ACIT5930

MASTER'S THESIS

in

Applied Computer and Information Technology (ACIT)

May 2022

Cloud-Based Services and Operation

A comparative study between Microservices and Serverless in the cloud

Beebu Nisha Yasar Arafath

Department of Computer Science

Faculty of Technology, Art and Design

OSLOMET

Abstract

The revolutionary transformation toward cloud computing microservices has gained rapid momentum among IT circles due to its agility, scalability, and resiliency. Many applications are deployed using microservices at their core to increase the agility and flexibility of management platforms. However, serverless computing has become a hot topic when deploying cloud-native applications. Compared to microservices, serverless architecture offloads management of underlying infrastructure and operational maintenance from the user to the cloud provider. Thus, the users focus only on building the business logic. With both these technologies, applications can take advantage of faster delivery, lightweight scalable, and lower development and maintenance costs. Hence, there are debates concerning which deployment strategy to use for their applications.

This thesis provides a comparison in terms of performance, cost, scalability, availability, security, stability, controllability, visibility, and development experience when deploying an application using microservices and serverless in two Cloud Platforms, AWS and Google cloud. Furthermore, the experimental results demonstrate the advantages and disadvantages of each type of deployment strategy under different scenarios and let the Organization decide and choose which deployment strategy to use. The microservice approach is best suited for predictable traffic, which results in better performance with pre-built instances. On the other hand, the large number of requests is handled by serverless due to its scaling agility and cost-effectiveness.

Table of Contents

List of Figures

List of Tables

Acknowledgment

I am grateful to Professor Raju Shrestha for supervising me and as my teacher for all his support, encouragement, and kindness during these two years of master's study. A special appreciation goes to Raju Shrestha as a knowledgeable professor and a great person who always makes time for me and my questions. His advice and comments on my technical questions, as well as on my report by proofreading, were very precise and useful. Thank you, Raju Shrestha, for accepting to be my supervisor and for all your invaluable guidance, encouragement, and constructive suggestions during my thesis work.

Thanks to Oslo Metropolitan University (OsloMet) for offering this master's degree program. I would like to thank my friends for their kindness and support during the last year.

Last but certainly not least, I would like to thank my husband Yasar Arafath, my daughter Tanisha, my beloved parents, my sisters, and my brother. Without your encouragement and support throughout my life, none of this would have happened.

Beebu Nisha Yasar Arafath
May 16, 2022
Oslo, Norway

Abbreviations

IT	Information Technology
AWS	Amazon Web Services
SOA	Service-Oriented Architecture
HTML	HyperText Markup Language
SOAP	Simple Object Access Protocol
WSDL	Web Services Description Language
ESB	Enterprise Service Bus
HTTP	HyperText Transfer Protocol
API	Application Programming Interface
FAAS	Function as a Service
PAAS	Platform as a Service
SAAS	Software as a Service
IAAS	Infrastructure as a Service
BAAS	Backend as a Service
DDOS	Distributed Denial of Service
EC2	Elastic compute cloud
VMs	Virtual Machine
IAM	Identity and Access Management
TCP	Transfer Control Protocol
CPU	Central Processing Unit
SSM	System Manager
YAML	Yet Another Markup Language
OSI	Open System Interconnection

Chapter 1

Introduction

Cloud computing offers computing power, databases, storage, and resources located somewhere else that can be consumed on-demand through the internet, which is pay-per-use. Around the world, these computers are located and can be consumed virtually through cloud providers like Google Cloud, Amazon Web Services, Azure, and other cloud providers (Rajan, 2020). To support the quality and crucial attributes of the software in the software lifecycle, software architecture plays a vital role (Ghayyur et al., 2018). In this thesis, Microservices and serverless architecture are developed and deployed in two cloud platforms. And the comparison is made on which architecture suites based on the requirement, literature study, and experimental results.

Microservices is an architectural pattern in which applications are broken down into independently deployable small services, Hence the term "microservices"(Chris Tozzi, 2021). Microservices have fine-grained business capabilities and are isolated from each other. Moreover, a microservices runs on its own process and communicates using lightweight protocols and standardized interfaces (Viggiato et al., 2018).

Microservices architecture has been used by many organizations to achieve a high degree of agility, speed of delivery, performance, and scale. Many organizations such as Uber, Netflix, and Amazon successfully used the divide-and-conquer technique to break their monolithic applications into smaller units. These organizations solved some prevailing issues they were experiencing with their monolithic applications. Following the success, many other organizations started adopting this as a typical pattern to refactor their monolithic applications(Packt, 2017). Microservices have gotten into the mainstream last few years along with the spread of DevOps practices and holder's advancements, like Kubernetes and Docker (Pahl & Jamshidi, 2016). We can see an expansion in the use of microservices architectural style since 2014 (Viggiato et al., 2018). The usage of microservices is high when compared to other software architecture models, which can be verified in the service-oriented industry (Richards, 2016). Microservices are more adjusted to business capabilities and have independently manageable life cycles, and they are the perfect choice for enterprises on DevOps and cloud.

On the other hand, Serverless Computing allows you to build and run the applications without thinking about servers. Compared to the microservices, serverless architecture discharges the effort of server management from the application developers, who now have to focus on the application logic (Baldini et al., 2017). With serverless, users can build nearly any type of application, and backend service that demands scale and high availability is handled by the cloud provider.

In recent years serverless has evolved due to offerings and investments from cloud providers. Some software teams spent time justifying a migration towards serverless. Usually, they needed to address management's concerns about cost, performance, and scale. All those things are relevant and remarkable to ask when exploring any new architecture. Still, advanced serverless observability is booming to fill the gaps in the market.

Both serverless and microservices computing has their own advantages and disadvantages, but the decision to adopt a design pattern depends on the business and project requirements. Hence there are debates concerning which deployment strategy to use for their applications. As a result, a comprehensive study and experiments are required to compare serverless and microservices (Sadaqat et al., 2018).

Presently, works of literature are not available for this topic which covers all the aspects of an application such as performance, cost, scalability, availability, security, controllability, visibility, stability, and development experience. This led us to conduct a study with experimentation. The main theme of this thesis is the comparative study between microservices and serverless deployment by evaluating the quantitative and qualitative metrics.

1.1 Problem Statement

With the evolution of the cloud, especially in the field of public cloud, Microservices architecture has already been well accepted by the developer community as it has developed the necessary club in part with cloud computing quite well. But since change is a natural phenomenon for further betterment, another architecture called serverless got evolved these days. In contradiction, few things are common in both microservices and serverless. To achieve modern, future-proof architecture, both have it. To build a better architecture for large-scale application development and leveraging distributed systems with innovations, both have that too. Where technology becomes crucial in the organization on what to choose in deploying an application, it is a key differentiator. To better understand addressing their business needs with the help of the technologies, business executives have a conversation with IT leaders on what to choose due to its much more similarities. Architects, Site Reliability Engineers and DevOps Consultants in most IT Organization finds it difficult to decide which one is best for their business applications and for their organization. There is always a gap in seeing the key difference between microservice and serverless from a holistic view covering all aspects of an application. The trigger for this research starts here. For the development of the project choosing appropriate infrastructure and architecture plays a crucial start and important step in leading the application to success. This will set the boundaries for the application at the beginning of the project. If an organization fails to choose a better and proper technology later date, it costs more to change and waste of time. Therefore, for the application to set up a long-term environment decision needs to be taken by the architect.

1.2 Motivation

The focus of the thesis is to provide and create a clear and informative document both theoretically and experimentally, saving both time and money for the projects. This gives an opportunity for the architect to decide on what architecture to choose for the requirement of the project. Taking off the gloves, what are the similarities and differences? What are the limitations and benefits? To find the answer, detailed research and experiment are carried out under the top comparison between serverless and microservices.

1.3 Objective and Research questions

The objective of this thesis work is to help the organization make an informed decision on the choice between microservice and serverless architectures and deployment strategy for cloud-based applications through a comparative study and by doing hands-on experiments. This is done in the context of deploying a backend system to support a cloud-based application on two cloud platforms such as AWS and Google Cloud. The implementation is done in AWS and in Google cloud for choosing the right technology platform by evaluating quantitative and qualitative attributes between microservices and serverless. In order to achieve the defined objectives, the following research questions are formulated:

- RQ1: What are the main factors that influence the decision of choosing between microservices and serverless?

- RQ2: Which architecture among the two (Microservices and Serverless) is suitable for deploying an application in the cloud?

I believe that this thesis work will be valuable to the enterprise business and industries involved in software development activities in providing a good knowledge and understanding of how, when, and where to use the microservices and serverless. It also eliminates the business risks due to selecting the wrong technology stack and providing good business value.

1.4 Outline

This thesis work consists of 8 chapters and is organized as follows:

- **Chapter 1** is the introduction has coverage of essential topics related to the history and periodic progress of microservices and serverless cloud computing. Introduction also includes subsections are problem statement, motivation, objective, and research questions.
- **Chapter 2** is about a literature study that has subtopics like background and related works. The background explains a brief knowledge of monolithic SOA, microservices, serverless technologies, benefits, and limitations. Related work shows quantitative and qualitative analysis of existing research.
- **Chapter 3** Discusses the methodology, which carries how data are searched, a systematic review of qualitative and quantitative methods, techniques and tools used in experimenting and providing results, and phases of research.
- **Chapter 4** Presents the application used and architecture design of both microservices and serverless in AWS and Google Cloud.
- **Chapter 5** presents an implementation of both architectures and deployment in microservice and serverless.
- **Chapter 6** Describes the experiment results, analysis, and evaluation between microservices and serverless metrics.
- **Chapter 7** Discussion, limitations, and challenges.
- **Chapter 8** Conclusion and future work.

Chapter 2

Literature Study

This chapter provides background about microservices, and serverless cloud computing has been reviewed to understand the development and existing issues. In addition, a number of research articles are studied deeply two technologies, microservices and serverless in the cloud. The essential findings of the related works are presented below under the background.

Section 2.1 provides an overview of the evolution of software architectures, starting from traditional to the most recent serverless architecture. This provides knowledge about how monolithic shifts to service-oriented, microservices and, at present serverless too evolved, which are similar in few concepts and contradictory in many. Section 2.1.1 briefly discusses the monolithic architecture and how large application was managed by the organizations. Some of the drawbacks that lose its popularity with some difficulties in the Organization. Then, section 2.1.2 is about service-oriented architecture and some of its drawbacks. Section 2.1.3 is about microservices architecture on how the large application broken down into small services and attracted by large companies moved their application to the cloud. Therefore section 2.1.4 about serverless computing is the trending topic that attracts many companies with its effective features are briefly explained below.

2.1 Background

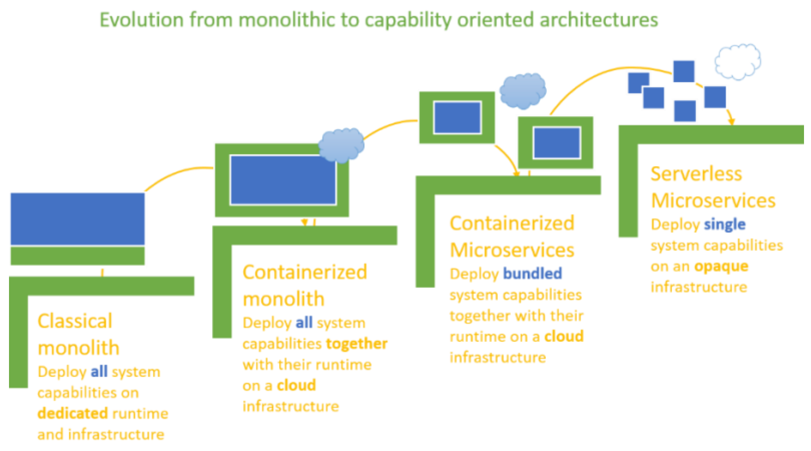

Evolution from monolithic to capability oriented architectures
Classical monolith
Deploy all system capabilities on dedicated runtime and infrastructure
Containerized monolith
Deploy all system capabilities together with their runtime on a cloud infrastructure
Containerized Microservices
Deploy bundled system capabilities together with their runtime on a cloud infrastructure
Serverless Microservices
Deploy single system capabilities on an opaque infrastructure

Figure 2.1 Evolution from Monolithic to Serverless Architectures (Zollingkoffer, 2017)

Figure 2.1 (Zollingkoffer, 2017) depicts how serverless evolve starting from monolithic, where an application is tightly bundled into a single package and run as one process in dedicated infrastructure, either on-premises or in the cloud. Frontend, application layer, and data store layers are tightly coupled in monolithic applications. There are some difficulties like long start-up time, single point of failure, and maintainability. To overcome this, microservices were introduced where an application is broken down into small independent, loosely coupled components working together to perform a task that forms a decentralized architecture. However, microservices are widely used across organizations running large applications. The evolution of serverless technology, which massively scales on public cloud platforms. It has gained more attraction among IT Architects and developers. In Serverless, cloud providers are responsible for running a piece of code by dynamically allocating the resources at the backend and only charging for the number of resources execution time used to run the code (Ivanovic, 2021).

2.1.1 Monolithic Architecture

Monolithic is an architectural framework. An application is bundled into one package and run as one process in a dedicated infrastructure. In monolithic architecture, the whole code is encapsulated into one single application, so each piece of code cannot be executed independently (Ponce et al., 2019).
Monolithic application with a single large codebase with one team (Ghayyur et al., 2018) offers tens or hundreds of services using different interfaces such as HTML pages and Web services. This codebase is given to multiple developers if they want to make any upgrades or changes in code. Artifacts are managed by the team manually. If the single artifact needs to deploy many times per day, then the whole application deploys on multiple machines. Here, developers and the operations team work separately. Scaling in monolithic is challenging because the same application with different services has some irregular consumption, including additional infrastructure for the whole application. Therefore, the unused resource for the other service is wasted, increasing the cost (Lehmann & Sandnes, 2017). However, one component of the application experiences load, which then scales the whole application. Even though that particular component needs to scale cannot be performed as it has a single process (Dragoni et al., 2017). Some technical and business-related issues are

highly coupled. Therefore, system maintenance is hard. Releasing new features took a long time and low productivity among developers. Apart from this, the application becomes complex, and as a team grows, it has some drawbacks. They are:

- The large application becomes complex to understand and modify. This leads to a slow-down the development.
- Few changes to a small part of the application need a whole application to be rebuilt and deployed. So, continuous deployment is complex.
- Long-term commitment to technology stack required.
- Monolithic can be scaled horizontally, as scaling to extend difficult (Ponce et al., 2019).

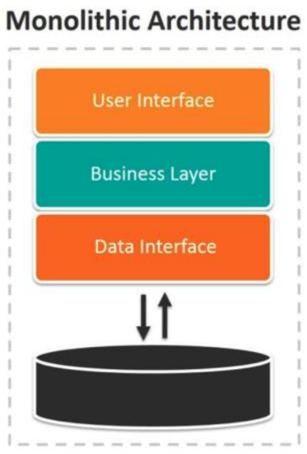


Figure 2.2 Monolithic Functional Architecture (Ravirala, 2019)

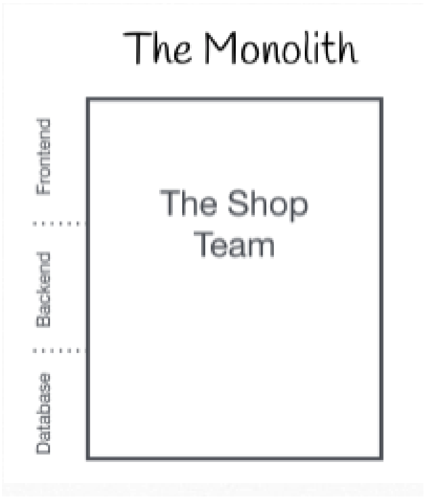


Figure 2.3 Monolithic Operational Model(Maherchandani, 2019)

Figure 2.2 (Ravirala, 2019) and Figure 2.3(Maherchandani, 2019) depict how a monolithic architecture can be visualized from a functional and operational perspective. For example, in a 3-tier application, all the components are bundled together and stacked into one module. However, it does not need to be single regarding the number of machines the application is running.

2.1.2 Service-Oriented Architecture

Service-Oriented Architecture (SOA) is a pattern designed to build a distributed system that offers services to other applications through the protocol. Figure 2.4 demonstrates SOA, the user sends a request to the service provider, and the service provider sends the response to the service user. Thus, both the service provider and the service connection are understandable (Javatpoint, 2020). This traditional architecture mainly focused on integration solutions. It has a heavy middleware technology that sends messages to services and strongly relies on an enterprise service bus that acts as middleware. Protocols used by service-oriented architecture are SOAP and WSDL (Jamshidi et al., 2018). In SOA, the module concept takes place. However, it has large granularity (Gan et al., 2019). Like monolithic, SOA has large granularity, which means behind the service-oriented interface hides the whole monolithic application. If the load hits the application, instead of scaling particular services, it scales as a whole and requires high availability (Dragoni et al., 2017). It is loosely coupled and self-contained.

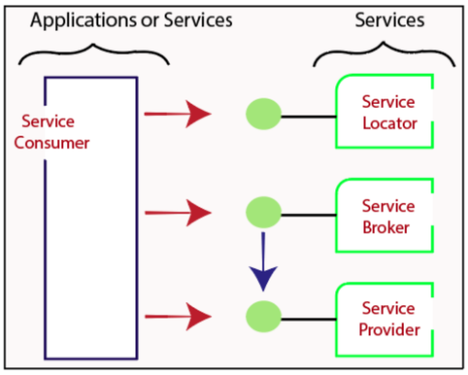


Figure 2.4 SOA Functional view (Javatpoint, 2020)

Some of the drawbacks of SOA are:

- Few organizations might need to implement SOA to meet their requirement. However, in that case, the large application becomes complex to develop and deploy the application.
- As the application grows, it consumes time and becomes expensive.

- Thousands of users in enterprise applications support workload and ESB as it interacts with the internet for scaling the application that has millions of users, leads to bottlenecks, provides high latencies, and a single point of failure.
- Adding and removing servers on-demand makes it complex as it is not designed for a cloud environment (Villamizar et al., 2017).

2.1.3 Microservices Architecture

Microservices architecture is used by many organizations that have started gaining popularity. It evolved to build individual software applications that can be designed, developed, and delivered in the form of modularization. But it has some technical boundaries. It is smaller in size and independently deployable, as shown in Figure 2.5 (Ravirala, 2019), and in figure 2.6, both the DevOps team work together. Each service is lightweight and has access to its data and internal logic. It relies on HTTP and API protocols. It is a subtype of SOA. The benefits of microservices are greater autonomy, faster delivery, and improved scalability. Many organizations have this architecture as a standard way of developing their projects (Jamshidi et al., 2018). Each service is a piece of software that has its own process that encapsulates some functionality, the Distributed task is performed, and the configuration of servers depends on the requirement done by teams. Each service is taken by a single user or team with five members, and yet it is a standalone server. Continuous integration and continuous deployment take place as new features are added to the application (Lehmann & Sandnes, 2017). However, microservices is a decentralized framework where each service performs one specific sub-task that comes up with a security mechanism in a trust-less network environment (Xu et al., 2019). It works on the latest platform, and during input load, microservices handle heavy concurrency of load (Ghayyur et al., 2018). DevOps comes along with microservice architecture in both industry and academia. These both have been a top trend in technology since 2014. It has a set of practices like integration, deployment, testing, and monitoring. Versioning is done in collaboration with developers, testers, and operators. Hence, the cross-functional team was performed, and the team's throughput and quality were increased along with automation tools and the fast feedback mechanism that helps to deliver applications quickly. Combining both enables the reuse of code in terms of reusability, and decentralized data governance offers agility and productivity that leads to low time to market (Waseem et al., 2020). It uses different programming languages like Python, C/C++, Go, JavaScript, Node.js, etc. Each service communicates through a well-defined network interface. Because of fine granularity, it reduces system complexity. The data center requires more hardware, network, operating system, and programming language framework while scaling large applications. To avoid this complexity, opting cloud to develop and deploy applications can be performed (Gan et al., 2019).

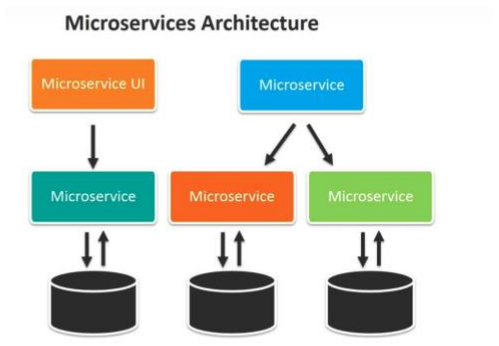


Figure 2.5 Microservices Functional Architecture (Ravirala, 2019)

2.1.3.1 Benefits

There are more benefits of using microservices architecture are below:

Distribution: SOA is also distributed in nature. However, microservices are smaller in size. Because of its extreme characteristics, business capabilities and their functionalities can be managed as an independent service and deployed on a host. As a result, the whole system can be managed efficiently as compared to the monolithic. It increases the availability of an application since the failure of a microservice doesn't affect another one (Dragoni et al., 2017).

Portability: Microservices are portable in nature. They are packed in a docker container in which all its dependencies, such as databases, and libraries, are included so that they can be deployed in a heterogeneous platform. Different versions of the same libraries were used without any conflicts, scaled, and relocated to other platforms.

Elasticity: The model of multiple services per host is better to be deployed in the cloud as when the load is at its peak, it uses resources efficiently and scales up. And once the load is stable, it will automatically de-provisioned the container and releases the host.

Availability: Microservices are highly available. They can replicate across datacentres and geographical locations to distribute the load. Another aspect is service updates. While updating a monolithic application, the entire system must stop and restart for a new feature which leads to long downtime. In microservices, because of its independence, a business capability can be updated by stopping the specific service related to the functionality and recreating, whereas the other service remains up and running. This leads to faster deployment time and no single point of failure.
Replaceability: As microservices are small independent services in a whole application, each service can be replaced easily. It can be renewed and replaced one by one, reducing the risk and does not impact other services (Dragoni et al., 2017).
Reconfigurability: In microservices, updates can be easily performed, and one of the main advantages. Protocols like HTTP/API allow a new version for service if any changes are done in the existing application (Gan et al., 2019).
No long Downtime: Application composed of multiple microservices; a portion of an application can be revised that offers recovery. Therefore, it does not affect the application as a whole (Lloyd et al., 2018).

2.1.3.2 Limitations

Some of the limitations of a microservices architecture are below.
Lack of relevant skills: Beforehand, using and developing applications with microservices architecture requires a clear knowledge of technology. When developing a single small application might not be so complex. However, as heterogeneous technology stacks come in for effective and efficient implementation, the large application requires advanced skills. The application complexity significantly increases as scaling, recovery, continuous development/delivery, monitoring, etc., cannot be managed manually. DevOps requires DevOps tools like Jenkins for CI/CD, Docker for deploying containers, Docker swarm for clustering orchestrating containers, Kubernetes for load balancing, and for scaling. It also requires a better understanding of fault tolerance, latency, non-functional aspects (Baškarada et al., 2020).
Resource monitoring and management: When a microservice application grows complex in size, infrastructure resources like servers, services, and containers become hard to monitor

and manage during runtime. Services are in different regions, and zones that generate information leads to data flow that becomes hard to monitor. With the help of automated tools, the flow of information can be monitored. This can be controlled by alerts even though it becomes strenuous to handle (Jamshidi et al., 2018).

Organizational structure: Having an independent team of developers for each service to develop and deploy might be a double-edged sword. Each member has the freedom to make decisions locally within their team. However, it might affect the business goal in an overall picture.

Governance: One of the challenges is to shift from the centralized governance model toward distributed one. Governance of microservice is distributed, which means service owner is responsible for the overall governance of their own services. Any changes in the service network interface or API endpoint will have a huge impact on the other service as both services will communicate together via interface/API endpoint. This cannot avoid, but proper communication and coordination with other teams will significantly reduce failure (Baškarada et al., 2020).

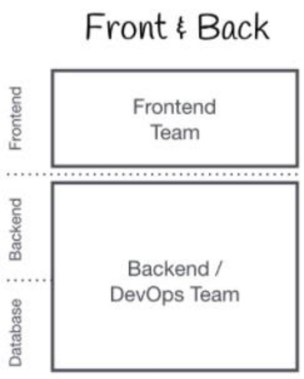


Figure 2.6 Microservices Operation Model (Maherchandani, 2019)

2.1.4 Serverless Architecture

Serverless is called function as a Service (FaaS) is an execution model most widely used by many organizations these days. It's a combination of 'Backend as a Service' and 'Function as a Service' (Kumar, 2019) adopted by most organizations that gives a production-ready-to-use platform (Adzic & Chatley, 2017). Serverless is a way to develop and build more agile applications. It provides a platform to run your piece of code with a prebuild runtime environment so that you can innovate and respond to change faster. Serverless doesn’t

mean there are no servers. It exists, but the maintenance and operational complexity of building server infrastructures are taken care of by the cloud providers so that the user or developer will focus on innovating their business functionalities.
Developers are free from choosing instance types and the number of instances as in microservices architecture. Serverless is on-demand. You pay for what you use. Cost is calculated based on the time duration taken by the function to execute the code. Therefore, the cost is charged less as, after execution, the resources provided by the cloud provider are released from the function. The function starts in a microsecond and shuts down in 15 minutes, applicable in Amazon Lambda (Shafiei et al., 2019). Users do not want to worry about scalability, load balancing, high availability, and security maintenance of the underlying hardware (Eivy, 2017).

