Zeebaree, S. R. et al. (2021). Sufficient comparison among cloud computing services: Iaas, paas, and saas: A review, *International Journal of Science and Business* **5**(2): 17–30.

Patterson, S. (2019). *Learn AWS Serverless Computing: A Beginner's Guide to Using AWS Lambda, Amazon API Gateway, and Services from Amazon Web Services*, Packt Publishing Ltd.

Rajan, R. A. P. (2018). Serverless architecture-a revolution in cloud computing, *2018 Tenth International Conference on Advanced Computing (ICoAC)*, IEEE, pp. 88–93.

Ritzal, R. (2020). *Optimieren von Java für Serverless Applikationen*, PhD thesis, University of Applied Sciences.

Roberts, M. and Chapin, J. (2017). *What is Serverless*, O'Reilly Media, Incorporated.

Sbarski, P. and Kroonenburg, S. (2017). *Serverless architectures on AWS: with examples using Aws Lambda*, Simon and Schuster.

Shafiei, H., Khonsari, A. and Mousavi, P. (2022). Serverless computing: a survey of opportunities, challenges, and applications, *ACM Computing Surveys* **54**(11s): 1–32.

Shahrad, M., Balkind, J. and Wentzlaff, D. (2019). Architectural implications of function-as-a-service computing, *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pp. 1063–1075.

Somu, N., Daw, N., Bellur, U. and Kulkarni, P. (2020). Panopticon: A comprehensive benchmarking tool for serverless applications, *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, IEEE, pp. 144–151.

Use API Gateway Lambda authorizers (2022).
URL: *https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-use-lambda-authorizer.html*