Application logic breaks down into functions and is defined by actions and event that triggers them. It really works well, and it is a good fit for mobile and IoT applications (McGrath & Brenner, 2017)**.** IoT devices are inexpensive. It requires minimal software and hardware; it collects the sensor reading and pushes it to the cloud at a regular interval (Savage, 2018). Serverless technology has different patterns that provide solutions to solve problems. Based on the requirement, a specific pattern was chosen to deploy an application. Some of those patterns are big data patterns, automation and deployment pattern, and Web application pattern (Taibi et al., 2020). In a serverless platform, multiple individual functions are composed with a choice of the runtime environment to be selected for their applications to run. One of the key issues with serverless functions is a cold start which means the time taken by the backend system to initialize your function for the first time. However, it is now possible to reuse an existing container to avoid time delay by pre-warming. This is otherwise known as warm-start (Jackson & Clynch, 2019).

Most organizations migrate to serverless because of its cost-efficiency. Cloud providers like Amazon and Azure take first and second place for deploying application decisions taken by organizations. Services like monitoring, logging, database, and storage are not much available in a private environment. Certain qualities are better provided by public cloud providers like Amazon, Azure, Google Cloud, etc. (Eismann et al., 2020). Functions are short-lived, which means ephemeral and isolated. It is stateless, and data is stored temporarily

(Hellerstein et al., 2018). Service is breakdown into a micro function that scales independently during execution and is termed nano services. The process in which the transformation of monolithic or microservices into functions is called FaaScification (Albuquerque Jr et al., 2017).

2.1.4.1 Benefits

Some benefits of serverless architecture below:

Scalability: One of the well-established benefits of serverless computing is scalability. Customers deploy their application without worrying about any scaling issues in serverless as it guarantees a highly scalable feature during execution. It is the nature of FaaS where it automatically scales or shrinks based on the on-demand. Scaling in serverless has no limits.

No maintenance and deployment complexity: Another benefit of serverless is free of management and maintenance infrastructure. This saves time for developers who desperately focus on production and do not want to manage infrastructure. In IaaS, users must install and configure packages and libraries. In PaaS, management is quite easier, and anyhow we still must configure some requirements in the application. In serverless, before executing code, register for function and, with the help of credentials, invoke the function. The underlying resource is carried out by cloud providers (Pellegrini et al., 2019).

Low cost: Unlike microservices, you pay only for the time the function runs. When resources are ideal, do not require to pay for underlying resources.

No infrastructure security required: At the backend, cloud providers manage their own infrastructures. The client-side only focuses on developing and deploying code in a function and maintaining security only for the code. This leads to zero responsibility for the security of backend infrastructures.

Low time to market: Setting up infrastructure is not required in the serverless concept. This gives an opportunity for all developers to create, manage and maintain only code and resources work is offloaded. This provides developers to deploy an application with minimal time (Kumar, 2019).

2.1.4.2 Limitations

Some limitations of serverless architecture are below.

Vendor lock-in: Once the application code is executed, it relies on underlying resources that cloud providers offer in serverless platforms. The code of the application is totally coupled with platform services. This leads to code dependent on services offered by the cloud provider. Further shifting the application code to any other cloud provider platform creates a work to rewrite the whole code.

Short life span: Control of underlying resources is not accessible by application developers. The activeness of function for the allotted code is short as it is stateless and ephemeral. All are managed and maintained by cloud providers. A single task is performed at the given time.

No local execution environment: The code environment created by the developer has no way to run and test code on his or her own local server. The platform provides a versioning, monitoring, and testing environment as the workflow, which is followed traditionally, was impossible to run the code for testing purposes. Unlike microservices, Working in a team becomes hard (Adzic & Chatley, 2017).

Start-up Latency: Cloud offers different run time environments to build and run the application. Most used are Java, python, and recently GO... Comparatively, Java takes a longer time, whereas python is lightweight and has a faster initialization time. Once a function is invoked and executed with no further request, the backend container goes to sleep mode and automatically de-provisions the backend resources. Again, when a new function is invoked, a new container spins up, and there is a high chance of start-up latency (Pellegrini et al., 2019).

Security problems: No access to backend resources managed by cloud providers; this may lead to testing, monitoring, support, and maintenance of the serverless application is complex with distributed nature chance of vulnerability due to a wide attack surface (Kumar, 2019).

2.1.5 Cloud Platforms

This section describes the three major cloud platform that supports microservice and serverless environment.

2.1.5.1 Amazon Web Services

AWS holds the market position in the cloud space with the more advanced set of serverless products. One such product widely used is AWS Lambda runs on an underlying container platform on Linux operating system (Lloyd et al., 2018). Lambda allows developers to build and run their applications. AWS hosts a public git repository where serverless projects are provided for public use with sample serverless architectural patterns that the developer can modify according to their business requirement. EC2, a compute resource, can be used to host microservices. Lambda@edge allows running the code nearest to the user location. Amazon S3 is object storage that allows the storage of objects and retrieval of data. Developer tools are Cloud Watch for monitoring, CI/CD code pipeline, and more. Amazon Kinesis is used for real-time data. The workflow orchestration-like step function provides a visual workflow for lambda. SNS, SQS is a cloud messaging service. For security IAM and Amazon Cognito are used. Dynamo DB is used for storing data in key-value pairs. AWS has its own flavor of the operating system known as Amazon Linux.

2.1.5.2 Microsoft Azure

As per Gartner, Microsoft Azure takes the second position in the cloud space. Some of the services provided by Azure are Azure functions that are similar to AWS Lambda. Runs code on IoT devices. It can run in intermittent connectivity conditions. Azure Cosmos DB has distributed a NoSQL database used for storing data. Azure storage is highly available and scalable object storage. For security and access control, Azure Active Directory is used. It offers workflow orchestration called Logic Apps to integrate with different systems writing complicated coding that is not required. For messaging events, the grid eliminates the need for polling and manages event routing services (Kumar, 2019). Azure Function is built on the Azure app service, an extension of web jobs with some effective features like scaling (Lloyd et al., 2018).

2.1.5.3 Google Cloud

Google Cloud Platform (GCP) is offered by Google. It provides a suite of cloud services that run on the same infrastructure that Google uses. It was ranked 3rd as per the report from Gartner on the magic quadrant for cloud infrastructure platforms and services. In terms of regions available, Google has the 2nd highest number of regions of 25 compared to AWS and Azure, with a total of 24 regions and 52 regions available today. Regions are something that can be important in terms of latency for some applications (Google, 2020). Google Cloud Function is a serverless offering from google cloud which can be compared with AWS Lambda and Azure Functions.

2.2 Related Work

This section presents the main theoretical contribution, which consists of several separate literature reviews on the comparison between microservices and serverless. In the end, this chapter concludes with the implications of the found literature for this work by answering the research questions.

Villamizar et al. (2017) conducted a study where the performance and cost of three different software architectures. They described the process of implementing a system in the monolith, microservices, and serverless architectures and the challenges faced during implementation. All versions of the application were deployed on Amazon Web Services, and serverless functions were run in lambda. By running performance tests and making cost comparisons, the study concluded that using FaaS platforms such as AWS Lambda can reduce infrastructure costs by up to 77.08%. Based on a review, when the request is static, the serverless function has a better performance in response time and scaling. Whereas, in microservices, the researcher found that the response time starts to rise when there is an increase in the workload. However, after the scale-out, the response time has decreased in microservice. In the serverless strategy, there are also high response times at the beginning of each test due to the cold-start problem, but afterward, it becomes stable. But the serverless has a lesser duration of high request response time than microservices. In another paper, Fan et al. (2020) conducted a study on microservices having a cost advantage for long-running services with the regular pattern over serverless due to limitations of maximum runtime in serverless functions (AWS Lambda). On the other hand, a request accompanied by the large size of the response with a random spike traffic pattern is more suitably handled by the serverless because of its scaling and agility.

In another study, Albuquerque Jr et al. (2017) deployed a simple application, where one version of the application was deployed as container-based microservices and another version as a serverless function in AWS Lambda. Measured performance between these two by sending a high amount of HTTP traffic to the application, triggering different application functionalities. The author performed some experiments on scalability and performance, and they found two solutions similar for performance, where cold starts can have a negative impact on serverless functions. The study also compared the cost between the two platforms and found that PaaS is more economically suitable for applications with longer or varied execution times while FaaS has a better cost-benefit for requests with short and predictable execution times. Also, the author kept their own topics for future research the below:

- Analyze cost by running the same set of applications in the different cloud service providers to get a more in-depth cost analysis. Serverless enables more efficient use of resources, thus making cost spent on infrastructure zero.
- Address cold start issues and possible solutions or techniques to avoid a cold start

Figure 2.7 shows the cost comparison between serverless function and VM in AWS. (BBVA, 2020)

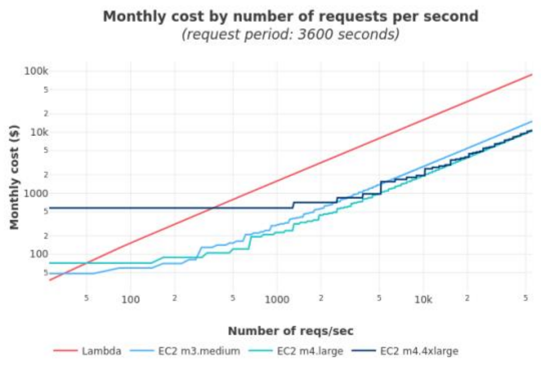


Figure 2.7 Cost Comparison in AWS on Lambda Vs. EC2 (BBVA, 2020)

In this paper, Adzic & Chatley (2017) explained serverless functions such as AWS Lambda automatically scale out and scale in their backend instances, so a new request might end up creating a completely fresh instance. Users have no control over this process. Creating an application with a new Lambda instance using python for code takes about one second, and in Java, it takes between three and ten seconds for other environments. The instance which is not reused can be active for 3 minutes. Later it is removed automatically after the inactivity for a longer period of time. There is no clean observation and statistics on this. As per the author, any infrequently accessed services might have constant high latency. Even for frequently accessed services, users might experience some additional latency during the process of new instances getting created. During this research, the author mentioned that it couldn't be possible to guarantee very low latency for each request deployed on lambda. Here Sadaqat et al. (2018), in a serverless computing study, mentioned that serverless takes operation concerns away from the developers and lets the cloud providers manage. And developers rely on provider solutions to monitor the deployment and execution on the serverless. This leads to additional costs for the serverless architecture, which includes monitoring, logging, and debugging. In the same study, the author mentioned that the cloud providers control the underlying infrastructure, and developers won't be able to customize or optimize the environment as they require, such as removing, updating functionalities, or changing APIs.

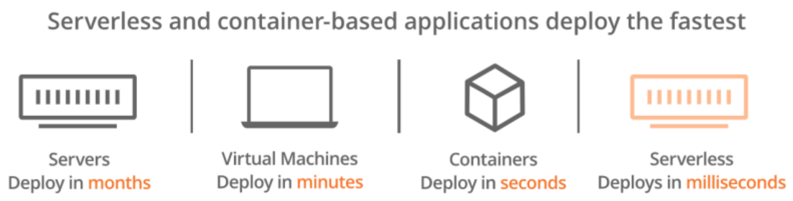


Figure 2.8 Time to Market (Cloudflare, 2021)

Figure 2.8 depicts that both microservices and serverless have a better time to market. However, serverless outperforms as it does not come with dependencies and takes milliseconds to deploy (Cloudflare, 2021).

On the other hand, Waseem et al. (2020) mentioned that the future study gap has to be focused on a deep study on DevOps and a proper study on monitoring, security, and performance issues. Pellegrini et al. (2019) have provided a clear view of performance in FaaS that creates a holistic view for IT architects and cloud customers to choose as a better solution for businesses. Indirectly it has been applied stress on underlying SaaS, PaaS, and IaaS to copy DDoS attacks. There might be changes in security vulnerabilities considered for future work. Baškarada et al.(2020), in a study, explained that Microservices architecture is widely used and gained popularity in many organizations. However, refactoring found few difficulties among architects. It has some benefits like operational scalability, development agility, and deployment agility. In contrast, some challenges like a lack of relevant skills, organizational structure, governance, organizational culture, and orchestration. Some organizations find it easier to adopt this architecture. Some found as a struggle in the qualitative study can be considered future work.

Based on the findings from the related works, some of the experimentation related to the comparative study between serverless and microservices was performed on one cloud provider, which is AWS. One potential research gap found was multiple cloud providers were not included in making a comparative study. Another observation from the existing research is to compare cost by running the same application in multiple cloud platforms between two approaches with two different memory configurations. And finally, no work was found covering the qualitative attributes such as security, development experience, controllability, and visibility.

To address the gaps found from the related work, experimentation is conducted by leveraging two cloud providers and then performing quantitative analysis, which drives the analysis for qualitative attributes. Hence this experimentation is needed to close the gaps.

Chapter 3

Research Methodology

3.1 Data Sources and Search Strategy

The study is based on a systematic review of the literature to find out the comparison between microservices and serverless approaches, which addresses organizational challenges in defining the right deployment strategy. The primary decision was made to include all literature from the last six years to capture any potentially related work prior to the beginning of the serverless trend, as microservice are already established and got famous within the IT sector. As a result, the publications search date was set to 2016 and till now.

Initial Search: For the systematic literature study, databases such as IEEE, Google Scholar, and Research gate were queried. To increase the possible literature, strings like "comparison serverless vs. microservices" and "serverless vs. containers" were queried. The academic literature on the comparison between serverless and microservices is very limited. There are no papers on the comparative study covering all aspects such as cost, performance, security, development experience, etc.

Two different papers were found but were only limited to cost and performance. The initial search resulted in collecting only two papers in total which is related to the research objective. Therefore, the study was conducted on MLR. MLR includes all accessible knowledge on a topic in the form of grey literature such as white papers, web pages, and blogs with the combination of academic literature, but still, those papers lag the depth required for this study.

Screening based on focus area: Due to the lack of literature papers on the comparison topic, a further review of the literature was carried out against the individual platform (serverless & microservices). Papers like Quality aspects of Serverless Architecture, Serverless Architecture efficiency, and a systematic review on microservices were reviewed individually. Data points were collected on each paper and evaluated performance, scaling, and cost both qualitatively and quantitatively for serverless and microservices.

3.2 Phases of Research

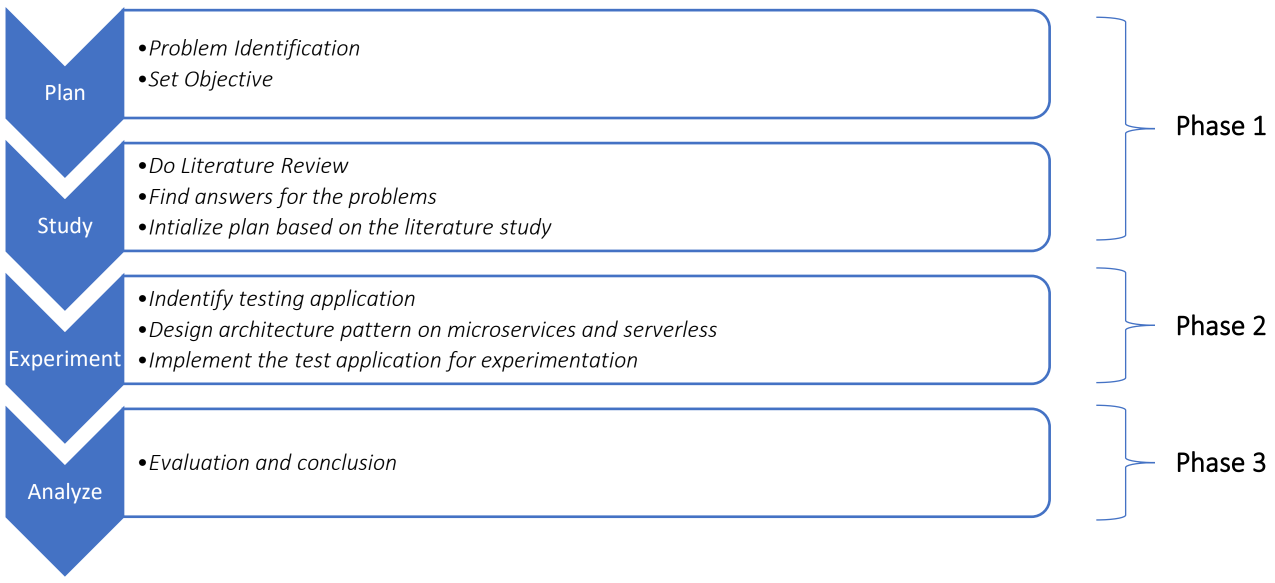

Plan
•Problem Identification
•Set Objective
Study
•Do Literature Review
•Find answers for the problems
•Intialize plan based on the literature study
Experiment
•Indentify testing application
•Design architecture pattern on microservices and serverless
•Implement the test application for experimentation
Analyze
•Evaluation and conclusion
Phase 1
Phase 2
Phase 3

Figure 3.1 Phases of research

3.3 Research Methods

The thesis is carried out in three phases. The goal for each phase is to find answers to the research questions and to gather valuable insight that can be used by anyone interested in deploying their application in the cloud using serverless and microservices. Phase one is about gathering background knowledge, focused mainly on understanding what monolithic architecture microservices are and how serverless is evolved and what challenges they introduce.

A literature study is performed to understand how researchers choose the right deployment platform for their application using microservices and serverless in the cloud. In addition, with no prior experience regarding microservices and serverless, which brings several uncertainties. Therefore, exploring the subject as part of a literature study could provide an opportunity to increase knowledge and dive deeper into the topic continuously.

After the literature review is complete, new experimentation is proposed based on the outcomes, and gaps are found in the literature review. This approach aims to fill the gaps that were missed during the previous research and find the answers to the research questions. The literature review formed the basis of experimentation.

During the second phase, an implementation strategy is created. This is split into different sub-phases depending on the size of the application and the number of public cloud platforms leveraged. Firstly, to build new infrastructure for microservices deployment in cloud platforms such as AWS and Google to deploy a sample application for the research and also build serverless architecture in the same platform for a detailed comparative study. A controlled experiment is conducted. A web application is deployed to Amazon Web services and Google cloud. The application was tested under four load patterns that simulate the real production traffic.

During the final phase of the research, experimental results are evaluated, and find the answers to the research questions. In order to address the research gaps found in the literature study and also to extend the comparison between microservice and serverless from the aspect of security, development experience, scalability, controllability, and visibility, further experimentation is conducted to evaluate the metrics and make a comparison. Both microservices and serverless architectures are compared and analyzed qualitatively and quantitatively. For the evaluation of results, post-experiment following metrics were analyzed.

3.3.1 Qualitative Analysis

As part of the qualitative analysis, the following aspects are considered to compare microservices and serverless.

- **Scalability:** The ability of the system to handle the stable and unstable load that must perform well during the up and down of the traffic(Gearheart, 2021).
- **Security:** To analyze the security feature between microservice deployed applications and serverless.
- **Development Experience:** To analyze the experience of a developer while deploying an identified application in the cloud with both the deployment pattern. Both pros and cons were evaluated.

- **Controllability and Visibility:** The ability to change the backend resources and the ability to visualize what is happening at the backend resources during scale.

3.3.2 Quantitative Analysis

The two architectures are compared and analyzed quantitatively based on the following attributes.

- **Performance**
    - **Response time:** Response time refers to the amount of time taken by an application to return the response back to the user(DNSstuff, 2019).
    - **Throughput:** Number of the requests processed successfully per second by an application in the backend.
    - **Memory Utilization:** Amount of memory utilized by an application to process the submitted request.
- **Cost:** spent on serverless vs. microservices in the form of numbers.

The above quantitative and qualitative metrics is the main source to evaluate the microservice and serverless deployment strategy. Response time is one of the key evaluation metrics which has an impact on the application performance. If the application is slow, there is a chance that the user will leave the application, which impacts business (PrinceSinha, 2022). Cost is another key factor that has an impact on the organization's estimated expense. The result of quantitative analysis drives the qualitative attributes.

3.3.3 Tools and Techniques used

Cloud Platform: Amazon Web Services and Google cloud is used as the cloud platform to conduct experimentation in this thesis. Most of the existing research is conducted using AWS. The reason for choosing AWS is that Microservice ECS is a homegrown service. On the other hand, Google Cloud was chosen because of its simplicity, and Kubernetes is a native service developed by Google, and it is user-friendly with CLI.

K6: K6 is used by developers for load testing, mainly developed by Grafana Labs and the community. It is scriptable using JavaScript. Thresholds are set in the script for the four load patterns to produce accurate results between microservices and serverless, which allows delivering which strategy outperforms better than the other in terms of performance(K6, 2022).

Influx DB: Influx DB is an open-source database developed by InfuxData. It helps to store and retrieve time series data. There are two query languages in InfluxDB. There are influxQL and Flux. In this thesis work, Flux is used as a query language that has more functionality.

Grafana: Grafana is an open-source observability stack that is used by many organizations and is well-integrated with other database tools like Influx DB, Prometheus, Graphite, and Elasticsearch. It allows you to visualize and monitor metrics and logs. The dashboard pulls data from the plugged-in data source. The above database tools are supported by Grafana(Grafana, 2022). In this thesis work, Grafana is integrated with Influx DB and used to display response time metrics for four different load patterns. For this setup, Influx DB is configured as a data source in Grafana. Flux query is used to pull metrics from the database, and dashboards are created in Grafana to visualize.

Serverless Framework: An open-source framework is used to deploy the serverless application in the cloud.

Microsoft Excel: Used to capture the metrics from AWS CloudWatch, Google Monitoring explorer, and Grafana, which then calculated to generate average from the results and create graphs.

Docker-compose: To build load test and monitoring tools that define and share multi-container applications.

Chapter 4

Application and Architecture Design

Architecture design with the components used will be addressed and presented in this chapter. It begins with the high-level architecture and understanding of its underlying components and continues with design approaches for two cloud providers.
To understand and compare how serverless and microservice architecture create an impact on an application with respect to performance, scalability, and cost in an organization, a case study with a sample application is undergone, which is then implemented and deployed in two public cloud platforms.
The rest is organized as follows: Identifying the application for evaluation. Design and Implementation of both the architecture in AWS Cloud and Google Cloud, Scenarios for generating load patterns for microservices and serverless-based applications. Capture metrics from the load test. Evaluate results against performance, cost, scaling and, present the trade-off between these architectures and suggest the best one to offer. Discuss the results, and conclusions are made.

4.1 Application Used

To evaluate the results, an application is required to deploy and run as a microservice and a serverless application. Various sources such as GitHub and AWS samples were analyzed to find the right application for this experimentation rather than reinventing the wheel. After various searches, an image processing application was identified (*Vt3199/Cloudimageproc*, 2022). The core functionality of the application is to get the image URL as an input, download the image from the given URL, resize the image and then upload the image to the backend provided storage. The identified application requires a computing platform to deploy and run. Hence the sample application is targeted to deploy in both Microservices and serverless platforms. The original source code from the public git repository is developed in Node.js and available for use to deploy in the AWS environment. Below the listing is the main function from the source code, which performs resize and upload to the backend.

```
fetchImage(photoUrl)
    .then(image => {
        imageType= image.getMIME();

        if(imageType == 'image/png'){
            objectKey= objectKey+'png';
        } else if(imageType == 'image/jpg') {
            objectKey= objectKey+'jpg';
        } else if(imageType == 'image/jpeg') {
            objectKey= objectKey+'jpeg';
        }
        image.resize(Jimp.AUTO, height);
        return image.getBufferAsync(image.getMIME());
    })
    .then(resizedBuffer => uploadToS3(resizedBuffer, objectKey, imageType))
    .then(function (response) {
        console.log(`Image ${objectKey} was upload and resized`);
        res.status(200).json(response);
    })
    .catch(error => console.log(error));
```

Listing 4.1 Application Source Code

4.1.1 Source Code Changes for AWS environment

For the ease of deployment and to deploy in the thesis AWS environment, the source code is updated with minor changes. Refer to appendices C.1 and C.3 section for the updated source code. Changes done for AWS deployment are environmental variables such as bucket names to store modified images.

4.1.2 Source Code Changes for Google cloud

For deployment in google cloud, source code is modified and updated with google SDKs as it needs to communicate with google cloud storage. Environmental variables are updated. The rest of the function code remains the same, and no changes are made. Refer to appendix C.2 and C.4 for the updated source code.

4.2 General Software Architecture

In this section, general software architecture is designed as 2-tier architecture, shown in Figure 4.1.

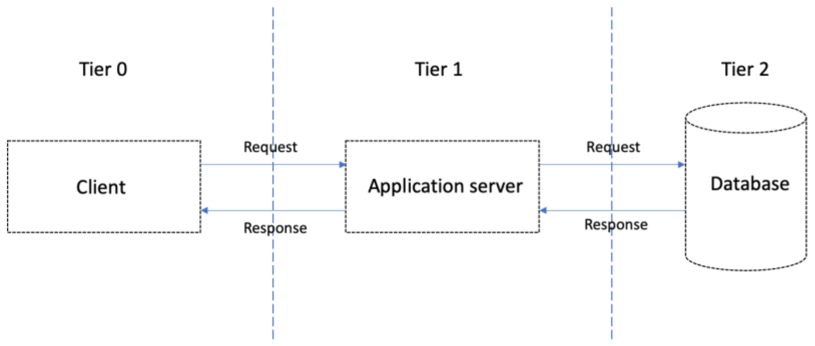


Figure 4.1 Two-Tier Architecture

The client forward incoming requests to the application tier, which does some computational work, and in doing so, it communicates to the backend tier to retrieve and insert data in the database. The application layer returns back the response to the client. The next section explains the architectural design and its detailed description of each of its components in the case of microservice and serverless in the two cloud platforms, AWS and Google Cloud.

4.2.1 Microservice Architecture

A Prototype is designed for the comparative study. A platform is needed in order to deploy an application as Microservice. In this section, a high-level architecture is designed for Microservice deployment in both cloud platforms. For a detailed comparative study and to close the gaps, the design is focused on building a Microservice architecture in Amazon Web Services and Google Cloud. Table 4.1 shows the list of the services used in AWS & Google cloud to deploy the container platform. Each service is mapped to the design component.

Components	**AWS**	**Google Cloud**
Application Server	AWS Elastic Container Services	Google Kubernetes Engine
Frontend Layer	Network Load Balancer	TCP Load Balancer
Compute Layer	AWS EC2	Google VMs
Monitoring	CloudWatch	Monitoring Explorer
Authentication	IAM	Service Accounts

Table 4.1 Cloud Components

4.2.1.1 Microservice architecture in AWS

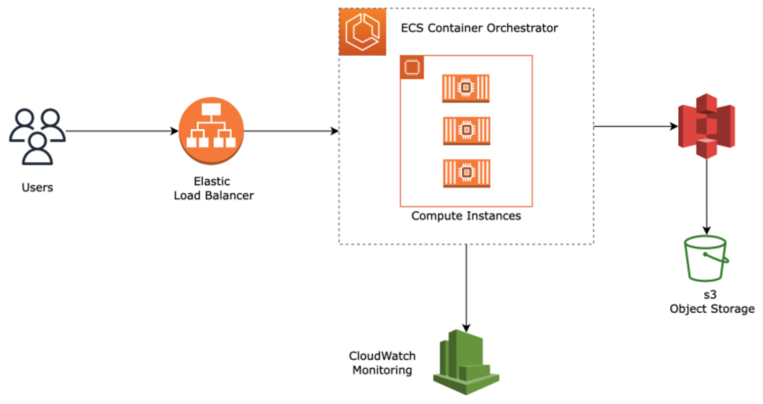


Figure 4.2 Microservice Architecture in AWS

4.2.1.2 Microservice architecture in Google Cloud

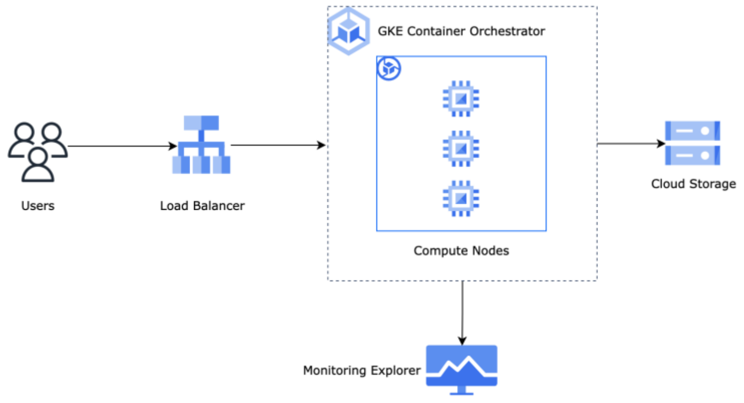


Figure 4.3 Microservices Architecture in Google Cloud

Figure 4.2 and Figure 4.3 are the high-level 2-tier architecture targeted to deploy a microservice application in AWS & Google cloud.

Database Layer: In AWS, S3 is used as the database layer for the microservice deployment. Amazon Simple Storage Service (S3) is object storage that is highly available and scalable in nature. It provides all the features required for data storage, such as security and performance. Amazon S3 is used to store large amounts of data at any scale most common use case for s3 is data lakes, analytics, log archive, etc. (AmazonS3, 2022). S3 is On-Demand. No upfront required. Pay per usage. S3 is a Global service means it is public in nature. In the deployment, S3 is used to store all the images uploaded by the user. But for the security concern, private is chosen for the S3 bucket with EU-west-2 London region for both the cloud platform and SSE-encryption is enabled, which is handled by the cloud provider (PrivatelinkS3, 2022). Similarly, in Google, cloud storage is used as the database layer. Like s3, cloud storage offers a platform to store millions of objects on a large scale (CloudStorage, 2022). In this deployment, cloud storage is used to store the processed image uploaded via the container application.

Application Layer: The microservices backend is deployed in AWS EC2 instances and leverages Amazon Elastic Container Service (ECS) to orchestrate all of the container instances, as shown in Figure 4.2. Amazon Elastic Container Service (ECS) is a container management service that orchestrates the life cycle of a container running as the tasks that support Docker and allows to run applications on a fleet of Amazon EC2 instances(AmazonECS, 2022). ECS is integrated with Amazon Elastic Container Registry (ECR) by using the ECS task execution IAM role, where the updated containerized images are uploaded, and the ECS service automatically pulls the latest images from it. Amazon Elastic Container Registry (Amazon ECR) is an AWS-managed container image registry service that is secure, scalable, and reliable. On behalf of the user, Amazon EC2 instances can access container repositories and images (AmazonECR, 2022). Service and task are registered in the ECS cluster. REPLICA service is the service type inside the ECS cluster that takes care of the container by placing and maintaining the desired number of tasks placed randomly on instances. In order to handle multiple requests, the ECS cluster is integrated with a load balancer that distributes the traffic to the containers.

In Google Cloud, the Google Kubernetes Engine (GoogleGKE, 2022) is provisioned to deploy the Microservice application. Like AWS ECS, GKE comes with two components Control plane and the Node plane. The Control plane is similar to the ECS cluster, which manages and schedules Pods in the Node group. Pods contain one container for the deployment of microservice-based applications. Application code is built using a docker file, and the image is pushed to Artifact Registry, which is the image registry in Google cloud. GKE cluster is created first, and then the Node group. A node group is a group of instances with identical nodes that runs the application workload. Under the GKE cluster, the workload contains a group of identical pods. Since it is the GKE, Pods are deployed using externally managed Kubectl, which is a command-line client.

Front end: A load balancer is used to distribute traffic to the backend containers and is introduced in the front of the architecture. For deployment in AWS, a Network load balancer is used. A Network Load Balancer operates at layer 4 of the OSI model. NLB handles millions of requests. After NLB receives traffic, it will listen to the port of the incoming traffic and then identifies the healthy target in a target group that matches the rule(protocol- TCP and port-3000 of the container) and enable TCP connection (AmazonELB, 2022). A load balancer is integrated with the AWS Autoscaling group, which is capable of scaling backend EC2 instances based on the load. ASG plays a major role in this architecture as it creates an impact on Infrastructure costs. Load balancer continuously performs a health check on the target by having the target group where all the targets are registered.

Similarly, in Google Cloud TCP load balancer(GoogleTCP, 2022) is used, which distributes the traffic to the backend Kubernetes Pods hosted on identical nodes under the single node pool. Both nodes and workloads are under the GKE cluster. Load balancer configuration and distribution of traffic will be covered in detail in the implementation section of the report.

4.2.2 Serverless Architecture

In this section, high-level architecture is designed for Serverless platforms in both cloud providers. In Google, the serverless platform used is Google Cloud Functions, and in AWS, the serverless platform is AWS Lambda.

4.2.2.1 Serverless Architecture in AWS

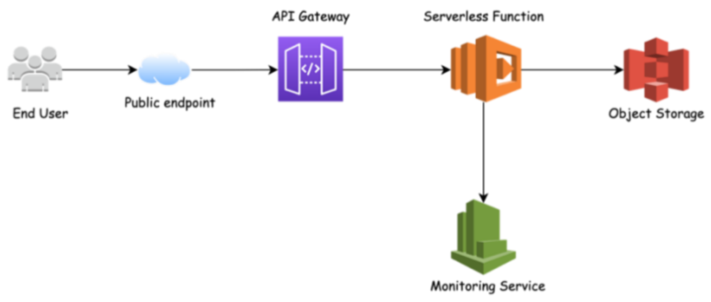


Figure 4.4 Serverless Architecture in AWS

4.2.2.2 Serverless Architecture in Google Cloud

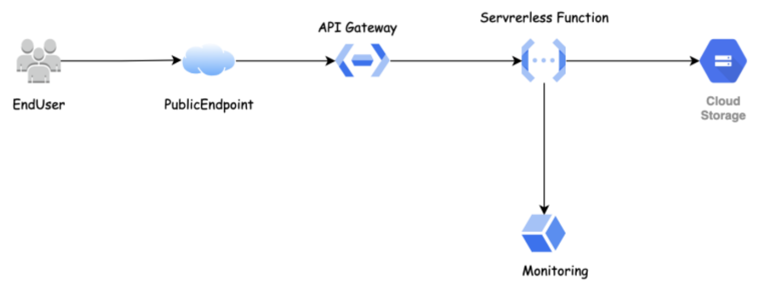


Figure 4.5 Serverless Architecture in Google Cloud

Figure 4.4 and Figure 4.5 depict the high-level architecture to deploy a serverless application in AWS and Google.

Like microservices, an image processing application needs a frontend endpoint to be accessible from the public network. The data should be retrieved in the form of a payload and pass it to the backend application. The front end should be flexible enough to redirect the request based on a path (Path-based routing). Hence API Gateway is used for the cloud platform for a serverless approach.

AWS Lambda is used to host the image processing application. It is a service that allows users, developers, and technology organization to run their application code without the need to set up and manage servers, which is often known as “serverless architecture ” (Sentinalone, 2021). In a nutshell, function as the code pay for the time the function actually gets executed. Also, lambda supports most programming languages. In comparison with microservice architecture, each service in serverless architecture can be implemented as a function which will be then invoked by the frontend layer. Here lambda function running the code is considered as the application layer. Similar to microservices architecture, where one load balancer is used for the backend services, in serverless, one API gateway is implemented for the functions that receive requests from end-users through the Internet. API Gateway is configured with method (POST) and linked to the respective lambda function. POST method, which forwards the traffic to the respective path. Here /upload resource path used to insert the image in S3 or Cloud Storage. It processes the image, then uploads it to the S3 bucket and returns the results to end-users. Every function in Lambda is mapped to a container, which runs in an AWS lambda worker (AWSLAMBDAsecurity, 2022) called a lambda execution environment set up by a cloud provider on a dedicated AWS account to run the application.

In contrast to microservice architecture, it is not necessary to assign specific ports since a specific method, resource path, and integrated target are configured, as shown in Figure 4.6. AWS Lambda is the managed serverless service where the scaling of backend infrastructure will be taken care of by AWS.

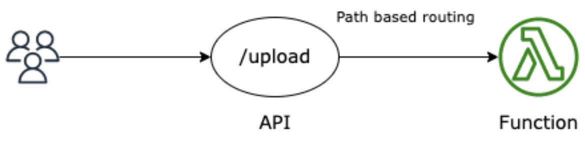


Figure 4.6 Serverless Path-Based Routing

The backend incorporates object storage, where all stateful data is saved. S3 is a fully managed service used to store objects.

Similarly, equivalent to AWS lambda, in Google is Google cloud functions. Google Cloud Functions is a scalable, OnDemand serverless product to help build and connect event-driven services underlying servers and containers are managed by cloud providers (Googlefunction, 2022). For the front end, the Google API gateway is used. The functionality of the API gateway is the same across all the Cloud Platforms as it routes the request to the backend function based on the path. Like S3, cloud storage is used as the backend to store images. The communication between functions and the cloud storage is secured with fine-grained access using an IAM service account in google cloud, whereas, in AWS, communication between services is secured using IAM roles.

Chapter 5

Implementation

In this chapter, the previously defined architecture is implemented with its associated components. The above-discussed architecture is deployed using the two strategies: Microservice, which is a DevOps approach, and the other is Serverless which is a NoOps approach. DevOps deployment is achieved by deploying the microservices on an ECS cluster and GKE cluster and configuring the scaling and other parameters manually. NoOps deployment is achieved by deploying the functions on AWS Lambda and Google Cloud Function. Additionally, two buckets are created S3 and Cloud Storage in AWS and in Google. One for microservices and the other for serverless to upload the image.

5.1 Microservice

To benchmark the application for the comparative study, the application has to deploy as the Microservice. Deployment using Monolithic is straightforward by provisioning a Virtual machine and installing an application. Whereas for the microservices deployment, the application code is packed as the docker container and run in a virtual machine. On top of that, to achieve scalability and fault tolerance. An orchestrator is required to manage the containers that run on virtual machines. In this section, step-by-step implementation of the microservice in AWS & Google cloud can be seen below.

5.1.1 Amazon Web Services

Setting up infrastructure and configuring the basic elements in the AWS are the prerequisites that need to be considered before implementing the system. This configuration includes the following items:

Network Configuration: To set up the container platform for Microservices deployment base network infrastructure is the pre-requisite. For serverless platforms, network setup is not mandatory as the service can run outside the customer network. For setting up a virtual network in AWS, the following services in Table 5.1 are provisioned, and the right IP CIDR is created to allocate to subnets.

Cloud Provider	Network Resources
AWS	VPC, Subnets, Routetable, Internet Gateway, NAT Gateway

Table 5.1 AWS Network Components

Security Configuration: An IAM role is created with the below access policy as mentioned in Appendix D.3 and attached to an instance running the containers. This is required for the EC2 instance to communicate with other services such as S3, ECS cluster, and SSM services. One IAM role is created and attached to all the Ec2 instances spin up during scaling action

Image Configuration: Amazon Machine Image (AMI) has been chosen from the marketplace to provision worker EC2 instances for the ECS cluster. Amazon ECS-optimized AMIs provided by Amazon that is preconfigured with the requirements to run your container workloads on Amazon ECS Linux instances (*Amazon ECS-Optimized AMI - Amazon Elastic Container Service*, n.d.). Listing 5.1 is the command to fetch the latest ECS-optimized image. This image has a pre-installed configuration such as an ECS agent to communicate with the ECS cluster, an SSM agent to communicate with the session manager instance, and other dependency libraries.

```
aws ssm get-parameters --names /aws/service/ecs/optimized-ami/amazon-linux-2/recommended
```

Listing 5.1 Command to Fetch Latest AMI

Image Repository: For microservices deployment, the docker image must be ready beforehand in the image repository to pull the image to run the container before creating an ECS cluster. In AWS, the Elastic Container Registry (ECR) is used as an image repo.

Instance Configuration: The T3 instances feature contains up to a 3.1 GHz Intel Xeon Scalable processor (Skylake or Cascade Lake) with baseline performance and burstable CPU(AmazonEC2types, 2022). After choosing the instance type to include VPC, subnets, IAM role, auto-assign IP address, user-data option, security group, and key pair are configured. Below, Table 5.2 list the instance configuration.

Node Type	vCPUs	Memory (GiB	No of minimum instances	On-Demand Price/hr*
t3.xlarge	4	16	4	$0.1670

Table 5.2 Microservice Instance Configuration in AWS

5.1.1.1 Building the system

In the above section, the prerequisite environment configuration for implementing the Microservice system in AWS is enabled.

Build an Image: Application code is converted into a docker image using a Docker file and pushed to the ECR image repository to provision a container from the image. Image is pushed to ECR using docker push commands. Listing 5.2 shows the Dockerfile used to build an image.

```
FROM node:14

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json
AND package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

EXPOSE 3000
CMD [ "node", "index.js" ]
```

Listing 5.2 Dockerfile for Application Build

ECS Cluster: Once the image is available in the ECR, it is time to create the Cluster for service provisioning. In this deployment, an empty ECS cluster is created. The reason for this is to manage and update cluster resources outside of the cluster from the default setup. The next setup is to add a capacity provider to the ECS cluster. To manage the infrastructure and the tasks in the cluster, ECS capacity providers are used. The capacity provider strategy decides how the tasks are spread across the cluster (ECSCapacityprovider, 2022). AWS Offers two different types of capacity providers EC2 Launch Type and Fargate. Fargate is AWS Managed computer layer where the user doesn’t have access to scale the compute layer. As discussed in the design phase, this study is focused on the EC2 Launch type, which means the containers will run on EC2 instances, which are managed and operated by the user.

Autoscaling Group: Amazon EC2 Auto Scaling helps to ensure that the correct number of Amazon EC2 instances are available to handle the load. A group of EC2 instances forms an Auto Scaling group. In this deployment, a new autoscaling group is created with a minimum of 4 instances with the type mentioned in the instance configuration section and a maximum of 16 instances. Under certain load conditions, there will always be four instances pre-allocated as desired, and minimum instances and running do not exceed 16. When a new instance is launched by an autoscaling group, the instance is configured with a user data script in Listing 5.3 to join the ECS cluster automatically with two availability zones. When an instance is terminated, it removes from the autoscaling group and unjoins the ECS cluster. Autoscaling group using launch template to create a new instance using a base configuration such as AMI, template version latest, key pair, instance type, and security group specified in the launch template. After creating the launch template choose the launch template with its version and choose the instance launch option containing VPC, subnet, and Instance type. Along with advanced option LB, health check of EC2 for 300 (default). If it is healthy adds to the service. Figure 5.1 depicts how the autoscaling group is configured with the minimum, maximum, and desired capacity of EC2 instances below(AmazonEC2AutoScaling, 2022).

```
#!/bin/bash
echo ECS_CLUSTER=imageproc >> /etc/ecs/ecs.config;
```

Listing 5.3 Instance Onboarding to ECS Cluster

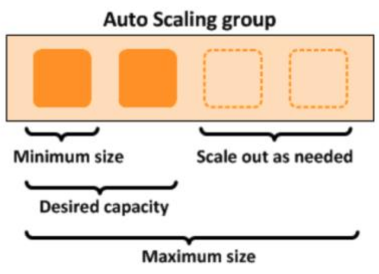
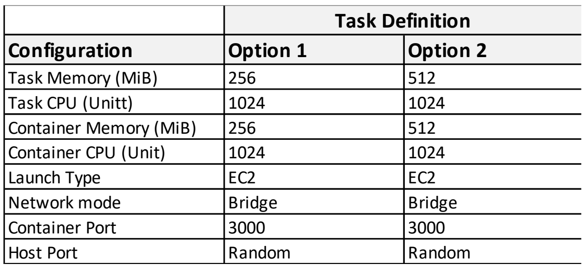


Figure 5.1 Autoscaling Model in AWS

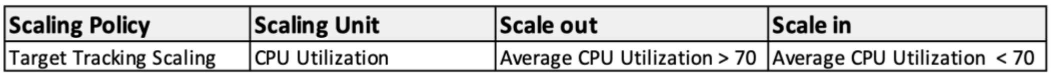
Container Configuration: Task Definition is required in ECS to run docker containers. Docker image is specified in each container within the task definition. And also how much CPU and memory to use with each task or each container within a task definition. In this deployment, two task definitions for two different memory configurations were created, as mentioned in Table 5.3. Versioning can be maintained in the task definition. One task can contain one or more containers. Here, 1 task is assigned to 1 container as the container is exposed to port 3000 with TCP protocol specified in the port mapping with a random host port. ECS task uses the virtual docker network to communicate with each other. ECS task execution IAM role is assigned to the task as it pulls the container image from ECR and sends the logs to Cloud Watch.



	Task Definition	
Configuration	Option 1	Option 2
Task Memory (MiB)	256	512
Task CPU (Unitt)	1024	1024
Container Memory (MiB)	256	512
Container CPU (Unit)	1024	1024
Launch Type	EC2	EC2
Network mode	Bridge	Bridge
Container Port	3000	3000
Host Port	Random	Random

Table 5.3 Container Task Configuration in AWS

Instance Autoscaling Configuration: In the previous step, the empty ECS cluster, capacity provider, task definition for container deployment, and ec2 autoscaling group were added to the cluster. Now it's time to configure Instance autoscaling. This is required for an autoscaling group to respond on how to scale up and scale down the ec2 instances during extreme load scenarios whenever the scaling is required. EC2 Autoscaling group has different scaling policies to configure. In this deployment, "Target Tracking Policy "was chosen as the scaling option. With the metric type "Average CPU Utilization" along with its target value "70 %" (cool-down period 300), Table 5.4 shows the instance scaling policy for microservices.



Scaling Policy	Scaling Unit	Scale out	Scale in
Target Tracking Scaling	CPU Utilization	Average CPU Utilization > 70	Average CPU Utilization < 70

Table 5.4 Instance Scaling Policy

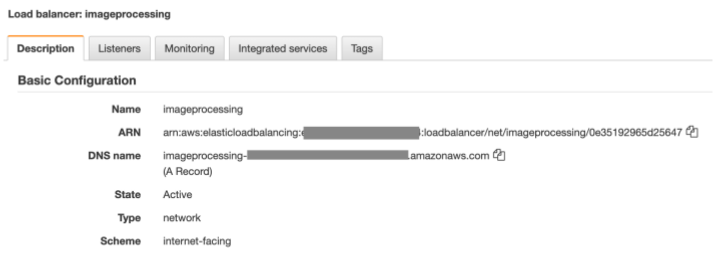
Service Scaling and configuration: Service is responsible for task autoscaling, and the service type used is REPLICA, which places and maintains the desired number of tasks on EC2 instances. The desired count of the task may increase or decrease in the Amazon ECS cluster by service, which has an ECS auto scaler role to scale the task. Firstly, create service under ECS cluster with EC2 launch type, above created task definition, latest version, cluster name all the details are configured. AWS ECS leverages the Application Auto Scaling service to provide this feature (ECSserviceautoscaling, 2022). In this deployment, we used the "Target Tracking Scaling Policy" to increase or decrease the number of tasks based on the target metric value. With this scaling policy, whenever the average CPU utilization for all running tasks goes beyond 70%, then the application autoscaling policy is triggered and places new tasks on the ec2 instance. Similarly, when the CPU utilization is less than 70%, then it will delete the tasks gradually from the instance and free up the space in the EC2 instance. This scaling policy depict in Table 5.5

Scaling Policy	Scaling Unit	Scale out	Scale in
Target Tracking Scaling	CPU Utilization	Average CPU Utilization > 70	Average CPU Utilization < 70

Table 5.5 Container Scaling Policy

Task Placement: For task placement, instances are selected. In this deployment, the task placement template custom is set with type random, which means the tasks are placed randomly across the underlying EC2 instance based on the resource availability.

Load Balancer: After configuring both the horizontal and vertical scaling of tasks now, it's time to expose microservices behind the load balancer. For this purpose, a Network Load balancer is used, which works on Layer 4. The load balancer is created and exposed to the internet with IP address type IPV4 and by placing them into the public subnet. A listener is created by configuring protocol (TCP) and port(80) for a random host. Hence, the load balancer listens to the traffic and passes it to the backend, which is in a dedicated virtual network environment created by the user with two availability zones. Once the load balancer is created, it returns the DNS name, which is used as the endpoint for accessing the Microservice. Below, Figure 5.2 shows the Load balancer created for the Microservice.



Load balancer: imageprocessing

Basic Configuration

Name	imageprocessing
ARN	arn:aws:elasticloadbalancing:[redacted]:loadbalancer/net/imageprocessing/0e35192965d25647
DNS name	imageprocessing-[redacted].amazonaws.com (A Record)
State	Active
Type	network
Scheme	internet-facing

Figure 5.2 Load Balancer in AWS

Service Deployment: The final step of the deployment has been reached. In order to handle multiple requests, the ECS cluster is integrated with the network load balancer (NLB) that distributes the traffic to the containers. Figure 5.3 shows a further detailed overview of the deployed back end.

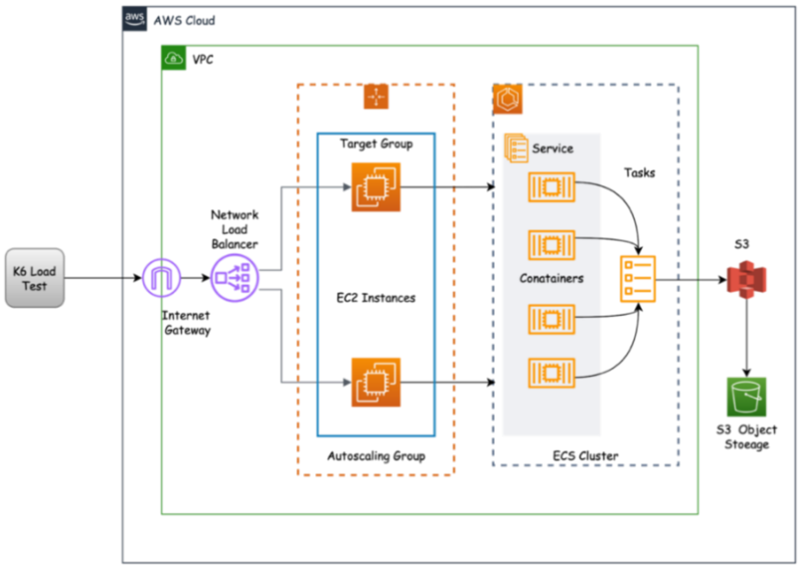


Figure 5.3 Detailed Microservice Architecture in AWS

The target group is now configured to route the traffic to one or registered targets. Targets are nothing but the EC2 instance that runs the application container. In this deployment, the target group is configured to perform a health check on the target instance over the traffic port. Once the target group is available, a new listener rule is created and attached to the load balancer. All the load balancer traffic now listens to the new listener rule, which looks up for traffic over port 80 and forwards the traffic to the target group. As a final step, the service is now deployed on the ECS cluster using the task definition created above and exposed the service to the load balancer over port 3000, which is the container port where the application listens, as shown in Figure 5.4

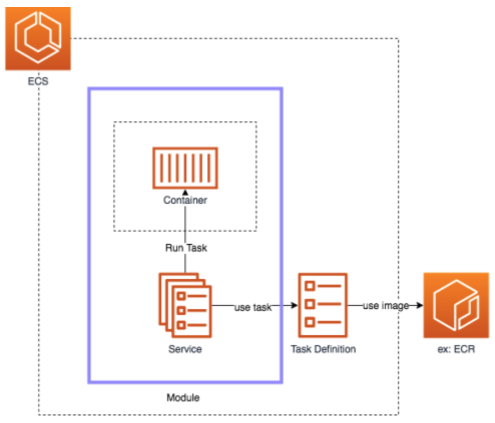


Figure 5.4 Cluster Components

5.1.2 Google Cloud

Like AWS, the deployment strategy using microservice and serverless similar infrastructure setup is made in Google Cloud Platform. The same image processing application code is built using the Docker file.

Network Configuration: Like AWS, a Network setup has to be completed before deploying the Microservice application in GCP. Most of the network services in GCP follow the same pattern as AWS except for Internet Gateway & NAT gateway, which are managed inside Routes (GooglecloudRoute, 2022), as shown in Table 5.6. By default, the Google account comes with a default VPC, Subnets, and Routes. For the purpose of this deployment, a default network setup is used.

Cloud Provider	Network Resources
GCP	VPC, Subnets, Routes, Forwarding rules

Table 5.6 Google Network Components

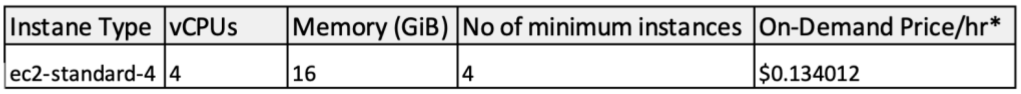
Security Configuration: Once the project is created in google, make sure billing, APIs, and services must be enabled to access API and resources of google cloud. The service account is authentication and authorization, which grant a role as an owner to access the Google APIs and resources.

Image Configuration: To provision the worker VM instances for the GKE cluster. Google provides various OS for node images. Container optimized OS with containerd that has pre-configured with the requirements to run your container workloads on GKE worker VM instances. By default, the above OS is created under the GKE cluster.

Image repository: For microservices deployment, the docker image must be ready beforehand in the repository to pull the image to run the container before creating a GKE cluster. In Google, the Artifact Registry is used as an image repo. It is used to manage and store container images.

Instance Configuration: Here, nodes created under the GKE cluster with machine type e2-standard-4 is a general-purpose machine with 4 VCPUs and 16 GiB memory. It has Skylake processors. The E2 machine offers both AMD EPYC Rome and Intel processors

(GoogleComputeEngine, 2022). It is the second generation with a CPU platform based on availability chosen. Refer to Table 5.7 for instance configuration.



Instane Type	vCPUs	Memory (GiB)	No of minimum instances	On-Demand Price/hr*
ec2-standard-4	4	16	4	$0.134012

Table 5.7 Google Instance Configuration

GKE cluster: In GKE, the cluster consists of a control plane and a group of nodes that contains pods that run containers. In this deployment, Once the image is available in the Artifact Registry, GKE standard cluster is created with four nodes. The control plane is running in one zone with version 1.21.6-gke.1503, and the control plane decides what to run on a group of nodes. This is similar to the ECS cluster with EC2 instance type. Unlike AWS, the autoscaling policy is not configured for nodes. The reason is that the cluster contains a default configuration that uses a cluster auto scaler to add or remove the identical nodes in a single node pool. Nodes and the control plane communicate through Kubernetes APIs. Master contains Kubernetes API server, resource controller, and scheduler (GoogleGKE, 2022).

Node pool configuration: A Group of nodes forms a node pool. A single Node pool is created within the GKE cluster to handle application workloads. Here, standard nodes are chosen so that the user can have flexibility in the configuration and managing of clusters and nodes. In contrast, Autopilot mode is not chosen because it offloads the node management from the users. Similar to AWS, EC2 instances run in 2 availability zones. Likewise, nodes run on two zones within one region to avoid a single point of failure. In this deployment, a minimum of 4 instances and a maximum of 16 instances are set. All the nodes have a Kubelet agent. Unlike AWS EC2 instance configuration, a user data script is not required to communicate with the GKE cluster. Auto-provisioning is not enabled as the GKE cluster automatically launches multiple node pools with different instance types to serve the load, which is not a fair setup for microservice deployment with both cloud platforms.

5.1.2.1 Building the system

The prerequisite environment configuration for implementing microservices deployment in google is explained above. Provisioning infrastructure using the commands in cloud Shell CLI below.

Build an image: In the console, similar to AWS, the same region is chosen as region EU-west-2 London and format Docker is chosen to create a repository in Artifact Registry. Cloud Shell is used to set up the infrastructure and for the deployment of microservices in the GKE cluster. Cloud Shell environment manages resources hosted on google cloud. gcloud CLI, Docker, and Kubectl are pre-installed by Google, where users can use the command-line interface to work with by clicking Activate Cloud Shell button. Hence OAuth authentication can be omitted. Before pushing the image, specify the project location, where the cluster and resources need to run, and specify the compute zone and region to authenticate to the artifact registry. Next step, the Application code is converted into a docker image using the Docker file. During this process, google cloud asks for authentication to make an API call to access Google services. Tag the target image that refers to the source image and pushes it to the destination, and push the image to the artifact registry. Listing 5.4 is the command used to build the image.

```
gcloud auth configure-docker europe-west2-docker.pkg.dev
gcloud config set project robust-resource-12345
gcloud config set compute/zone europe-west2-a
gcloud config set compute/region europe-west2
docker build -t europe-west2-docker.pkg.dev/robust-resource 12345/imageprocessing/imageprocessing:latest .
docker tag europe-west2-docker.pkg.dev/robust-resource-2345/imageprocessing/imageprocessing:latest
docker push europe-west2-docker.pkg.dev/robust-resource-12345/imageprocessing/imageprocessing:latest
```

Listing 5.4 Google Artifactory Authentication

Cluster configuration: In this section, the GKE cluster is created along with four nodes in a single node pool of identical size. During cluster creation, gcloud request credentials are required to make an API call. The next step to interact with the cluster authentication credentials is required. Using kubectl command-line client that fetches the cluster endpoint and auth data. Later, a Kubeconfig entry is generated for the created cluster. Listing 5.5 represents cluster configuration.

```
gcloud container cluster create imagecluster --num-nodes=4 --machine-type e2-standard-4 --cluster-version 1.21.6-
gke.150
gcloud container clusters get-credentials imagecluster
```

Listing 5.5 GKE Create Cluster

Kubernetes Pod Deployment: After the cluster setup, there are two stages defined in the deployment YAML file. The pods and the TCP load balancer are deployed using the deployment YAML file. In order to handle multiple requests, the GKE cluster is integrated with a TCP load balancer which exposes to port 80. LB distributes the traffic to backend nodes by performing a health check of the targets. Load balancer works on TCP protocol and port 80. Similar to AWS, one pod is assigned to one container. Horizontal scaling is performed for nodes based on the load, and pods are controlled by the GKE cluster. In contrast, vertical scaling for pods is defined in the deployment.yml file as well as horizontal scaling is defined. Pods are called replicas, and all the properties of pods are defined in the YAML file (k8poddeployment, 2022). The controller is responsible for creating and managing pods. Both the memory of 256 and 512 MB are configured in the deployment.yml file. Hence vertical autoscaling is performed for pods to benchmark the microservices and serverless technologies. After deployment using kubectl pods are seen in workload under the cluster. Horizontal autoscaling for nodes is managed by the GKE cluster. Horizontal pod auto-scaler is set, using the below Listing 5.6.

```
kubectl apply -f deployment. yaml
kubectl autoscaler deployment demo-gke-test-deployment --min=12 --max=35 --cpu-percent=70
```

Listing 5.6 PoD Deployment

The load balancer forwards the traffic to a container that is exposed to port 3000. TCP load balancer operates on layer 4. After the workload and TCP load balancer are created, the external IP of the load balancer can be exposed to the internet.

5.2 Serverless Deployment

Serverless deployment is of two main components. They are FaaS (Function as a service) and BaaS (Backend as a Service). FaaS is a service that runs small pieces of code known as functions in the cloud. Some of the examples are AWS Lambda and Google cloud functions. They are ephemeral and become active when an event is triggered. The developer uploads the code that is stored as a function in the cloud. Backend as a service allows developers to focus on the client-side application while things like authentication, user management, and storage are the backend implementation is handled by someone else on the remote server. As mentioned above, for this comparison study to benchmark the technologies, application code is already into a docker image and deployed in AWS & Google container platform, and now it’s time to deploy the same piece of code into the Serverless platform. AWS Lambda and Google cloud functions are the services used to deploy the serverless application. Same as Microservices architecture, the Application here is an image processing application that downloads the image, resizes the image, and uploads the image to an s3 and cloud storage.

Function Configuration: Below Table 5.8 shows the configuration parameters set for AWS Lambda functions.

Configuration	Option 1	Option 2
Memory (MB)	256	512
Ephemeral storage (MB)	512	512
Timeout (sec)	30	30
Enhanced Monitoring	enabled	enabled
Unreserved account concurrency	1000	1000
Provisioned concurrency	Nil	Nil

Table 5.8 Function configuration in AWS

Serverless Framework: The deployment of serverless Architecture is carried out using a serverless framework. It is an open-source web framework written using Node.js. It is developed for building applications on AWS Lambda, Google cloud functions, etc.

5.2.1 Serverless Deployment in Amazon Web Services

Building the System

Pre-requisite:

- IAM user or IAM role used to authenticate into AWS account.
- A system with NPM installed to deploy a serverless framework.
- Serverless Framework is installed.

Build Framework: In Serverless Framework, the application is deployed using the *serverless.yml* configuration file. This file describes the programming language and the entire infrastructure required for an application to deploy. serverless.yml is the starting point and heart of the application. The deployment file is split into two pieces:

- Serverless Yaml file with the resources required to build frontend.
- Backend resources in a separate Yaml file.

Front End Resources: AWS API Gateway is used as the front-end for this application. Amazon API Gateway is a fully managed serverless component that allows to create, maintain, monitor, and secure APIs at scale. The file used for frontend deployment consists of two stages.

Stage 1: Configure provider name as AWS, function runtime environment as node.js, and region to deploy as shown in Listing 5.7

```
service: shared-gateway
provider:
  name: aws
  runtime: nodejs12.x
  region: eu-west-2
```

Listing 5.7 Configure Provider for Serverless Deployment

Stage 2: In this stage, the actual resource to deploy. Amazon API gateway is created under the resource section, and API gateway ids are exported in the outputs as shown in Listing 5.8

```
resources:
  Resources:
    SharedGW:
      Type: AWS::ApiGateway::RestApi
      Properties:
        Name: SharedGW
  Outputs:
    apiGatewayRestApiId:
      Value:
        Ref: SharedGW
      Export:
        Name: SharedGW-restApiId
    apiGatewayRestApiRootResourceId:
      Value:
        Fn::GetAtt:
          - SharedGW
          - RootResourceId
      Export:
        Name: SharedGW-rootResourceId
```

Listing 5.8 Provision API Gateway

Once the frontend is defined in the serverless.yml, it is time to configure the backend in a separate file.

Backend Resources: In this section, the whole backend resource required for the application is defined. Similar to the frontend, this is again defined in the serverless.yml file but in a different folder. Please read the appendix for code details. This file contains three stages Stage 1: Provider is configured as same as frontend and additionally exported API Gateway id from the previous serverless.yml file is imported here, and IAM roles for backend communication is provisioned under provider section as shows in Listing 5.9

```
provider:
  name: aws
  runtime: nodejs12.x
  region: eu-west-2
  stackName: imageUploader
  apiGateway:
    restApiId:
      "Fn::ImportValue": SharedGW-restApiId
    restApiRootResourceId:
      "Fn::ImportValue": SharedGW-rootResourceId
  iamRoleStatements:
    - Effect: "Allow"
      Action:
       - "s3:PutObject"
       - "s3:GetObject"
       - "s3:ListBucket"
       - "s3:DeleteObject"
       - "s3:PutObjectAcl"
       - "s3:CreateObject"
       - "s3:ListObjects"
      Resource:
        - "arn:aws:s3:::${self:custom.bucket}"
        - "arn:aws:s3:::${self:custom.bucket}/*"
```

Listing 5.9 IAM Role for Function Integration

Stage 2: In this section, the AWS Lambda function is provisioned with the index.js, and the event trigger is configured as the API Gateway. Post method is configured in the APIs for image upload. An environment variable is passed in this section for the application to consume, as shown in Listing 5.10

```
functions:
  UploadImage:
    handler: uploadImage.handler
    # The `events` block defines how to trigger the uploadImage.handler code
    events:
      - http:
          path: upload
          method: post
          cors: true
    environment:
      Bucket: ${self:custom.bucket}
```

Listing 5.10 Function Deployment in AWS

Stage 3: Finally, the backend database layer is created in this stage as in Listing 5.11. AWS S3 is used as the Storage data layer to store the images. Below is the snip to create an S3 resource as part of the serverless framework. All three layers of AWS services are created using a serverless framework template (ServerlessFramework, 2022).

```
resources:
  Resources:
    StorageBucket:
      Type: "AWS::S3::Bucket"
      Properties:
        BucketName: ${self:custom.bucket}
```

Listing 5.11 Provision Object Storage in AWS

Deploy Framework: Once the resources are defined in the serverless YAML file now, it is time to deploy the resources. To deploy this, prepare a system with npm installed and then use the npm install command to install the serverless framework.

Step 1: Authenticate into AWS account using IAM user or Role.

Step 2: Deploy resources using the serverless deploy command. Following bash script prepared to deploy a resource in sequence as there is a dependency. The below script will create API gateway resource and then deploy backend AWS Lambda function, API Gateway methods, IAM role & Policy, and S3 bucket. Once the script shows in Listing 5.12 runs successfully, resources defined in the Architecture are provisioned in the AWS environment.

```
#!/bin/bash

cd gateway
serverless deploy
sleep 5s

cd ..
cd images
serverless deploy
sleep 5s

aws s3api put-object --bucket oslometx --key destination/ --content-length 0

echo "Press any key to continue"
read
```

Listing 5.12 Deploy Serverless Application in AWS

5.2.2 Serverless Deployment in Google Cloud

Pre-requisite:

- Authentication into the Google account, using Cloud shell.
- A system with NPM installed to deploy a serverless framework.
- Serverless Framework is installed.

Build Framework: Like AWS, Google cloud resources are defined in the serverless YAML file, and in addition, google APIs are used to provide some of the services.

Front End Resources: In this deployment, the API gateway is provisioned using Google APIs. In google, the Open API specification document which routes the calls to the google cloud function securely. Here, API methods are defined. Using google official documentation swagger file is created (GooglecloudAPIGateways, 2022). Please refer to the appendix section for the OpenAPI document. Once the API is defined in the swagger file, the API gateway is created using the below Listing 5.13.

```
# Create an API
gcloud api-gateway apis create $API_ID --project=$PROJECT_ID

# Describe the API to see the details
gcloud api-gateway apis describe $API_ID
#gcloud beta api-gateway apis create $API_ID --project=$PROJECT_ID

# Create an API config
gcloud api-gateway api-configs create $CONFIG_ID \
  --api=$API_ID --openapi-spec=openapi2-functions.yaml \
  --backend-auth-service-account=$PROJECT_NUMBER-
compute@developer.gserviceaccount.com

# Describe the API config to see the details
gcloud api-gateway api-configs describe $CONFIG_ID \
  --api=$API_ID

# Create a gateway with the API config
gcloud api-gateway gateways create $GATEWAY_ID \
  --api=$API_ID --api-config=$CONFIG_ID \
  --location=$REGION

# Describe the gateway to see the details
gcloud api-gateway gateways describe $GATEWAY_ID \
  --location=$REGION
```

Listing 5.13 Provision API gateway in Google cloud

Backend Resources: Like AWS Lambda and S3, Google cloud functions and cloud object storage are the two backend components used to deploy the serverless application, as shown in Listing 5.14. Resources for both services are defined in the serverless.yml is illustrated below. The file is executed using the serverless framework deployment (ServerlessFramework, 2022)

```
provider:
  name: google
  runtime: nodejs14
  region: europe-west2
  project: robust-resource-12345

plugins:
  - serverless-google-cloudfunctions

functions:
  uploadHandler:
    handler: uploadHandler
    # The `events` block defines how to trigger the uploadImage.handler code
    events:
      - http: upload
      |
package:
  include:
    - index.js
  excludeDevDependencies: false
```

Listing 5.14 Provision function in Google cloud

Deploy Framework: Like AWS deployment, all pre-requisites must be met to run the serverless.yml file and google APIs.

Step 1: Authenticate into a Google account using Cloud shell.
Step 2: Navigate to the serverless.yml file and run the serverless deploy command. This will launch all the resources defined in the file. In this context, google cloud function and object storage are provisioned.
Step 3: Final step is to deploy the frontend API gateway solution.
For the ease of deployment, all the APIs to create a gateway are packed into a shell script and executed. Once the bash script is successfully run, the Google API gateway and its methods are created. Refer to Appendix D.3 for the deployment script.

Chapter 6

Experiments and Results

This section consists of the experiments performed and the evaluation of results performed against the load test. In this thesis, all the experiments are done to compare the quantitative metrics such as Performance (Response time, Throughput), Cost, and qualitative metrics such as Security, Scalability, Developer experience, and Controllability.

6.1 Infrastructure Scaling for Evaluation

To benchmark the metrics generated from both cloud platform, a certain amount of load for different scenarios is applied to the deployed application. The following sections provide more detail about the infrastructure scaling configuration setup before applying the load test on both platforms.

6.1.1 Microservice

For the microservice architecture, AWS ECS is used to deploy and orchestrate containers. Two different task definition is created in the ECS cluster with the below-mentioned configuration for each container within the task. The potential reason for allocating more CPUs with less memory is because the image processing application used in this thesis is CPU intensive as it requires more computation power.

- One vCPU core (equal to 1024 CPU units) and 256 MB memory
- One vCPU core (equal to 1024 CPU units) and 512 MB memory

For maintaining the desired state of the containers, an automatic task scaling policy is enabled under the ECS service. AWS provides a target tracking scaling policy chosen for scaling the task. The average CPU utilization metric is chosen for scaling the task, and the target value is set to 70%. The scale-in and scale-out cooldown periods are set to 60 seconds to prevent over-provisioning. When the average CPU utilization of the deployed service goes beyond 70%, service auto-scaling triggers and adds another task to the service. Conversely, when the average CPU utilization of the service drops below the target utilization for a sustained period of time, a scale-in alarm triggers, which then removes the tasks gradually and reaches the desired count (ECSserviceautoscaling, 2022). On the other side, at the compute level AWS auto-scaling group plays a major role as it is responsible for scaling underlying instances within the cluster. Like task scaling, a target tracking scaling policy is enabled here. When an average CPU utilization of all the instances goes beyond 70% for a consecutive 1 minute, then one or more instances scales to balance the load, which keeps the average CPU utilization close to 70%. If there is no load, ECS gradually de-provisioned the instances from the cluster. This comes with a default health check grace period of 300 seconds and a cool-down period of 60 seconds.

6.1.2 Serverless

Unlike Microservices, a Serverless application doesn't require a scaling policy to configure scales backend managed by the cloud provider. Serverless Function allows to set only the memory required to execute the function, and AWS proportionally allocates CPU to the subsequent memory (Sentia, 2020), as mentioned in Table 6.1. Provisioned concurrency is not enabled for the Lambda function due to the high-cost factor. However, AWS set a concurrency quota of 1000 for each user account as a soft limitation.

AWS	
Memory	vCPUs
128 - 3008 MB	2
3009 - 5307 MB	3
5308 - 7076 MB	4
7077 - 8845 MB	5
8846+ MB	6

Table 6.1 Memory-CPU Allocation in AWS Lambda

Similarly, no configuration is required for scaling in the Google Cloud function, and the function is configured and tested against 256 & 512 MB of memory. Min instance is not configured due to the high-cost factor. The Google Cloud allocates proportional CPU for memory configured to function(GoogleCloudFunctionPricing, 2022). Table 6.2 list the CPU allocation for different memory configuration.

Google Cloud	
Memory	vCPUs
128MB	.0833
256MB	.1666
512MB	.3333
1024MB	.5833
2048MB	1
4096MB	2
8192MB	2

Table 6.2 Memory-CPU Allocation in Google Functions

6.2 Infrastructure setup for load test

In order to generate the subsequent amount of load to evaluate the application performance, a load test tool is required to simulate the load and applied against the application endpoints. For this evaluation purpose, k6 is considered. K6 is an open-source load testing tool that makes performance testing easy (K6, 2022). K6 is generally used for common use cases such as Load tests, Performance tests. For installing k6, an open-source docker image is downloaded from the Docker Hub and run as the docker container within an AWS EC2 instance. The idea of running K6 in AWS EC2 as a docker container is because it could use container scaling capability when it needs more resources during the load test. Each load test execution in k6 is defined in the form of JavaScript that is supported by K6, which is explained in the next section. Below, Figure 6.1 depicts the load test architecture executed and the deployment of k6 using the docker file with some modifications based on the requirement (XK6-output-influxdb, 2020) is mentioned in the reference section 6.1

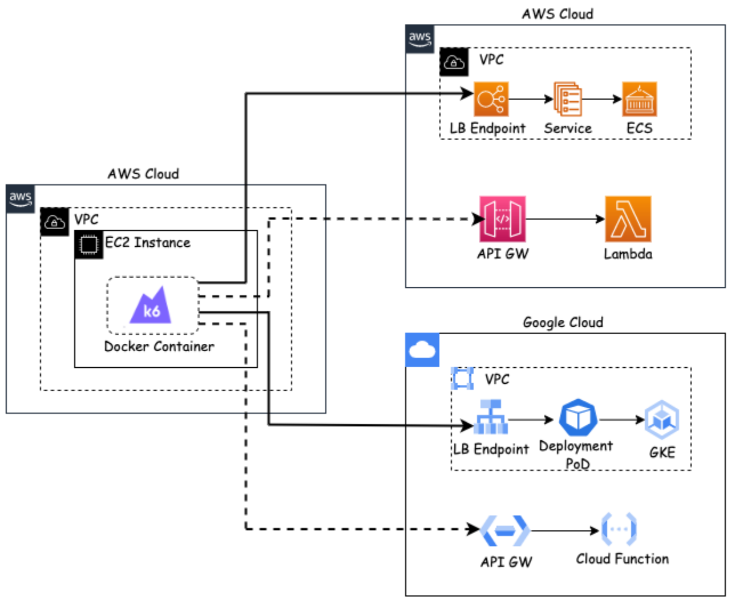


Figure 6.1 Load Test Architecture

K6 generates multiple metrics as the result of the load test. All these metrics generated in K6 need persistent storage to monitor and visualize for evaluation. To achieve this, continuous monitoring pipeline, as mentioned in Figure 6.2, is introduced to capture the metrics and send them to the target system to visualize. Influx DB, an open-source time-series database, is used as the persistent storage that captures the metrics generated by K6. Grafana is then used to create graphs for better visualization. Grafana is open-source analytics and interactive visualization web application which provides charts and graphs for the metrics from the supported data sources.

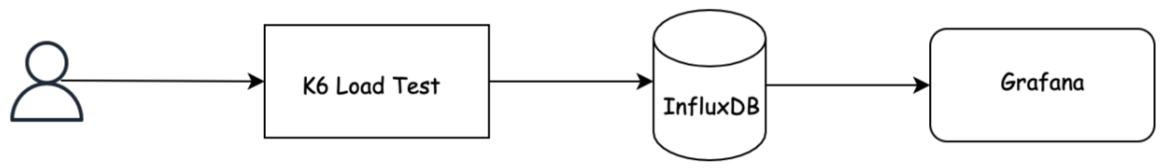


Figure 6.2 Monitoring Pipeline

Like K6, Influx DB and Grafana are deployed as the docker container in the same EC2 instance where K6 is deployed. Both tools are packed and deployed using docker-compose. Please refer to Appendix A. Influx, and Grafana are mounted with the local volume of the EC2 instance to store data persistently. All the metrics generated by K6 are sent to Influx DB using the DB endpoint, which is exposed over port 8086. Once the data is in influx DB, Grafana queries the data using the influx DB data source. Grafana uses Flux query to pull metrics and then create a dashboard in Grafana. In addition to the above monitoring tools, cloud-native monitoring services are utilized to retrieve the backend performance metrics such as memory utilization. AWS CloudWatch is used to retrieve the memory metrics, and in Google, monitoring explorer is used. CloudWatch container and lambda insights are enabled for deeper insights.

6.3 Load Pattern

This section covers in detail the load pattern defined to evaluate performance metrics such as response time, throughput, and associated cost against various scenarios. Each scenario is a simulation of the general traffic pattern seen across. k6 generates different patterns of the user workload to the deployed application. k6 uses a script for running the tests where the target endpoint is specified in the script file. K6 uses virtual users to execute the test. Virtual Users (VUs) are the entities that generate a request to the target. They run concurrently and parallel and will continuously iterate through the request until the test ends. In this thesis evaluation, four scenarios have been identified, with each test run duration of 15 minutes. Each scenario is repeatedly tested five times to calculate the average number. Between each test, 30 minutes of pause is made to make sure that the testing infrastructure reaches a normal state. Below are the four different types of load patterns that were executed.

6.3.1 Constant request rate

The constant number of iterations is executed for a specified period of time so that it cannot be affected by the performance of the system under test. Here, 50 iterations per second are executed, allowing K6 to schedule dynamically to 200 virtual users. The total scenario duration is 15 minutes. Pre-allocated virtual users are 2, and the maximum number of virtual users is up to 200, as depicted in Listing 6.1, and Figure 6.3 is the visualization of the applied load.

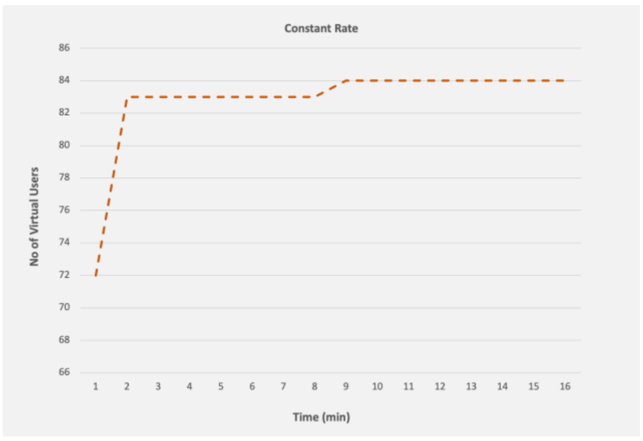
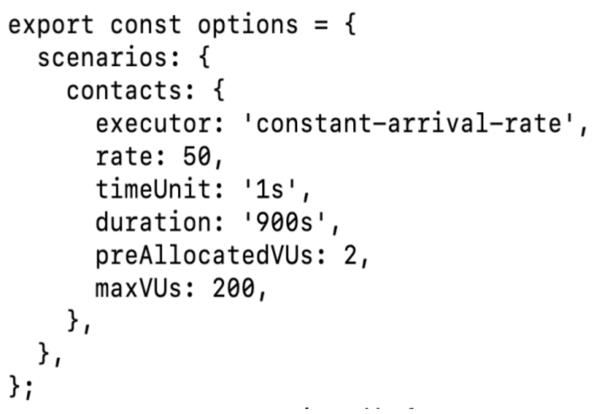


Figure 6.3 Load Pattern – Constant



```
export const options = {
  scenarios: {
    contacts: {
      executor: 'constant-arrival-rate',
      rate: 50,
      timeUnit: '1s',
      duration: '900s',
      preAllocatedVUs: 2,
      maxVUs: 200,
    },
  },
};
```

Listing 6.1 Constant Load Simulation

6.3.2 Random Load Test

This load test is concerned with assessing system performance. To ensure the application performs satisfactorily when the number of users randomly increases or decreases to access the system at the same time. To meet the performance goal, success criteria are set that the load test should have an average response time of fewer than 1000 milliseconds (avg< 1000ms). The maximum duration for the test is 15 minutes. Load ramp up and down randomly up to 150 concurrent virtual users. Listing 6.2 is the script to simulate the load, and Figure 6.4 is the load pattern.

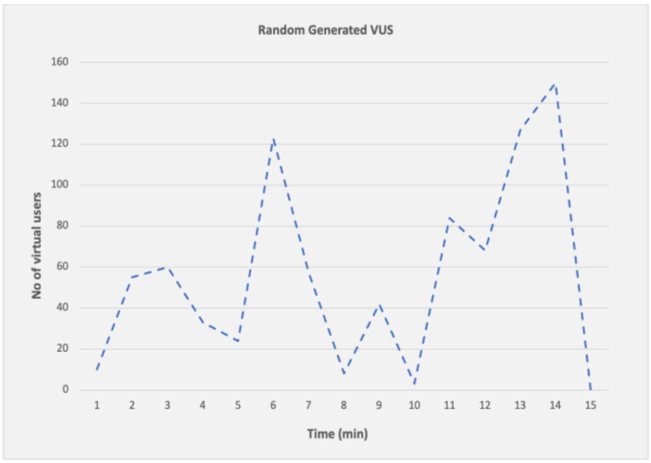


Figure 6.4 Load Pattern – *Random*

```
export const options = {
 stages: [
          { target: 10, duration: '1m' },
          { target: 55, duration: '1m' },
          { target: 60, duration: '1m' },
          { target: 33, duration: '1m' },
          { target: 24, duration: '1m' },
          { target: 123, duration: '1m' },
          { target: 57, duration: '1m' },
          { target: 8, duration: '1m' },
          { target: 42, duration: '1m' },
          { target: 3, duration: '1m' },
          { target: 84, duration: '1m' },
          { target: 68, duration: '1m' },
          { target: 127, duration: '1m' },
          { target: 150, duration: '1m' },
          { target: 0, duration: '1m' },
 ],
 thresholds: {
   'http_req_duration': ['avg < 1000'],
 },
};
```

Listing 6.2 Random *Load Simulation*

6.3.3 Stress Incremental load

This test is concerned with the availability, stability, and reliability of the system. Incremental test, which simulates the stress test used to determine the limit of the system and test beyond its breaking point. The load increases gradually up to 500 concurrent virtual users. The test plan was executed for 15 minutes. In k6, configuring the options object increases the load by 100 users every 1 minute and stays for 2 minutes. The recovery stage is set to decide whether the system can serve requests as load decreases. Listing 6.3 is the script to simulate the load, and Figure 6.5 is the visualization for the load pattern.

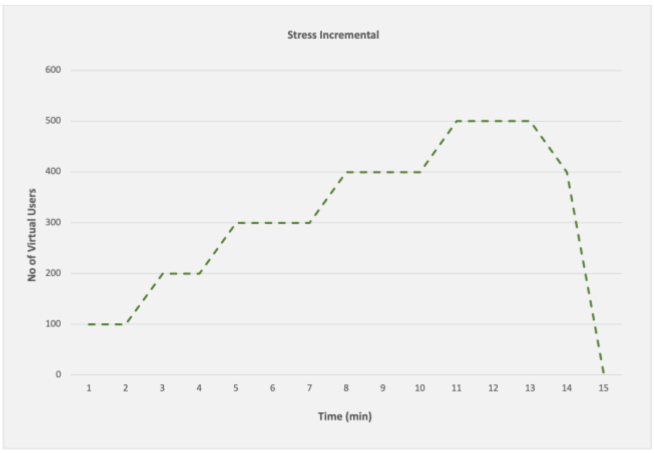


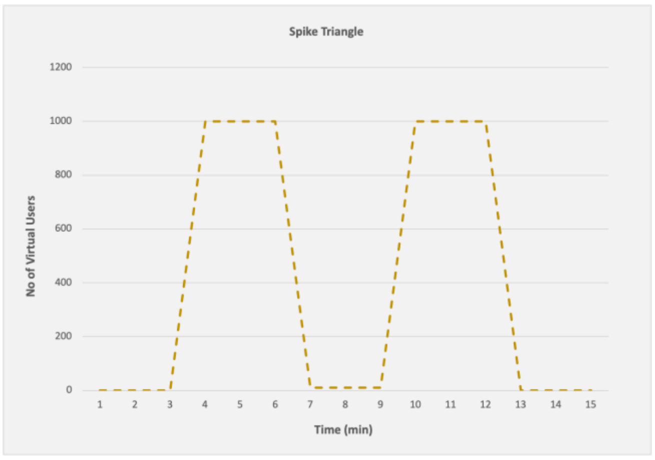
Figure 6.5 Load Pattern – *Stress Incremental*

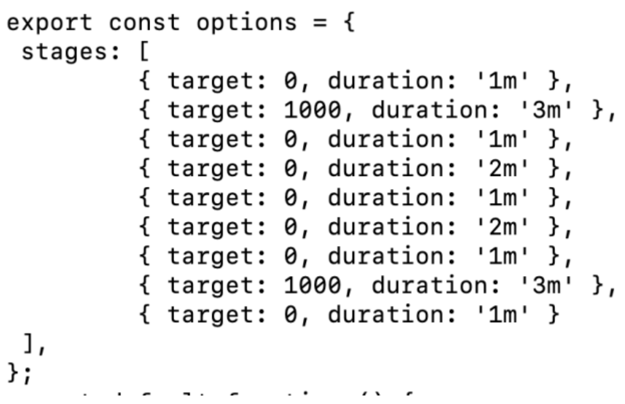
```
export const options = {
 stages: [
          { target: 100, duration: '1m' },
          { target: 100, duration: '1m' },
          { target: 200, duration: '1m' },
          { target: 200, duration: '1m' },
          { target: 300, duration: '1m' },
          { target: 300, duration: '2m' },
          { target: 400, duration: '1m' },
          { target: 400, duration: '2m' },
          { target: 500, duration: '1m' },
          { target: 500, duration: '2m' },
          { target: 400, duration: '1m' },
          { target: 0, duration: '1m' }
 ],
};
```

Listing 6.3 Stress load *Simulation*

6.3.4 Spike Load Test

This test immediately overwhelms the system. Unlike incremental tests, it does not gradually increase the load. Instead, it simulates a sudden spike in traffic. This helps to decide how the system behaves under heavy spike load. This is used to examine the latency of an application when the traffic is irregular. This pattern's maximum duration is 15 minutes. In K6, configuring the options object for the first 3 minutes, no load is specified. After that, the load surges to 1000 concurrent virtual users and ramps down to 10 for 3 minutes, and again replicates for another 7 minutes (K6, 2022), as depicted in Listing 6.4 and Figure 6.6





```
export const options = {
 stages: [
         { target: 0, duration: '1m' },
         { target: 1000, duration: '3m' },
         { target: 0, duration: '1m' },
         { target: 0, duration: '2m' },
         { target: 0, duration: '1m' },
         { target: 0, duration: '2m' },
         { target: 0, duration: '1m' },
         { target: 1000, duration: '3m' },
         { target: 0, duration: '1m' }
 ],
};
```

Figure 6.6 Load Pattern - Spike

Listing 6.4 Spike Load Simulation

To execute the load test for the above scenarios, the below configuration of instance is used to run the load test. To avoid throttling instances with better configuration as mentioned in the below table 6.3.

Instance Type	t3.medium – 2 vCPU , 4 GB Memory
Region	London

Table 6.3 Load Test Instance Configuration

6.4 Results

In this section, data gathered from the experiments were analyzed. Each load test results are evaluated with a separate subsection that covers response time and throughput at first and then followed by memory as part of quantitative metrics. In the end, evaluation on qualitative metrics such as security, development experience, controllability of an application running as a microservices, and serverless.

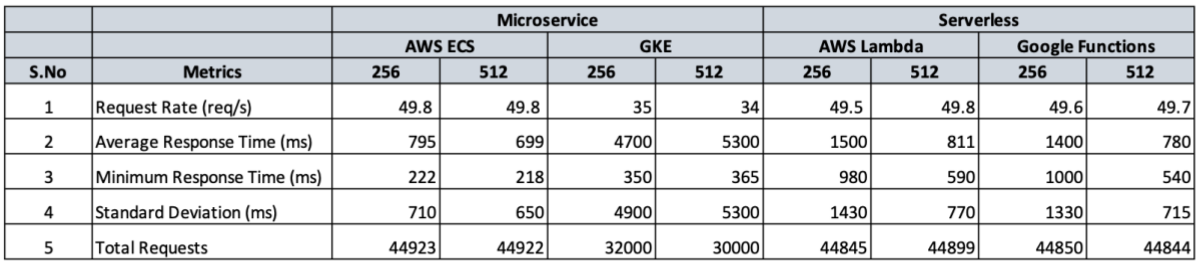
6.4.1 Performance

This section evaluates results against the quantitative metrics. The key metric considered here is response time and throughput. Response time is the amount of time taken by an application to process the request and return the response to the user. It is a critical factor in an application lifecycle as it decides the usability of the application. On each graph representing the response time, the x-axis denotes the duration of the test in minutes, and Y-axis denotes the response time in milliseconds. Each load test scenario is analyzed and compared in the following sections. These performance tests benchmarked microservices and serverless between two cloud providers. The tested serverless platforms were AWS Lambda and Google Cloud Functions, and the tested Microservices platform was AWS Elastic container service and Google Kubernetes Engine.

Constant request rate

Response Time

The first scenario executed against the performance test is the constant request rate. This test aims to validate whether the system is processing a fixed number of iterations with better performance. Table 6.4 displays the comparison results from the constant request rate load tests.



		Microservice				Serverless			
		AWS ECS		GKE		AWS Lambda		Google Functions	
S.No	Metrics	256	512	256	512	256	512	256	512
1	Request Rate (req/s)	49.8	49.8	35	34	49.5	49.8	49.6	49.7
2	Average Response Time (ms)	795	699	4700	5300	1500	811	1400	780
3	Minimum Response Time (ms)	222	218	350	365	980	590	1000	540
4	Standard Deviation (ms)	710	650	4900	5300	1430	770	1330	715
5	Total Requests	44923	44922	32000	30000	44845	44899	44850	44844

Table 6.4 Constant Load Metrics

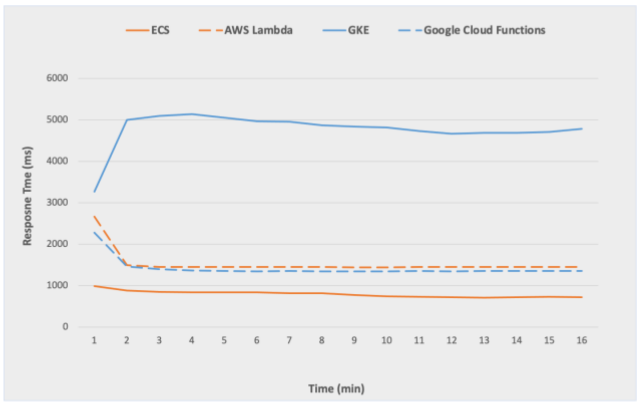


Figure 6.7 Constant - Response time [256 MB]

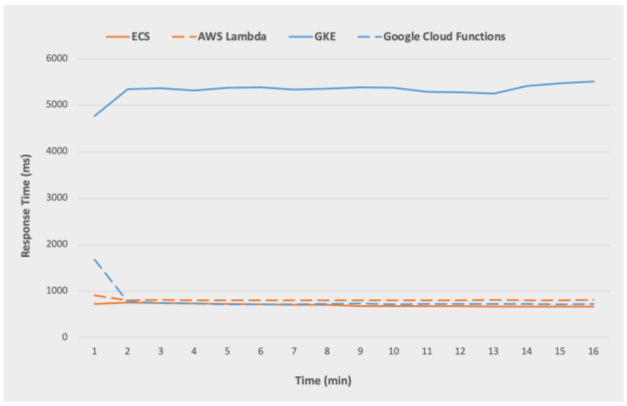


Figure 6.8 Constant - Response time [512 MB]

According to the results, both AWS Lambda and Google cloud function satisfy the requirement by handling approximately 50 req/sec, which is the benchmark set to evaluate. Also, the average response time on both functions looks similar, which is around 1450 milliseconds. Though serverless satisfies the requirement, it is not ideal to declare it as the winner of the test. However, microservice deployed in AWS Elastic Container Services outperform compared to serverless with a better average response time. ECS has provided consistent performance with stable response time throughout the test. However, GKE is not up to the mark as it provides a high response time and only serves 36 Req/Sec, which is not satisfying the requirement. It is not fair to compare AWS ECS & GKE as it is hosted by different cloud providers and the results are not comparable.

Since serverless has a slightly lower response time than ECS, a further step is executed to make the test more granular by increasing the memory configuration in both serverless and microservice. With both 256 and 512 MB memory assigned for containers in the Microservices platform, AWS ECS seems to perform better with respect to average, min response time, and the standard deviation is kept relatively low. It is noticed that ECS has no cold start. This is due to the minimum base infrastructure prebuilt for ECS to run. The other reason for providing a stable response throughout the test is due to the minimum task run on the ECS cluster. It took min 15, and a max of 17 tasks to run the whole test with one instance scale, which means there is not much scaling happened and due to which there is no delay in the latency. From this, it is evident that if the Microservice platform has minimum infrastructure for a predefined constant rate requirement, then it offers better performance on response time. This again adds to the minimum base infrastructure cost.

Figure 6.7 and 6.8 shows the average response time during the whole test run. Except for GKE, both platforms have quite constant average latency without any fluctuations. AWS Lambda has faced a slight cold-start compared to Google Cloud Functions; this is due to both being different cloud providers with diff hardware in the backend. But on average, both have more or less the same latency and still satisfy the requirement during the whole test. One thing to notice here in serverless is when the memory was 256; the response was more than 1000ms in both AWS & Google. Whereas when the function memory is set to 512 MB, the response time drastically comes down and provides much better performance. This is due to the CPU being bound to memory in serverless if the memory is increased or decreased, then it has an impact on the CPU performance of the function. In the end, a microservice platform backed by AWS ECS could be a better choice for the constant workload.

In the case of GKE, it initially started with a low response time, but with the increase in the number of requests, the scaling of replicas on nodes and the requests started to take a longer time to respond. This points towards the load balancing, Pod scheduling, and traffic re-distribution problem in the Kubernetes. On the other hand, in serverless, both deployments suffer from the cold start problem and hence have longer response times in the beginning. However, both show a constant request response time even with the increase in the number of requests for the rest of the test duration in both the workload patterns. This indicates that the serverless is more agile in terms of scalability and can provide a constant response time baring the initial cold starts. Furthermore, with the increase in memory serverless function was able to serve a greater number of requests than any other deployment strategy for this compute-intensive function. Therefore, Serverless deployment can be used for applications requiring constant request latency. Though the Microservices backed by AWS could also provide a constant request latency but finding the optimal scaling and configuration parameters is difficult and requires a prior load testing of the application.

Memory Use

Figures 6.9 and 6.10 depict the memory utilization of microservice and serverless during constant workload.

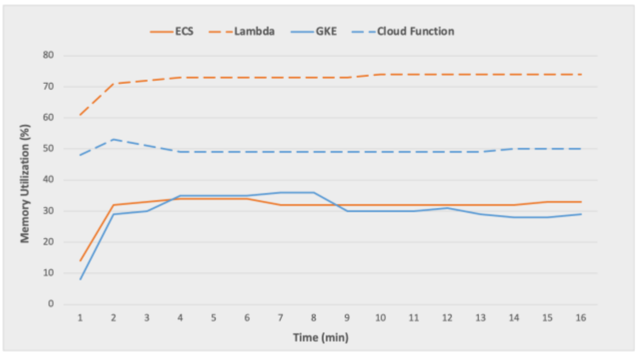


Figure 6.9 Constant - Memory Utilization [256 MB]

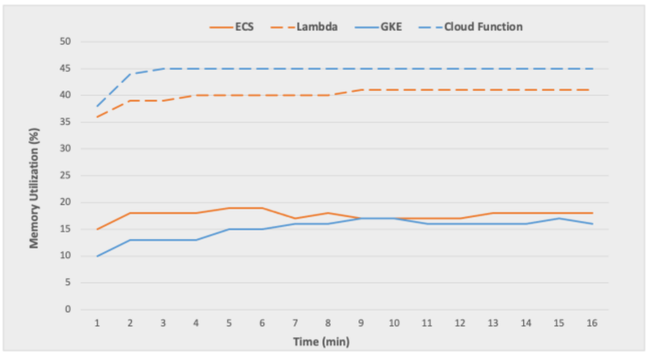


Figure 6.10 Constant - Memory Utilization [512 MB]

At first sight, it is evident that the memory usage is stable on both platforms across the whole test. Compared to serverless, microservices occupy less memory which is 15 to 30% of the overall memory limit. In contrast, Lambda and Cloud Function occupy 40 to 70% of overall memory when the limit is 256 MB and 512 MB. One potential reason for this is for each lambda or cloud function invocation, a dedicated run environment is provided with 256 or 512 MB, and the consumption of memory is only by the specific request. Whereas in Microservices, the container is shared across multiple requests, and hence the memory utilization is average for multiple containers. Microservices consume less memory to serve the request, whereas in serverless, one request is served by one execution environment at a time. The backend mechanism is totally different between the two techniques. Therefore, it cannot be comparable.

Random Load Test

Response Time

The second scenario is random with 150 concurrent virtual users, and the duration is set to 15 minutes. Thus, the performance threshold is given (AVG< 1000ms) for both 256 and 512 MB memory. Table 6.5 depicts the metrics of random test.

		Microservice				Serverless			
		AWS ECS		GKE		AWS Lambda		Google Functions	
S.No	**Metrics**	**256**	**512**	**256**	**512**	**256**	**512**	**256**	**512**
1	Request Rate (req/s)	32	32	18.5	18.5	22	31	23.5	32.5
2	Average Response Time (ms)	800	725	2000	2020	1480	786	1350	738
3	Minimum Response Time (ms)	225	214	365	372	970	585	99	548
4	Standard Deviation (ms)	550	530	1730	1780	1450	775	1350	722
5	Total Requests	29800	30000	16800	16800	20400	28100	21300	29500

Table 6.5 Random Load Metrics

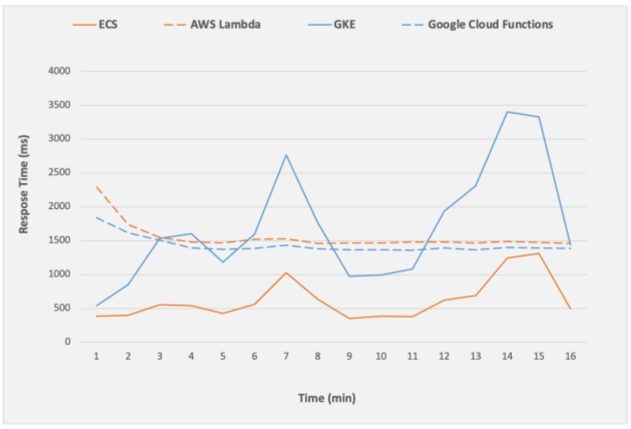


Figure 6.11 Random - Response Time [256 MB]

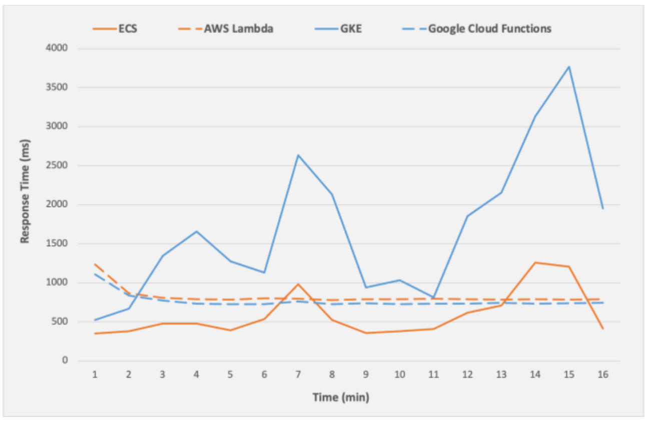


Figure 6.12 Random - Response Time [512 MB]

Figure 6.11 and 6.12 shows the average response time during the whole test run. For 256 MB memory, serverless backed by AWS Lambda and Google Cloud Function has an average response time of more than 1000ms. Both the serverless function does not satisfy the performance goal which is set for them. Some thresholds failed with no request error. The requested rate for AWS Lambda is 22 req/sec, and Google Cloud Function is 23 req/sec, which is almost the same with slight differences. However, GCF performs slightly better with throughput compared to AWS Lambda. It is not fair to compare serverless platforms. The actual comparison between microservices and serverless. For 512 MB memory, the average response time is in milliseconds and meet the requirement with more request served. This proves that as the memory increases for serverless provides a better response time and best throughput. A cold start problem occurs for 1 minute in AWS Lambda and GCF. For 256 MB memory allocation, AWS Lambda suffers from a more cold start. Compared to AWS Lambda, GCF has a relatively low cold start. In 512, both serverless functions start with the same cold start problem. After 1 minute, it becomes stable.

In microservice deployment, both 256 and 512 MB memory in AWS ECS has low latency compared to both serverless deployments with 256 MB memory. AWS ECS response time and throughput seem similar to the serverless functions with 512 MB memory. Moreover, it satisfies the performance goal for average response time must be less than 1000 ms. In microservice deployment, increasing memory does not make any difference as this application is purely CPU-intensive. The instances are pre-allocated with 4 and 15 tasks prior to satisfy the performance threshold and to avoid request time out during scaling. There is fluctuation in response time depending on the load increases and decreases randomly in both DevOps platforms. The reason is that pre-allocated instances and tasks serve the whole incoming request by distributing it in AWS ECS. Similar to GKE, nodes and pods are fixed. Whereas in serverless deployment, cloud providers allocate proportional CPU to memory which is configured for the function. This is evident that an increase in memory plays a major role in serverless platforms. Though it is a random pattern with variation in load, the response time served with stability due to each request is served by a dedicated instance at the backend, which is not visible to users.

In GKE, both the memory has high latency and low throughput of 18.5 req/sec compared to both serverless functions. The performance threshold which is set for it does not meet the requirement for both memory configurations. The reason behind this is the scheduling of pods and launching of new nodes by the GKE cluster consumes time resulting in the minimum request served with high latency and poor throughput.

Taking into consideration, an increase in memory for serverless deployment results in better response time and meets the requirement. Thus, the application behaves satisfactorily under many users' access at the same time. This demonstrates that in comparison between two deployment strategies using microservice and serverless, the winner for this random scenario is AWS ECS with both the memory configuration with a fixed number of instances to meet the performance goal.

Memory Use

Figure 6.13 and 6.14 depicts the memory utilization during the Random load test.

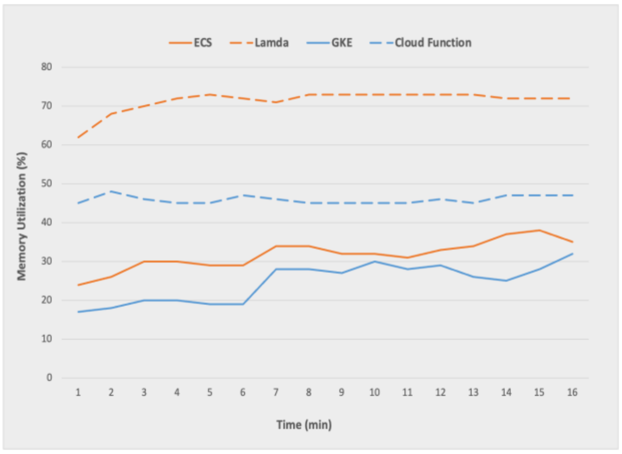


Figure 6.13 Random - Memory Utilization [256 MB]

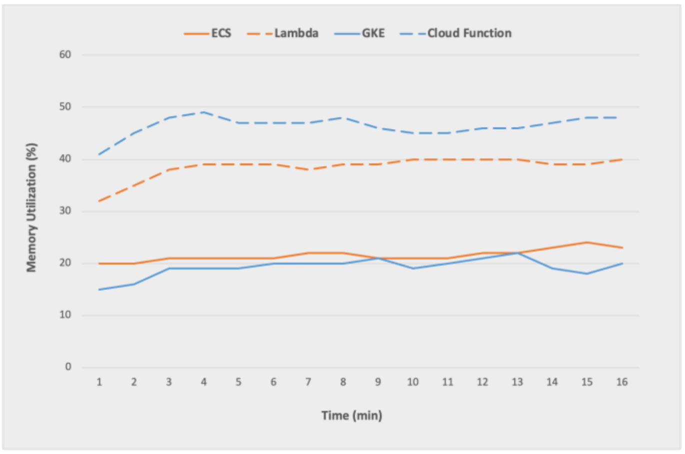


Figure 6.14 Random - Memory Utilization [512 MB]

The memory results of this test are more similar to the previous constant rate. Usage of memory in microservice is less compared to serverless. The reason for this behaviour depends on how Microservice and serverless architecture patterns are defined by the cloud provider. Though serverless utilizes more memory than its limit, it doesn't have any impact on the cost. The more memory it has, the better performance because Cloud providers allocate CPU power with respect to the memory allocated. AWS or Google don't charge based on memory utilization. Rather, they charge based on the execution and amount of memory allocated. Memory utilizes to resize the image is 210 MB for both 256 and 512 MB in serverless.

Stress Load Test

Response Time

The incremental load test was performed for 15 minutes. This test is used to find out the limit of the system. During the test, the load gradually increases up to 500 concurrent users. The ramp down stage is defined to check how the system behaves as the load decreases. Each user simulates multiple requests, and no request is fixed in this test. Table 6.6 depicts the stress metrics and Figure 6.15 and 6.16 shows the response time.

		Microservice				Serverless			
		AWS ECS		GKE		AWS Lambda		Google Functions	
S.No	Metrics	256	512	256	512	256	512	256	512
1	Request Rate (req/s)	60	62	39	42	124	171	125	172
2	Average Response Time (ms)	4070	3910	6820	6270	1460	784.63	1450	773.87
3	Minimum Response Time (ms)	223.1	221.6	387.58	351.4	765.93	603.55	972.01	527.05
4	Standard Deviation (ms)	4100	3950	6300	6800	1430	778	1395	755
5	Total Requests	54512	56301	35438	37887	112142	154769	112691	155732

Table 6.6 Stress Incremental Metrics

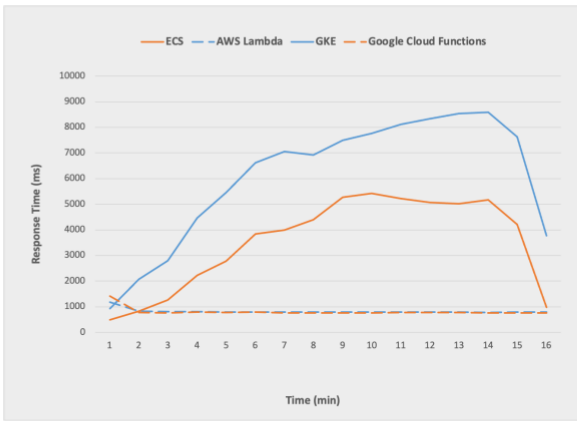


Figure 6.15 Stress - Response Time [256 MB]

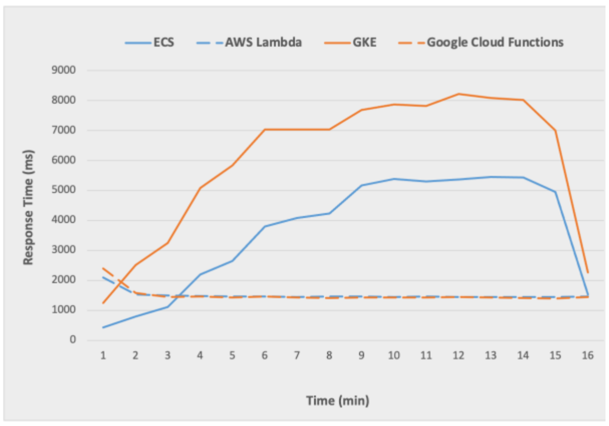


Figure 6.16 Stress - Response Time [512 MB]

According to the results, both the microservices platform has poor performance while handling the incremental load. For 256 MB memory settings, the average response time for both microservices backed AWS ECS and GKE is 4070ms and 6820ms. Take a further step to test with 512 MB to check whether the microservices platform delivers better performance or not. But the outcome shows that both memory settings provide similar response time and request rate with a slight difference. The reason for high latency in AWS ECS and GKE because both serve the request with four nodes with 15 tasks. When the load increases gradually, nodes start scaling to balance the load. Launching and scheduling the pods or tasks in the instance takes time. This adds delay, and this leads to an increase in response time. Each container serves four req/sec as load increases, and no additional space for more requests to serve. The AWS ECS cluster scales up to 9 instances and 28 tasks for both 512 MB and 256 MB memory settings. In GKE, the cluster scales up to 12 nodes and 25 pods to balance the request with 36 errors from the total request served. The error throws during scaling results in instability.

For t3.xlarge with 4 VCPU and 16 GB memory, the maximum capacity of the system in terms of users are 250 VUs with 3861 requests served. Here, the VUs simulate two requests in this test. The request is not fixed for all the concurrent users. Some generate one request, or some users with no request. The autoscaling is enabled as the load increases, launches the instance and balances the load with some request fail due to time out, and serves the request in newly launched instances without any manual intervention. Serverless backed AWS lambda and Google Cloud Function have the best throughput and average response time. In 256 memory allocation, both the serverless function has more or less similar throughput with slight variation. AWS Lambda has 124 req/s and GCF has 125 req/s. Even though both the function is hosted on different cloud provider uses different backend resources, which is not visible to customers. Both deliver closer results in throughput and response time. The total request is served more in both serverless platforms. Similarly, for 512 MB memory, both the functions have 171 req/sec and 172 req/sec request rate, and the average response time in milliseconds are 784.63ms 773.87ms. Figures 6.20 and 6.21 are noticeable that AWS Lambda and Google Cloud Function have cold start problems for 1 minute. Though it has a cold start problem, functions scale up without introducing additional delay and handle 500 concurrent users by serving 155732 requests very well. There is no request error time that occurs during the test, which shows consistent performance. The reason is that at the backend, serverless resources scale instantly. The dedicated runtime environment with allocated memory and CPU assigned by cloud providers serves a single request. As the request hit's function, multiple instances scales to serve multiple requests. Thus, the serverless function has the scaling agility of the Microservices platform.

Microservice platforms are far behind in the results compared to both serverless platforms. AWS Lambda and Google Cloud Function are suitable to handle load which gradually increases, and under heavy load, delivery availability and stability throughout the test with a cold start at the beginning. On the other hand, AWS ECS and GKE struggle to handle the load as it increases. This reason is microservices deployment starts to scale automatically only after the instance reaches the defined criteria for 1 minute with a setting like CPU and Linux instances take time to launch and schedule containers. It took 60-90 seconds. However, this can be reduced by pre-allocating the instances that provide low latency. Therefore, results in increased cost. The winner of this load pattern is serverless deployment with scaling agility.

Memory Use

Figure 6.17 and Figure 6.18 shows the memory utilization of Stress test. Similar to other load tests on, an average serverless consumes more memory and provides better response time. Each request in Lambda & Cloud function will be provided a dedicated memory space. Whereas each request in ECS & GKE will use the shared memory. Hence it can't be a head-to-head comparison on memory utilization.

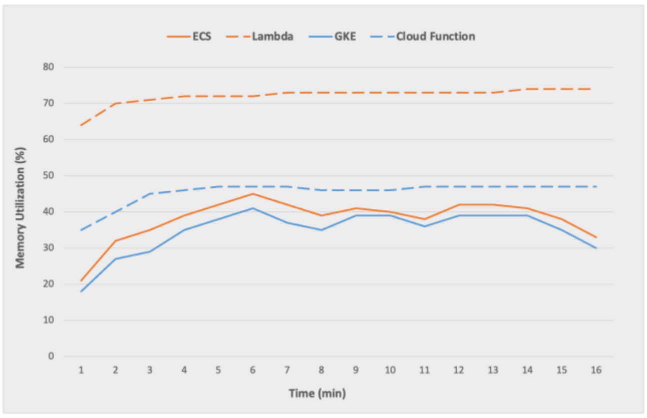


Figure 6.17 Stress - Memory Utilization [256 MB]

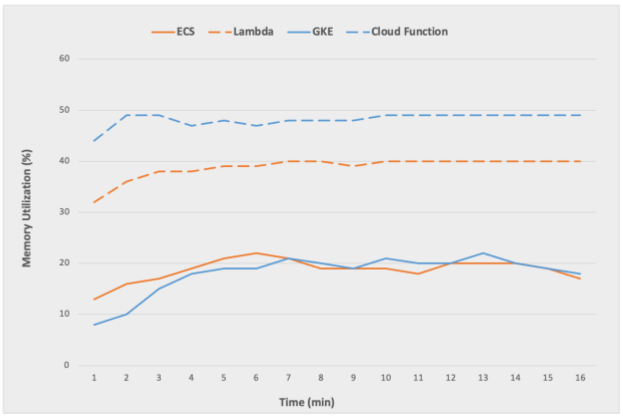


Figure 6.18 Random - Memory Utilization [512 MB]

Spike Load Test

Response Time

The Spike test is another variation of stress testing, but it does not gradually increase the load. Instead, it spikes to extreme load over a very short period of time. This test is executed to check how the system performs under a sudden surge of traffic and to check how the system recovers.

		Microservice				Serverless			
		AWS ECS		GKE		AWS Lambda		Google Functions	
S.No	Metrics	256	512	256	512	256	512	256	512
1	Request Rate (req/s)	48	48	32	32	163	225	160	225
2	Average Response Time (ms)	7000	7000	11000	12000	1450	785	1500	780
3	Minimum Response Time (ms)	235	226s	370	360	940	580	988	520
4	Standard Deviation (ms)	7100	6900	10000	10500	1400	770	1460	760
5	Total Requests	32000	42000	28000	28000	147180	203500	146000	204000

Table 6.7 Spike Load Metrics

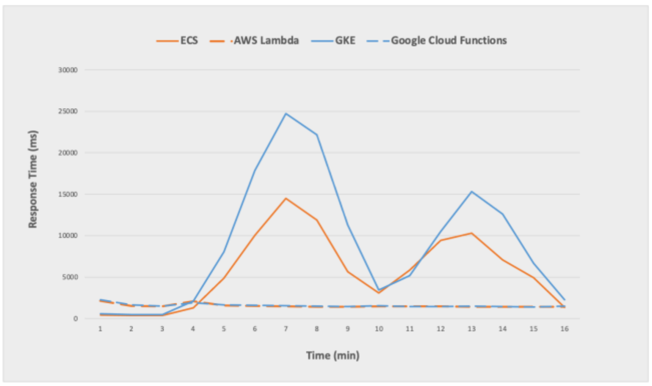


Figure 6.19 Spike - Response Time [256 MB]

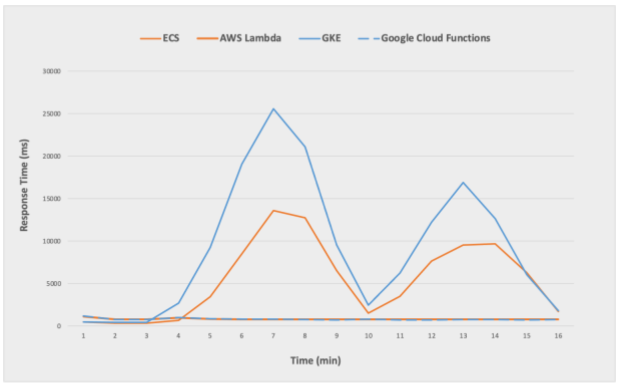


Figure 6.20 Random - Response Time [512 MB]

Compared to incremental test, AWS Lambda & Google cloud function has identical results. Both serverless functions seem to handle spike load without any errors or additional delays. AWS Lambda Function with 256 MB memory returned with a response time of 1450ms, processing 162 req/s with a total of 147K requests during peak load, and similarly, Google functions have the same results for 256 MB. From the figure, it is evident that both the functions have a slight cold start but seem to have an aggressive scaling while handling spikes. Response time of 1450ms seems to be reasonable as it offers consistent performance throughout the test. From Table 6.7, It is clearly noticed that after adding more memory, both the functions performed much better, and the average response time has drastically come down to half of the previous test's average response time.

On the other hand, the Microservices platform has poor performance while handling a drastic spike. After initially running with 256 MB, decide to check whether microservice platform performance improves with 512 MB. As a result, both platforms struggle to handle the peak load. Figures 6.19 and 6.20 displays all requests in a timeline. The reason for increased response times in AWS ECS and GKE is apparent. One valid reason for increased response time is initially, AWS ECS serves the request with 4 EC2 instances, and when the requests are more, EC2 starts scaling, and that adds a huge delay of request latency Until the new instance is launched and tasks are placed, ECS continues to serve the request with a slower rate and also noticed that each container serves five req/s and no additional space for more request. At the peak load, the ECS cluster scaled up to a max of 8 instances and 29 containers.

Microservice platform backed by AWS ECS unable to handle simultaneous requests without causing additional delays. In contrast, AWS Lambda is capable of withstanding sudden spikes and is able to serve simultaneous requests. Compared to the microservice platform, the Serverless function is quick and instantly creates new runtime.
AWS Lambda scales up effortlessly without introducing additional delays and handles scaling from 0 to 1000concurrent requests very well. Compared to AWS, Google Functions has a less cold start and handles scaling. Based on the tests, both the serverless functions on AWS & Google are suitable to handle spike workload if the initial limited cold start is not a problem. Both AWS Lambda and Google functions can also handle request bursts. Where the Microservices platforms on AWS ECS & GKE is not optimized to handle sudden burst, this is mainly due to the scaling behaviour with a setting like CPU, target utilization, etc. However, this can be mitigated by spinning pre-warmed instances with high CPU & Memory configuration, which intern results in increased cost. In the end, the winner of this case study is the Serverless platform. The serverless strategy has a better performance in terms of performance stability and scaling agility regardless of the cold-start problem, but afterward, it becomes stable.

Memory Use

Figure 6.21 and 6.22 depicts the memory utilization under the Spike load test.

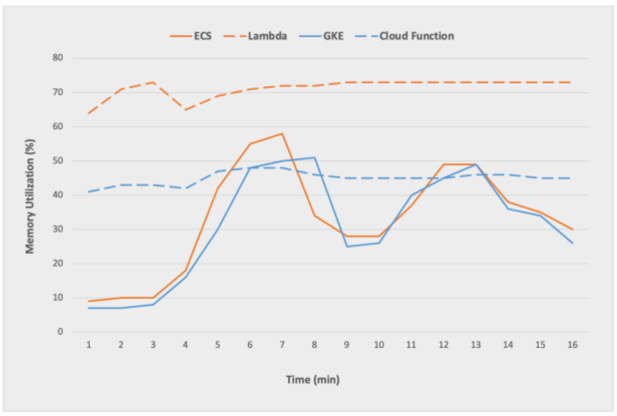


Figure 6.21 Spike - Memory Utilization [256 MB]

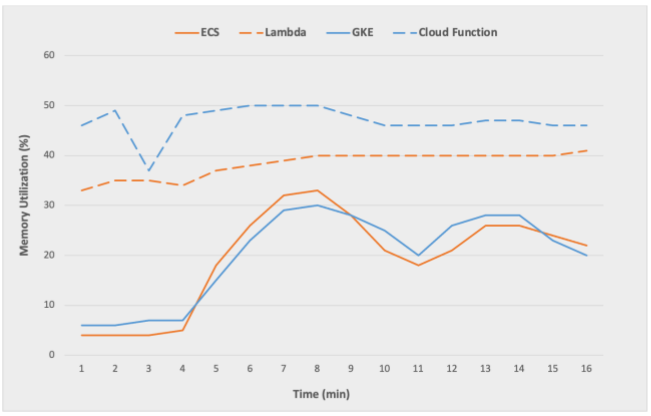


Figure 6.22 Spike - Memory Utilization [512 MB]

Though the usage of memory in microservice is less under normal load, it demands more memory during a sudden spike in traffic which doesn't contribute to the overall response time. Whereas AWS Lambda and cloud functions utilize more memory during the overall duration, it provides better response time and high throughput compared to ECS & GKE. Comparing memory utilization between Serverless and microservice doesn't affect the overall application performance.

6.4.2 Cost

This section examines the cost between microservices and serverless deployment tested for different scenarios in AWS and Google cloud platforms. Each cloud provider has its own billing service that allows to easily understand the spending, usage, and manage billing. The below Tables 6.8, 6.9, 6.10, 6.11, 6.12 shows the cost of cloud services used to deploy microservices and serverless architecture during load test scenarios.

Virtual machines in the public cloud are usually billed on an hourly basis. Each instance type has a different price based on the computing, memory, and network configuration. In this evaluation, the cost is calculated for each service used during the load test duration, which is 15 minutes. Once the cost is derived for 15 mins, it is converted into a monthly cost for better comparison. For e.g., In Table 6.8 constant rate scenario requires five instances to complete the test in 15 mins to meet the load test criteria. Then the cost is calculated for five instances of t3.xlarge runs for a whole month (720 hours)

Total Instance cost = [(no of instances * 720) * cost per hour]

Similarly, the cost for serverless is calculated based on the number of requests, their configuration, and the execution time. Based on the results, serverless supports a greater number of requests compared to microservice in stress and spike scenarios. To make even comparison, the number of requests served by microservices is taken into consideration to calculate the cost for serverless. Irrespective of memory configured for containers, the number of instances and load balancer remains unchanged in the Microservice architecture; hence the cost is the same for both the configuration. For each scenario, the serverless cost is calculated by multiplying the number of requests per hour with the number of hours per month (no of request/hr * 720). Serverless cost is calculated using the cost calculator provided by AWS & Google (AWSLambdaPricingCalculator, 2022; CloudGoogleCalculator, 2022).

Based on the results, each scenario has generated a certain number of requests. However, the cost of a serverless environment is economically low for both cloud providers. It is obvious that both lambdas offer 1M free requests per month(AWSLambdaPricing, 2022) and google offers 2M free requests per month (GoogleCloudFunctionPricing, 2022)when keeping the serverless cost low. Further analysis is done by varying the memory configuration of the serverless function to process the same number of requests. It was noticed from the function pricing model that increases in memory increase the cost of serverless function, which is almost 50% more, as shown in Table 6.12, and tries to match the cost of the microservice environment.

In order to have a better cost-benefit from a holistic view for running and operating an application in the cloud, both architecture patterns should be rightly balanced depending on application requirements. For an application with a predictable load, microservice offers lower cost-benefit compared to serverless due to its autoscaling feature of scaling down the instances when not used or when the traffic is low. On the other side, when dealing with small duration and less memory footprint, serverless is a viable low-cost option that is good for an unpredictable spike use case that offers better performance compared to microservice.

Cost for Microservice Architecture

- **Constant Rate Load Test**

Provider	Service	No of Instance	Cost per hour (USD)	Quantity per month	Cost per month (USD)
AWS	Ec2 instance (t3.xlarge)	5	0.1888	720*5	679,1
	Monthly Infra Cost				679,1
Google	e2-standard-4	6	0.134012	720*6	578,931
	Monthly Infra Cost				**578,931**

Table 6.8 Constant Load - Cost Calculation

- **Random Load Test**

Provider	Service	No of Instance	Cost per hour (USD)	Quantity per month	Cost per month (USD)
AWS	Ec2 instance (t3.xlarge)	4	0.1888	720*4	543,744
	Monthly Infra Cost				543,744
Google	e2-standard-4	5	0.134012	720*5	483,443
	Monthly Infra Cost				483,443

Table 6.9 Random Microservice - Cost Calculation

- **Stress Incremental Load Test**

Provider	Service	No of Instance	Cost per hour (USD)	Quantity per month	Cost per month (USD)
AWS	Ec2 instance (t3.xlarge)	9	0.1888	720*9	1223,424
	Monthly Infra Cost				1223,424
Google	e2-standard-4	9	0.134012	720*9	868,297
	Monthly Infra Cost				**868,297**

Table 6.10 Stress Microservice - Cost Calculation

- **Spike load Test**

Provider	Service	No of Instance	Cost per hour (USD)	Quantity per month	Cost per month (USD)
AWS	Ec2 instance (t3.xlarge)	8	0.06701	720*8	1087,488
	Monthly Infra Cost				1087,488
Google	e2-standard-4	9	0.134012	720*9	868,297
	Monthly Infra Cost				**868,297**

Table 6.11 Spike Microservice - Cost Calculation

Cost for Serverless

		Constant		Random		Stress		Spike	
	Cost Metrics	**256**	**512**	**256**	**512**	**256**	**512**	**256**	**512**
AWS	Total Number of Executions	126720000	129312000	83520000	84960000	152640000	158400000	120960000	123840000
	Estimated Execution Time (ms)	742	742	742	742	742	742	742	742
	Request Costs ($)	25.34	25.86	16.70	16.99	30.53	31.68	24.19	24.77
	Execution Costs ($)	391.85	799.74	258.27	525.44	472.01	979.64	374.04	765.90
	Total Cost ($)	**417.20**	**825.60**	**274.97**	**542.43**	**502.53**	**1011.32**	**398.23**	**790.67**

		Constant		Random		Stress		Spike	
	Cost Metrics	**256**	**512**	**256**	**512**	**256**	**512**	**256**	**512**
Google	Total Number of Executions	126720000	129312000	83520000	84960000	152640000	158400000	120960000	123840000
	Estimated Execution Time (ms)	742	742	742	742	742	742	742	742
	Request Costs ($)	49,89	50,92	32,61	33,38	60,26	62,56	47,58	48,74
	Execution Costs ($)	431,87	884,54	140,31	580,12	520,83	1084,18	412,11	846,97
	Total Cost ($)	**481,76**	**935,46**	**172,92**	**613,5**	**581,09**	**1146,74**	**459,69**	**895,71**

Table 6.12 Serverless - Cost Calculation

6.4.3 Development Experience

This section covers the application development and deployment experience on microservice and serverless during experimentation.

One of the drawbacks identified in the serverless function is the testing of an application. For the developer to test the code, a development AWS or Google environment is required. For the image processing application, code is tested by packaging and uploading the code to Lambda or Google function and then trigger using API gateway. This has a potential delay in the development process. Whereas in the microservice environment, the application is deployed as a docker container in the local development machine and invoked using an HTTP endpoint, which enables the developer to perform testing faster than serverless.

One of the improvements noticed in the serverless evolution is the flexibility of developing and deploying multiple resources of an application together, which was not the case before. With the Serverless Framework, this is now possible to manage the application with a single serverless file. Similarly, in microservices, tools like docker-compose and Kubernetes deployment help to achieve this.

The deployment of microservice architecture in AWS & Google requires an additional skill set to deploy and manage. These services have their own configuration to maintain in cloud infrastructure. Whenever a new feature is released or new versions are published, the team should be held the responsibility to communicate with other teams as the new version might break other services. There should be collaboration and communication among multiple teams depending on the upgrade.

During Microservice deployment, it is noticed there is a dependency on other services like Autoscaling, Target group, AMI Management, Kubernetes cluster Management, etc. The infrastructure required to build this microservice platform needs to be managed by the user. This includes life cycle management of Amazon machine images used to deploy the underlying compute, Network infrastructure, monitoring and maintenance of autoscaling group to keep instance healthy, and configuration for integrating with AWS monitoring services like CloudWatch. This is an additional overhead for the users or DevOps team to maintain and manage the Cloud Infrastructure.

On the other side, ready-to-use services like AWS Lambda & Google function reduce the overhead of managing the underlying cloud infrastructure. This helps to save a lot of time and spend more time on improving business logic. Since both the architecture are deployed using a Serverless framework. The deployment of Serverless is faster than microservices because the number of resources involved in the serverless architecture is minimal, whereas in microservice, it is more self-managed resources.

Chapter 7

Discussion

This section discusses the overall detail of the work done and the results obtained from the experiments.

This thesis study focuses on comparing microservices and serverless platforms in the cloud by deploying an application with the aim of identifying how these two different deployment strategies affect the performance and cost of running and scaling an application in the cloud. A sample application is deployed in AWS and Google cloud, and several load test scenarios are executed, and results are obtained. Apart from cloud platforms, some of the tools like K6, Influx DB, Grafana, and Serverless Framework were used to support the overall experiment. The following section of the discussion is split into two main sections, one is a comparison based on quantitative metrics, and the other one is based on qualitative metrics.

7.1 Quantitative Metrics

7.1.1 Performance

This section discusses the application performance results when deployed in microservice and serverless platforms. Further reading is split into two sections to discuss the research metrics.

Response time

It is evident from all experiments conducted serverless strategy suffers from the cold-start problem. This happens when a function is invoked for the first time or when the function has not been used for some time, or when it is updated. Both AWS and Google provision a new backend container in the Micro VM for the function to execute. This takes a certain amount of time, and the request needs to wait until the function backend is ready to serve. This wait is usually taken by the container to initialize the run time environment, pull the function source code and install the dependency libraries. This causes a certain portion of requests that are served by new instances to have higher latency, otherwise known as a cold start. There already have been many types of research already performed to decrease the cold start time, like using pre-warmed containers (MarKusThommas, 2017), periodic warming consisting of submitting dummy requests periodically to induce a cloud service provider to keep containers warm and pause containers (Mohan et al., 2019). But these recommended methods add complexity to keep the environment warm. To overcome this, cloud providers introduced provisioned concurrency to keep functions initialized and ready to respond in milliseconds. This is to keep in mind that provisioned concurrency incurs an additional price which increases the overcall cost of the application infrastructure while considered serverless. **Cold start issue in serverless thus adds latency to the overall response time which is noticed in all starts**. Before deciding serverless choice of programming language used for application development should be keep in mind.

Despite the cold start issue in the serverless deployment during all the load tests, the microservices deployment strategy suffers from the traffic redistribution problem, which leads to a high response time during the Spike load test. This affects the overall average response time when the application needs to handle sudden spikes to accommodate more requests. One potential reason is that these spikes coincide with the scaling out activity of the autoscaling in AWS and scaling of PODS and Nodes in GKE, which results in the increased response time. **Despite a cold start, If an application demands stable latency overall, then the choice of serverless, either in AWS or Google, is the right deployment strategy**.

In another load test, it is evident that microservices deployed using AWS ECS outperform while handling the constant request. If the application requests are static and constant, then a microservice deployment strategy would provide better response time with less infrastructure cost and no cold start. One potential reason could be due to no scaling activity; hence microservice is able to provide consistent performance. Compared to ECS, the microservice deployed by GKE has a higher response time. This could be improved by enabling the Node provisioning feature in GKE, and there by, the response time for the constant load would be better.

Memory Use

Memory utilization or usage is one of the key attributes while evaluating an application's performance. In the modern application world, this metric plays a different role as compared to monolithic applications. Based on the results, memory utilization in microservice is less compared to serverless, which is 85% of overall memory in all the tests. One potential reason for this behavior is in serverless, each invocation or each request invoked will have a dedicated run time container environment with memory available to use, and utilization is calculated against the single request. Whereas in microservices, each container processes multiple requests, and the utilization of memory is shared by multiple requests. Hence memory metrics can't be considered the key metrics to making a decision against a better strategy. Similarly, for CPU, Cloud Providers don't offer CPU utilization Metrics for Serverless. The reason is that the CPU is not a configurable item in both AWS Lambda and Google functions. CPU is proportionally allocated to Memory in the backend by providers, as mentioned in the section, due to which CPU is not the right metric to benchmark the results.

Concluding the performance discussion, there is no clear winner in terms of performance. Serverless shows great potential while handling unpredictable traffic with better performance. At the same time, microservice shows better response while handling constant and predictable load traffic.

7.1.2 Cost

As we compare the cost between serverless and microservice platform. Serverless has a higher response time while offering a lower cost in constant load tests when compared to microservice. At the same time, in another test comparison, the serverless cost is relatively low, with a better response time when dealing with peak workloads. Based on the results, it is evident that serverless offers the same price irrespective of the load test. But the price varies when there is a change in memory configuration. Hence, serverless is most suited for applications that run for a shorter duration and the need for memory is relatively low. More memory relates to high cost. On the other side, for a long-running application with a predictable load, microservice offers lower cost-benefit compared to serverless due to its autoscaling feature of scaling down the instances when not used or when the traffic is low

7.2 Qualitative Metrics

7.2.1 Scalability

In terms of comparing scalability and agility of microservice and serverless platform. Serverless wins over the race of scaling and agility. This is because the cloud provider offloaded the overhead of scaling activity and is managed by the cloud provider itself. Microservices deployment starts to scale after the system has reached the defined criteria for at least one minute. There is always a delay in responsiveness to re-balance the current workload. As a result, there is an increase in response time with the increasing workload, and then it drops after the new containers have been launched. For e.g., in the above microservices deployment, the average CPU utilization is 70% for a consecutive 1 minute than the scaling trigger. This limitation in microservice impacts the scaling and has a delayed response which is not the case with serverless. In one of the spike and stress load tests in microservices, some of the requests were errored and dropped. It is due to the instance scaling activity, and the existing container couldn't accommodate any more requests, which lets down the average response time low.

7.2.2 Security

Based on the literature study, both the deployment architecture can be integrated with an external authentication system. However, Serverless AWS API Gateway & Google API has the native integration support with AWS Cognito (AWSCognito, 2022) and Google Firebase Authentication (FirebaseAuthentication, 2022), which is a service for managing user and application authentication. No additional configuration is required to manage, and it is serverless by nature. This feature pushes serverless to the top of the list on this topic. Authentication in microservices needs custom configuration in the form of code or external tools, which adds additional complexity to managing the infrastructure.

Microservice and serverless are similar in several aspects, and the strategies to manage them and keep them secure should be similar, too. However, there are some important differences when it comes to managing and securing certain dimensions of a serverless or

Microservice workload, such as the extent of the responsibility for the host environment and the tools to use.

In Serverless, permitter security can be controlled by having a separate IAM role with restricted permission attached to the functions. By implementing this, access between services is controlled with fine-grained access. Similar to Google cloud, the service account held the role of applying fine-grained access privileges to the function there by increasing the level of permitter security around the serverless application. In addition, AWS Lambda & Google function has the control to define who can invoke the function
On the other hand, both AWS ECS & Google Kubernetes engine access to services is controlled by the task role in ECS and service account in Kubernetes. In AWS, ECS tasks have an IAM role attached to them. The permissions attached to an IAM role are assumed by the containers running in the task. Similar to GKE, service accounts are bound to POD to have controlled permission on the POD Level.

7.2.3 Development Experience

From the deployment perspective, the deployment of application code into the serverless platform is quicker and faster compared to Microservice-based deployment. With the use of an open-source serverless application, the framework application can be easily built by integrating all resources into one YAML file and triggering the deployment; this has made the serverless deployment complete within a couple of hours. Whereas microservice it's a two-step process, the application code has to first convert them into docker images and then build underlying infrastructure to deploy the image as the container. The time duration to complete the whole deployment process is more compared to serverless.
From the operational perspective, the serverless function takes care of everything required to run and scale the implementation to meet the demand with high availability. No administration of infrastructure is needed. No operational team is required to manage. This allows the team to innovate faster and move quickly, and the developer focuses on business logic because security and scaling are managed by AWS. On the other hand, microservices-based deployment needs a dedicated operation team to run and operate the underlying infrastructure along with the developer team. This drives a need for the DevOps team to

collaborate with developers and operations together to deliver higher quality software much faster, which in turn increases the cost of the project. And in addition, an organization needs to address the skill gap between Developers and the Operations team

7.2.4 Controllability and Visibility

Though serverless addresses the complexity of orchestrating the container by handing over the code to the cloud provider to run on the cloud platform, the control over the underlying layer is lost when it comes to managing the dependency libraries, run time, and resource limit. And also, it losses visibility on how the cloud provider scraps the function environment when a function is deleted and the user is not sure whether the data is completely removed from the backend. At the same time, the microservice-based environment is fully owned by the end-user, who has full control and visibility over the environment. On the other side, serverless is improved over the last couple of years on gaining visibility over the application characteristics. Based on the theoretical study, services like AWS Cloudwatch, AWS X-ray, Monitoring explorer, and Cloud Trace improves the visibility of the Serverless platform in the aspect of monitoring latency between the Application APIs.

7.3 Limitations

Though serverless is gaining popularity among architects and developers, certain limitations in the serverless platform will create a potential delay in adopting serverless. Some of them are 1. Runtime - Each function has a max run-time of around 15 minutes in AWS Lambda and 10 minutes in event-driven, and 60 mins for HTTP function in Google. Hence long-running applications are not suitable for Serverless deployment.

7.4 Challenges Faced

In this section, the main challenges faced during experimentation were mentioned. The challenges reported here would help other researchers to address during the early stage of experimentation.

Public Cloud with Free Credit

The experiment carried out in this thesis study is non-fundable. Public cloud provider AWS and Google cloud platform were considered to conduct the experimentation. Personal accounts were created in both these providers to deploy the sample application and run load tests. Both accounts initially come up with free credits with certain limitations. During experimentation in AWS, one of the key items evaluated is the cold start in Lambda. Researchers addressed this issue by mitigating it using pre-warmed containers. As technology continues to evolve, AWS addressed this using Provisioned concurrency, which is high in cost and doesn’t fall under free credit. Hence to avoid the accumulation of high costs in the personally-owned AWS account experiment against enabling provisioned concurrently is skipped, and results are analyzed based on theoretical data shared by AWS. The same applies to Google cloud.

Setting up Kubernetes platform

One of the gaps addressed in this report is extending the comparison study to other cloud providers. Hence along with AWS, Google cloud is considered for evaluation. In order to have a fair comparison between microservice and serverless in two different providers, the same kind of service needs to be evaluated. In AWS, elastic container services are used for evaluation in microservice. ECS is AWS managed container orchestrator with no cluster management and is easy to deploy. Whereas, in Google, the only available container orchestrator solution is GKE (Google Kubernetes Engine). To deploy and operate GKE, one should need the skill set on Kubernetes to deploy and run the application. It took time to understand the Kubernetes technology and then adapt GKE is very time-consuming, which adds potential delay in overall thesis work.

7.5 Summary

Decision Drivers	Serverless	Microservice
Performance	• Provides better performance while handling high and unpredictable traffic • Due to faster scaling, able to accommodate the high number of requests with a better response time • Suffers from a cold start	• The predictable and constant request has better performance • Performance is impacted due to load balancing and traffic distribution while scaling.
Cost	• Cheaper compared to container-based Microservice platform • No charges for function when not used • Granularity pricing makes cost-effective • Functioning with more memory will increase the cost.	• Compute level pricing irrespective of CPU or memory configuration • Cost is high when a high configuration instance is used, which results in better performance
Development Experience	• Simple deployment stands out unique. • No infrastructure management • No configuration of docker or Kubernetes. • No Ops – Operations team is not required • All of the above results in faster time to market	• An operations team is required to manage the underlying infrastructure • Complexity in managing Kubernetes cluster and configuration • Due to underlying infrastructure, the time taken to deploy microservice is high and needs knowledge on managing infrastructure components
Scalability	• Highly scalable • Scaling of functions taken care of by the cloud provider	• Highly scalable • The developer needs to configure and manage to scale • Scaling starts after the system reaches the desired criteria for one minute
Controllability & Visibility	• No visibility of underlying infra • Lack of control of underlying runtime environment	• High visibility of underlying infra • Developer controlled run environment using docker containers

Chapter 8

Conclusion

Microservices and serverless have emerged a long way from monolithic architectures. Both satisfy the modern generation application requirements of highly scalable, flexible, and agility, and the technologies to enable them are continuously improving. Based on the experimental analysis for the POST method concludes that not all qualitative and quantitative attributes fit one approach. For random and constant load, microservices has better performance than serverless in both memory configuration with pre-built instances. In contrast, serverless outperforms better than microservices for both memory configuration having stability, performance, and scaling agility with stress and triangle load. The reason for instability occurs in microservices during the surge of traffic due to the scaling of instances. Microservices have more controllability and visibility than serverless for backend services. Serverless is better than microservices with certain characteristics in some of the areas to be specially mentioned, such as faster deployment and NoOps. However, deep consideration is still needed choosing between them when designing infrastructure for an enterprise. It is about choosing wisely and leveraging the advantages of each approach.

8.1 Answer to Research questions

This section answers the research questions, which were formulated at the beginning of this thesis. The research questions are answered based on the tests, observations, and findings which were made during the research work as a part of this thesis.

RQ1: What are the main factors that influence the decision to choose between microservices and serverless?

Performance, cost, and scalability are the three main factors that influence the decision of microservices and serverless deployment. The microservice platform provides low average latencies when the load is static and predictable. However, not being the right fit for applications with unpredictable traffic results in high latency. On the other side, serverless can offer better latency for applications with unpredictable spike traffic. However, high latency can occur in serverless platforms, especially with a cold start which could be

mitigated using provisioned concurrency or min instances. In conclusion, serverless can be feasible if the occasional high latency is not an issue.
In terms of costs, Serverless platforms were found to be a very cost-effective solution as it priced on demand. In conclusion, different platforms should be carefully considered and compared in terms of costs when making architecture decisions.

RQ2: Which architecture among the two (microservices & serverless) is suitable for deploying an application in the cloud?
Although serverless application shares many of the same benefits as a microservice, there are a plethora of differences if we see it from a holistic view. Select serverless computing when there is a need for automated scaling as well as low runtime costs. Serverless architecture is a good choice for applications that run only for short periods of time and to handle the surge in traffic. Services that need to be highly available will make a perfect fit for serverless.
Choosing between the two-deployment strategy is not always necessary. It is recommended to integrate the benefits and limitations of microservices and Serverless technologies by using them together. However, one can also integrate both architectures to build a cost-efficient application platform and leverage the technology of Microservice-based serverless platform as well.

8.2 Future work

The research in this thesis has its limitations and can be complemented with further research. The following topics are proposed for further research

Extend the comparison toward serverless-based Microservice platform: This thesis examined the performance, cost, and security comparison between Container Orchestrator platform and Serverless, more specifically between ECS and Lambda in AWS and between Cloud Functions and GKE standard mode. The research could be extended to compare serverless-based microservices platforms versus function-based serverless. For example, AWS Fargate vs. AWS Lambda. AWS Fargate is a serverless, pay-as-you-go compute engine that lets you focus on building applications without managing servers, and in google, GKE

Autopilot mode is the serverless based container platform. The appendixes section of this thesis provides the scripts to deploy the Serverless and Microservice platform, which could be reused for further research.

Extending the comparison to additional cloud providers: The experiment in this thesis is conducted in Amazon Web Services and Google Cloud Platform. The tests could be extended to cover other big cloud service providers like Microsoft Azure and some other providers.

Reference List

Adzic, G., & Chatley, R. (2017). *Serverless computing: economic and architectural impact*. 884–889. https://doi.org/10.1145/3106237.3117767

Albuquerque Jr, L. F., Ferraz, F. S., Oliveira, R. F. A. P., & Galdino, S. M. L. (2017). Function-as-a-Service X Platform-as-a-Service: Towards a Comparative Study on FaaS and PaaS. *The Twelfth International Conference on Software Engineering Advances Function-as-a-Service*, *c*, 206–212.

Amazon ECS-optimized AMI - Amazon Elastic Container Service. (n.d.). Retrieved May 2, 2022, from https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-optimized_AMI.html

AmazonEC2AutoScaling. (2022). *What is Amazon EC2 Auto Scaling? - Amazon EC2 Auto Scaling*. https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html

AmazonEC2types. (2022). *Amazon EC2 Instance Types - Amazon Web Services*. https://aws.amazon.com/ec2/instance-types/

AmazonECR. (2022). *What is Amazon Elastic Container Registry? - Amazon ECR*. https://docs.aws.amazon.com/AmazonECR/latest/userguide/what-is-ecr.html

AmazonECS. (2022). *Fully Managed Container Orchestration – Amazon Elastic Container Service (Amazon ECS) FAQs – AWS*. https://aws.amazon.com/ecs/faqs/

AmazonELB. (2022). *What is a Network Load Balancer? - Elastic Load Balancing*. https://docs.aws.amazon.com/elasticloadbalancing/latest/network/introduction.html

AmazonS3. (2022). *What is Amazon S3? - Amazon Simple Storage Service*. https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html

AWSCognito. (2022). *What is Amazon Cognito? - Amazon Cognito*. https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html

AWSLambdaPricing. (2022). *Serverless Computing – AWS Lambda Pricing – Amazon Web Services*. https://aws.amazon.com/lambda/pricing/

AWSLambdaPricingCalculator. (2022). *AWS Lambda Pricing Calculator*. https://s3.amazonaws.com/lambda-tools/pricing-calculator.html

AWSLAMBDAsecurity. (2022). *Lambda Executions - Security Overview of AWS Lambda*.

https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/lambda-executions.html

Baldini, I., Cheng, P., Fink, S. J., Mitchell, N., Muthusamy, V., Rabbah, R., Suter, P., & Tardieu, O. (2017). The serverless trilemma: Function composition for serverless computing. *Onward! 2017 - Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Co-Located with SPLASH 2017*, 89–103. https://doi.org/10.1145/3133850.3133855

Baškarada, S., Nguyen, V., & Koronios, A. (2020). Architecting Microservices: Practical Opportunities and Challenges. *Journal of Computer Information Systems*, *60*(5), 428–436. https://doi.org/10.1080/08874417.2018.1520056

BBVA. (2020). *Economics of "Serverless" | BBVA*. https://www.bbva.com/en/economics-of-serverless/

Chris Tozzi. (2021). *Microservices vs. Serverless Architecture | Sumo Logic*. https://www.sumologic.com/blog/microservices-vs-serverless-architecture/

Cloudflare. (2021). *Serverless computing vs. containers | How to choose | Cloudflare*. https://www.cloudflare.com/en-in/learning/serverless/serverless-vs-containers/

CloudGoogleCalculator. (2022). *Google Cloud Pricing Calculator*. https://cloud.google.com/products/calculator

CloudStorage. (2022). *Cloud Storage | Google Cloud*. https://cloud.google.com/storage

DNSstuff. (2019). *Best Server and Application Response Time Monitoring Tools + Guide - DNSstuff*. https://www.dnsstuff.com/response-time-monitoring

Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R., & Safina, L. (2017). Microservices: How to make your application scale. *ArXiv*.

ECSCapacityprovider. (2022). *Amazon ECS capacity providers - Amazon Elastic Container Service*. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/cluster-capacity-providers.html

ECSserviceautoscaling. (2022). *Service auto scaling - Amazon Elastic Container Service*. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-auto-scaling.html

Eismann, S., Scheuner, J., van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., Abad, C. L., & Iosup, A. (2020). A Review of Serverless Use Cases and their Characteristics. *ArXiv*. http://arxiv.org/abs/2008.11110

Eivy, A. (2017). Be Wary of the Economics of "Serverless" Cloud Computing. *IEEE Cloud Computing*, *4*(2), 6–12. https://doi.org/10.1109/MCC.2017.32

FirebaseAuthentication. (2022). *Firebase Authentication | Firebase Documentation*. https://firebase.google.com/docs/auth

Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., … Delimitrou, C. (2019). An open-source benchmark suite for cloud and IoT microservices. *ArXiv*.

Gearheart. (2021). *How to Build a Scalable Web Application for Your Project | Gearheart*. https://gearheart.io/articles/how-build-scalable-web-applications/

Ghayyur, S. A. K., Razzaq, A., Ullah, S., & Ahmed, S. (2018). Matrix clustering based migration of system application to microservices architecture. *International Journal of Advanced Computer Science and Applications*, *9*(1), 284–296. https://doi.org/10.14569/IJACSA.2018.090139

Google. (2020). *Global Locations - Regions & Zones | Google Cloud*. https://cloud.google.com/about/locations

GooglecloudAPIGateways. (2022). *Quickstart: Secure traffic to a service with the Cloud console | API Gateway Documentation | Google Cloud*. https://cloud.google.com/api-gateway/docs/secure-traffic-console

GoogleCloudFunctionPricing. (2022). *Pricing | Cloud Functions | Google Cloud*. https://cloud.google.com/functions/pricing

GooglecloudRoute. (2022). *Routes overview | VPC | Google Cloud*. https://cloud.google.com/vpc/docs/routes

GoogleComputeEngine. (2022). *General-purpose machine family | Compute Engine Documentation | Google Cloud*. https://cloud.google.com/compute/docs/general-purpose-machines

Googlefunction. (2022). *Cloud Functions | Google Cloud*. https://cloud.google.com/functions

GoogleGKE. (2022). *GKE overview | Kubernetes Engine Documentation | Google Cloud*. https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview

GoogleTCP. (2022). *Configuring TCP/UDP load balancing | Kubernetes Engine Documentation | Google Cloud*. https://cloud.google.com/kubernetes-engine/docs/how-to/service-parameters

Grafana, I. (2022). *With Grafana and InfluxDB | Grafana documentation*. https://grafana.com/docs/grafana/latest/getting-started/getting-started-influxdb/

Hellerstein, J. M., Faleiro, J., Gonzalez, J. E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., & Wu, C. (2018). Serverless Computing: One Step Forward, Two Steps Back. *ArXiv*, *3*.

Ivanovic, N. (2021). *Montecha - Blog - Serverless platform*. https://montecha.com/blog/serverless-platform/

Jackson, D., & Clynch, G. (2019). An investigation of the impact of language runtime on the performance and cost of serverless functions. *Proceedings - 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018*, 154–160. https://doi.org/10.1109/UCC-Companion.2018.00050

Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J., & Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, *35*(3), 24–35. https://doi.org/10.1109/MS.2018.2141039

Javatpoint. (2020). *SOA - Service Oriented Architecture - javatpoint*. https://www.javatpoint.com/service-oriented-architecture

K6. (2022). *k6 Documentation*. https://k6.io/docs/

k8poddeployment. (2022). *Deployments | Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

Kumar, M. (2019). Serverless Architectures Review, Future Trend and the Solutions to Open Problems. *American Journal of Software Engineering*, *6*(1), 1–10. https://doi.org/10.12691/ajse-6-1-1

Lehmann, M., & Sandnes, F. E. (2017). A framework for evaluating continuous microservice delivery strategies. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3018896.3018961

Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., & Pallickara, S. (2018). Serverless computing: An investigation of factors influencing microservice performance. *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*, 159–169. https://doi.org/10.1109/IC2E.2018.00039

Maherchandani, J. (2019). *The Idea of Micro-frontend. The term Micro Frontends extends the… | by Jitin Maherchandani | Medium*. https://medium.com/@jitin.maher/the-idea-of-micro-frontend-85028131112f

MarKusThommas. (2017). *Squeezing the milliseconds: How to make serverless platforms blazing fast! | by Markus Thömmes | Apache OpenWhisk | Medium*. https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0

McGrath, G., & Brenner, P. R. (2017). Serverless Computing: Design, Implementation, and Performance. *Proceedings - IEEE 37th International Conference on Distributed Computing Systems Workshops, ICDCSW 2017*, 405–410. https://doi.org/10.1109/ICDCSW.2017.36

Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., & Sukhomlinov, V. (2019). *Agile Cold Starts for Scalable Serverless*.

Packt. (2017). *Microservices and Service Oriented Architecture | Packt Hub*. https://hub.packtpub.com/microservices-and-service-oriented-architecture/

Pahl, C., & Jamshidi, P. (2016). Microservices: A systematic mapping study. *CLOSER 2016 - Proceedings of the 6th International Conference on Cloud Computing and Services Science*, *1*(January), 137–146. https://doi.org/10.5220/0005785501370146

Pellegrini, R., Ivkic, I., & Tauber, M. (2019). Towards a Security-Aware Benchmarking Framework for Function-as-a-Service. *ArXiv*, 1–4.

Ponce, F., Marquez, G., & Astudillo, H. (2019). Migrating from monolithic architecture to microservices: A Rapid Review. *Proceedings - International Conference of the Chilean Computer Science Society, SCCC*, *2019-Novem*(September). https://doi.org/10.1109/SCCC49216.2019.8966423

PrinceSinha. (2022). *Monitoring Application Response Times | Scout APM Blog*. https://scoutapm.com/blog/application-response-time-monitoring

PrivatelinkS3. (2022). *AWS PrivateLink for Amazon S3 - Amazon Simple Storage Service*. https://docs.aws.amazon.com/AmazonS3/latest/userguide/privatelink-interface-endpoints.html

Rajan, R. A. P. (2020). A review on serverless architectures-Function as a service (FaaS) in cloud computing. *Telkomnika (Telecommunication Computing Electronics and Control)*, *18*(1), 530–537. https://doi.org/10.12928/TELKOMNIKA.v18i1.12169

Ravirala, S. (2019). *Monolithic vs Microservices - Santhosh Ravirala*. https://santhoshravirala.com/monolithic-vs-microservices/?utm_source=rss&utm_medium=rss&utm_campaign=monolithic-vs-

microservices

Richards, M. (2016). *Microservices vs. service-oriented architecture – O'Reilly*. https://www.oreilly.com/radar/microservices-vs-service-oriented-architecture/

Sadaqat, M., Colomo-Palacios, R., & Knudsen, L. E. S. (2018). Serverless computing: a multivocal literature review. *NOKOBIT - Norsk Konferanse for Organisasjoners Bruk Av Informasjonsteknologi*, *26*(1). https://brage.bibsys.no/xmlui/handle/11250/2577600

Savage, N. (2018). Going Serverless. *Communications of the ACM*, *61*(2), 15–16. https://doi.org/10.1145/3171583

Sentia. (2020). *Sentia Tech Blog | AWS re:Invent 2020 Day 3: Optimizing Lambda Cost with Multi-Threading*. https://www.sentiatechblog.com/aws-re-invent-2020-day-3-optimizing-lambda-cost-with-multi-threading?utm_source=reddit&utm_medium=social&utm_campaign=day3_lambda

Sentinalone. (2021). *AWS Lambda Use Cases: 11 Reasons to Use Lambdas*. https://www.sentinelone.com/blog/aws-lambda-use-cases/

ServerlessFramework. (2022). *Serverless Framework - AWS Lambda Guide - Serverless.yml Reference*. https://www.serverless.com/framework/docs/providers/aws/guide/serverless.yml#lambda-events

Shafiei, H., Khonsari, A., & Mousavi, P. (2019). Serverless computing: A survey of opportunities, challenges and applications. *ArXiv*, 1–13. https://doi.org/10.31224/osf.io/u8xth

Taibi, D., El Ioini, N., Pahl, C., & Niederkofler, J. R. S. (2020). Patterns for serverless functions (Function-as-a-Service): A multivocal literature review. *CLOSER 2020 - Proceedings of the 10th International Conference on Cloud Computing and Services Science*, *Closer*, 181–192. https://doi.org/10.5220/0009578501810192

Viggiato, M., Terra, R., Rocha, H., Valente, M. T., & Figueiredo, E. (2018). *Microservices in Practice: A Survey Study*. *September*. http://arxiv.org/abs/1808.04836

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A., & Lang, M. (2017). Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing and Applications*, *11*(2). https://doi.org/10.1007/s11761-017-0208-y *vt3199/cloudimageproc*. (2022). https://github.com/vt3199/cloudimageproc

Waseem, M., Liang, P., & Shahin, M. (2020). A Systematic Mapping Study on Microservices Architecture in DevOps. *Journal of Systems and Software*, *170*(August). https://doi.org/10.1016/j.jss.2020.110798

XK6-output-influxdb. (2020). *xk6-output-influxdb/docker-compose.yml at main · grafana/xk6-output-influxdb*. https://github.com/grafana/xk6-output-influxdb/blob/main/docker-compose.yml

Xu, R., Nikouei, S. Y., Chen, Y., Blasch, E., & Aved, A. (2019). BlendMAS: A blockchain-enabled decentralized microservices architecture for smart public safety. *Proceedings - 2019 2nd IEEE International Conference on Blockchain, Blockchain 2019*, 564–571. https://doi.org/10.1109/Blockchain.2019.00082

Zollingkoffer, M. (2017). *Marc Zollingkoffer on Twitter: "My infographic on evolution from monolithic to capability oriented architectures https://t.co/076FSi1qg1 #Microservices #Cloud #Serverless https://t.co/ZRkaCtLmVv" / Twitter*. https://twitter.com/mz_74/status/841959762099085313

Appendix A

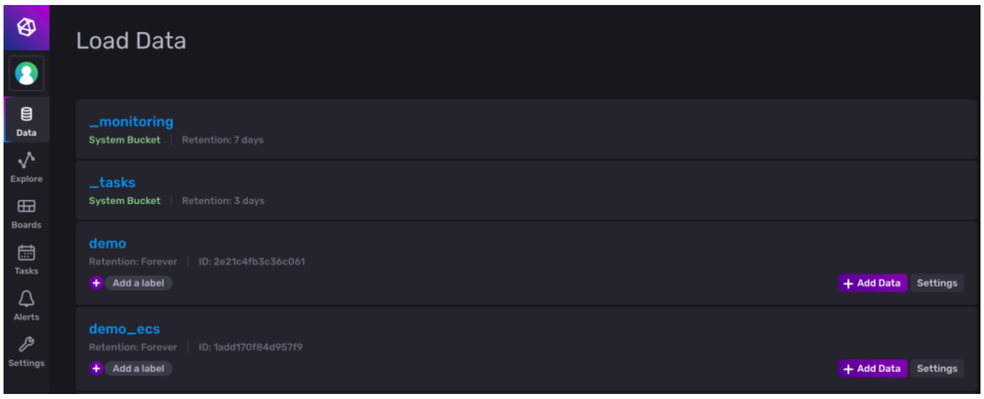
Load Test Code and Scripts

A.1 K6, Influx DB, and Grafana Deployment file

```
version: '3.8'

networks:
  k6:
  grafana:
  influxdb:

services:
  influxdb:
    image: influxdb:2.0-alpine
    networks:
      - k6
      - grafana
      - influxdb
    ports:
      - "8086:8086"
    environment:
      - DOCKER_INFLUXDB_INIT_MODE=setup
      - DOCKER_INFLUXDB_INIT_USERNAME=croco
      - DOCKER_INFLUXDB_INIT_PASSWORD=password1
      - DOCKER_INFLUXDB_INIT_ORG=k6io
      - DOCKER_INFLUXDB_INIT_BUCKET=demo
      -
DOCKER_INFLUXDB_INIT_ADMIN_TOKEN=EEKpryGZk8pVDXmIuy484BKUxM5jOEDv7YNoeNZUbsNbpbPbP6kK_qY9Zsyw
7zNnlZ7pHG16FYzNaqwLMBUz8g==

  grafana:
    image: grafana/grafana:8.1.3
    networks:
      - grafana
      - influxdb
    ports:
      - "3000:3000"
    environment:
      - GF_AUTH_ANONYMOUS_ORG_ROLE=Admin
      - GF_AUTH_ANONYMOUS_ENABLED=true
      - GF_AUTH_BASIC_ENABLED=false
    volumes:
      - ./grafana:/etc/grafana/provisioning/

  k6:
```

```
  build: .
  networks:
   - k6
  ports:
   - "6565:6565"
  environment:
   - K6_OUT=xk6-influxdb=http://172.31.214.14:8086
   - K6_INFLUXDB_ORGANIZATION=k6io
   - K6_INFLUXDB_BUCKET=demo
   - K6_INFLUXDB_INSECURE=true
    # NOTE: This is an Admin token, it's not suggested to use this configuration in production.
    # Instead, use a Token with restricted privileges.
   -
K6_INFLUXDB_TOKEN=EEKpryGZk8pVDXmIuy484BKUxM5jOEDv7YNoeNZUbsNbpbPbP6kK_qY9Zsyw7zNnlZ7pHG16FYzN
aqwLMBUz8g==
  volumes:
   - ./scripts:/scripts
```

A.2 Setup Repo in Influx DB



Load Data

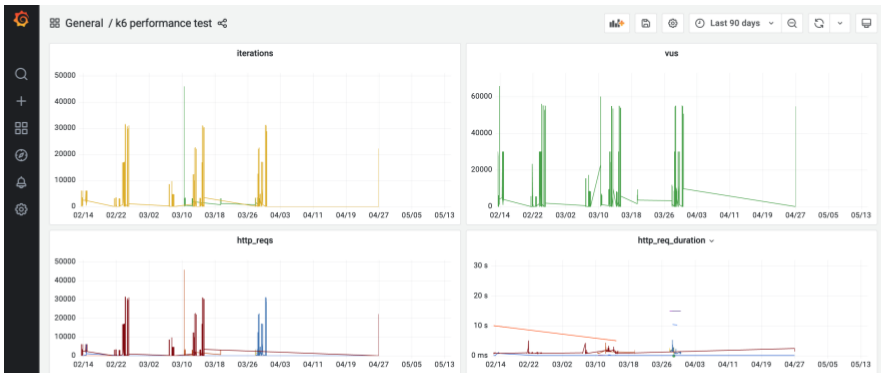
_monitoring
System Bucket | Retention: 7 days

_tasks
System Bucket | Retention: 3 days

demo
Retention: Forever | ID: 2e21c4fb3c36c061
Add a label — Add Data — Settings

demo_ecs
Retention: Forever | ID: 1add170f84d957f9
Add a label — Add Data — Settings

A.3 Setup Dashboard in Grafana



A.4 Build K6, InfluxDB and Grafana

```
Navigate to the path where docker compose file exists and run
```

```
docker-compose up -d
```

A.5 To run Load Test in AWS

```
sudo docker-compose run k6 run -<lambda_rps.js
sudo docker-compose run k6 run -<lambda_spiketriangle.js
sudo docker-compose run k6 run -<lambda_stressincremental.js
sudo docker-compose run k6 run -<lambda_loadrandom.js
sudo docker-compose run k6 run -<ecs_rps.js
sudo docker-compose run k6 run -<ecs_spiketriangle.js
sudo docker-compose run k6 run -<ecs_stressincremental.js
sudo docker-compose run k6 run -<ecs_loadrandom.js
```

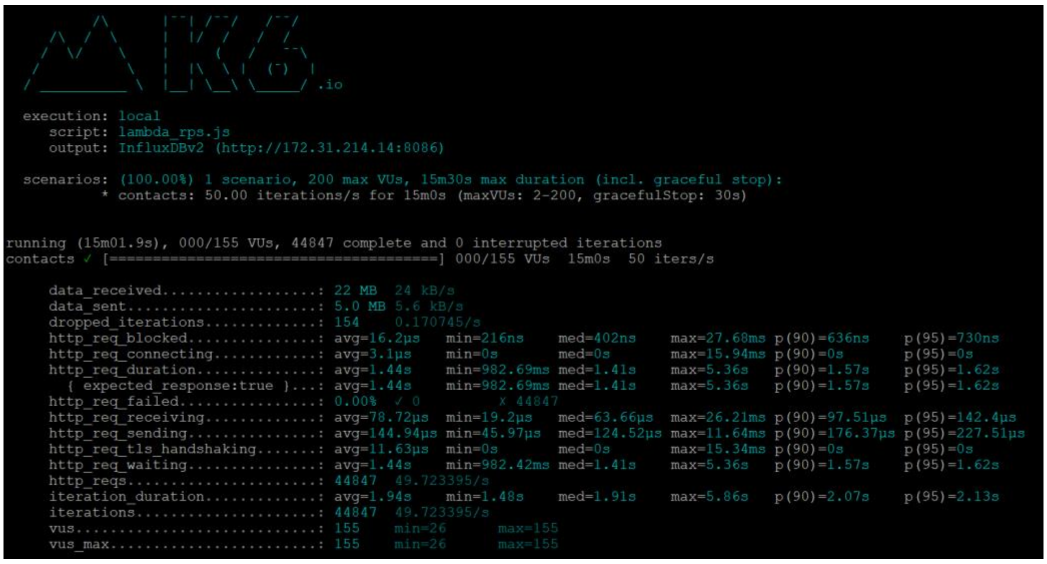
A.6 To run load test in Google

```
sudo docker-compose run k6 run -<gfunc_rps.js
sudo docker-compose run k6 run -<gfunc_spiketriangle.js
sudo docker-compose run k6 run -<gfunc_stressincremental.js
sudo docker-compose run k6 run -<gfunc_loadrandom.js
sudo docker-compose run k6 run -<gke_rps.js
sudo docker-compose run k6 run -<gke_spiketriangle.js
sudo docker-compose run k6 run -<gke_stressincremental.js
sudo docker-compose run k6 run -<gke_loadrandom.js
```

Appendix B

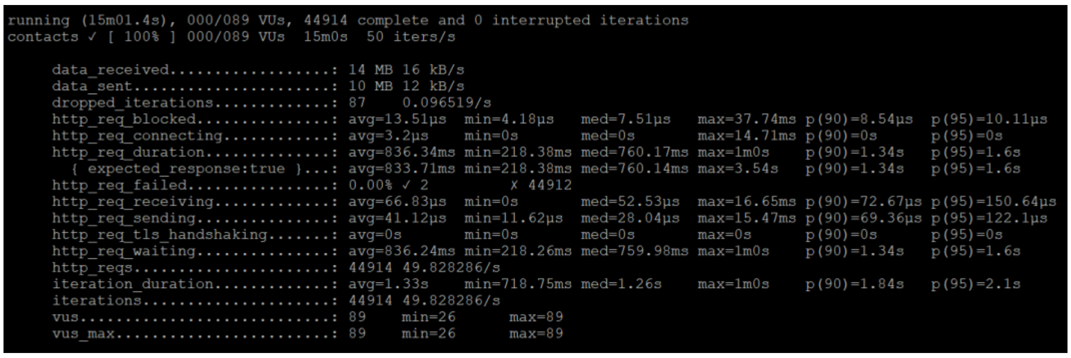
Load Test Results

B.1 Constant Load Test – AWS Lambda



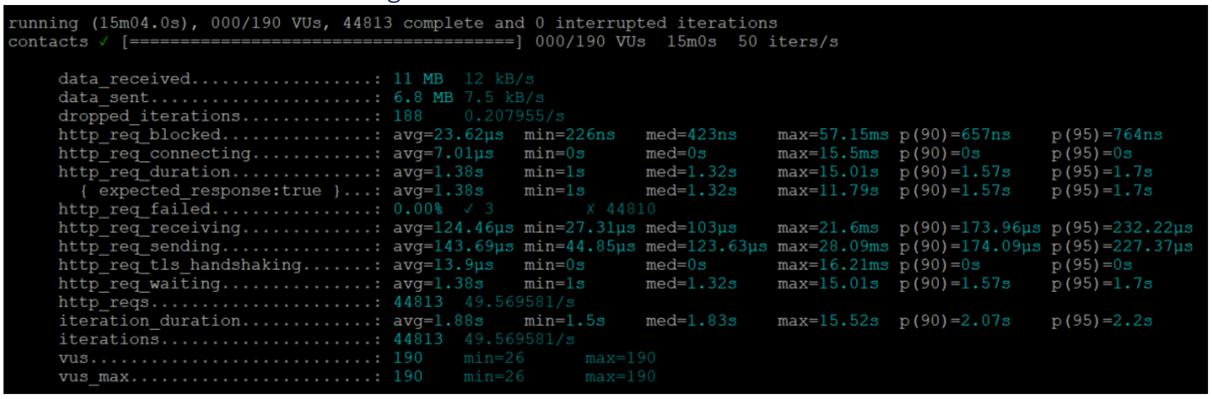
```
          /\      |‾‾| /‾‾/   /‾‾/
     /\  /  \     |  |/  /   /  /
    /  \/    \    |     (   /   ‾‾\
   /          \   |  |\  \ |  (‾)  |
  / __________ \  |__| \__\ \_____/ .io

  execution: local
     script: lambda_rps.js
     output: InfluxDBv2 (http://172.31.214.14:8086)

  scenarios: (100.00%) 1 scenario, 200 max VUs, 15m30s max duration (incl. graceful stop):
           * contacts: 50.00 iterations/s for 15m0s (maxVUs: 2-200, gracefulStop: 30s)


running (15m01.9s), 000/155 VUs, 44847 complete and 0 interrupted iterations
contacts ✓ [======================================] 000/155 VUs  15m0s  50 iters/s

     data_received..................: 22 MB  24 kB/s
     data_sent......................: 5.0 MB 5.6 kB/s
     dropped_iterations.............: 154    0.170745/s
     http_req_blocked...............: avg=16.2µs   min=216ns    med=402ns    max=27.68ms p(90)=636ns    p(95)=730ns
     http_req_connecting............: avg=3.1µs    min=0s       med=0s       max=15.94ms p(90)=0s       p(95)=0s
     http_req_duration..............: avg=1.44s    min=982.69ms med=1.41s    max=5.36s   p(90)=1.57s    p(95)=1.62s
       { expected_response:true }...: avg=1.44s    min=982.69ms med=1.41s    max=5.36s   p(90)=1.57s    p(95)=1.62s
     http_req_failed................: 0.00%  ✓ 0         ✗ 44847
     http_req_receiving.............: avg=78.72µs  min=19.2µs   med=63.66µs  max=26.21ms p(90)=97.51µs  p(95)=142.4µs
     http_req_sending...............: avg=144.94µs min=45.97µs  med=124.52µs max=11.64ms p(90)=176.37µs p(95)=227.51µs
     http_req_tls_handshaking.......: avg=11.63µs  min=0s       med=0s       max=15.34ms p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=1.44s    min=982.42ms med=1.41s    max=5.36s   p(90)=1.57s    p(95)=1.62s
     http_reqs......................: 44847  49.723395/s
     iteration_duration.............: avg=1.94s    min=1.48s    med=1.91s    max=5.86s   p(90)=2.07s    p(95)=2.13s
     iterations.....................: 44847  49.723395/s
     vus............................: 155    min=26      max=155
     vus_max........................: 155    min=26      max=155
```

B.2 Constant Load Test - AWS ECS



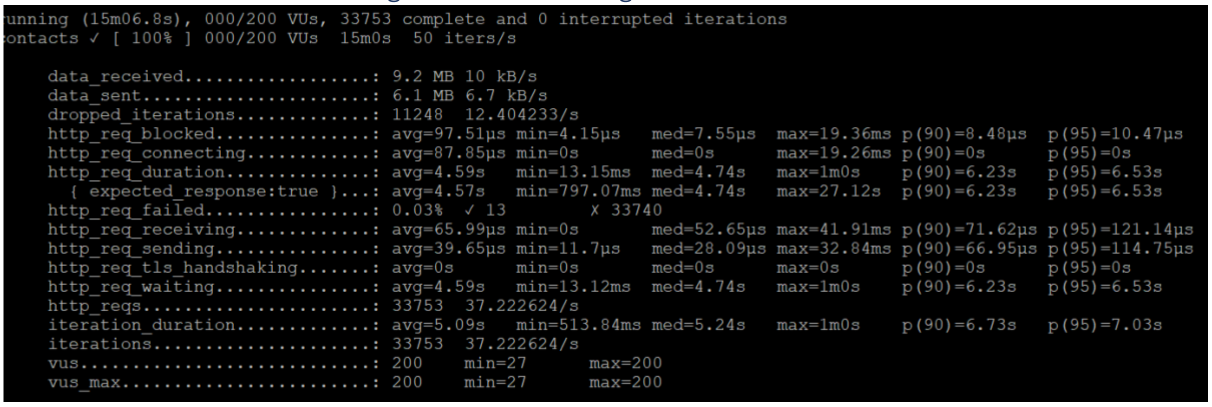
```
running (15m01.4s), 000/089 VUs, 44914 complete and 0 interrupted iterations
contacts ✓ [ 100% ] 000/089 VUs  15m0s  50 iters/s

     data_received..................: 14 MB 16 kB/s
     data_sent......................: 10 MB 12 kB/s
     dropped_iterations.............: 87    0.096519/s
     http_req_blocked...............: avg=13.51µs  min=4.18µs   med=7.51µs   max=37.74ms p(90)=8.54µs  p(95)=10.11µs
     http_req_connecting............: avg=3.2µs    min=0s       med=0s       max=14.71ms p(90)=0s      p(95)=0s
     http_req_duration..............: avg=836.34ms min=218.38ms med=760.17ms max=1m0s    p(90)=1.34s   p(95)=1.6s
       { expected_response:true }...: avg=833.71ms min=218.38ms med=760.14ms max=3.54s   p(90)=1.34s   p(95)=1.6s
     http_req_failed................: 0.00% ✓ 2          ✗ 44912
     http_req_receiving.............: avg=66.83µs  min=0s       med=52.53µs  max=16.65ms p(90)=72.67µs p(95)=150.64µs
     http_req_sending...............: avg=41.12µs  min=11.62µs  med=28.04µs  max=15.47ms p(90)=69.36µs p(95)=122.1µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s      p(90)=0s      p(95)=0s
     http_req_waiting...............: avg=836.24ms min=218.26ms med=759.98ms max=1m0s    p(90)=1.34s   p(95)=1.6s
     http_reqs......................: 44914 49.828286/s
     iteration_duration.............: avg=1.33s    min=718.75ms med=1.26s    max=1m0s    p(90)=1.84s   p(95)=2.1s
     iterations.....................: 44914 49.828286/s
     vus............................: 89    min=26      max=89
     vus_max........................: 89    min=26      max=89
```

B.3 Constant Load Test - Google Function



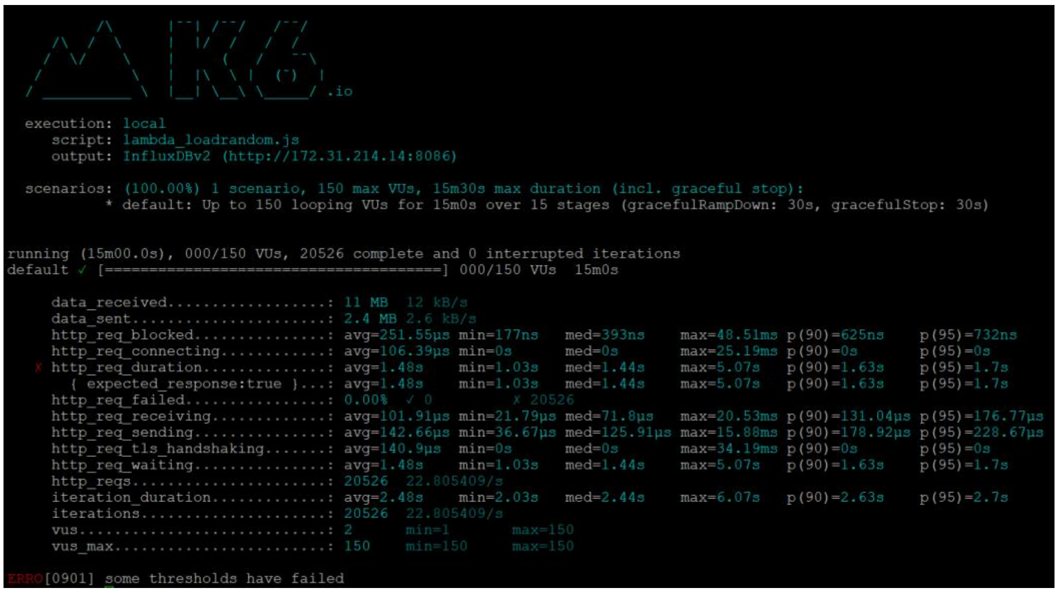
```
running (15m04.0s), 000/190 VUs, 44813 complete and 0 interrupted iterations
contacts ✓ [======================================] 000/190 VUs  15m0s  50 iters/s

     data_received..................: 11 MB  12 kB/s
     data_sent......................: 6.8 MB 7.5 kB/s
     dropped_iterations.............: 188    0.207955/s
     http_req_blocked...............: avg=23.62µs  min=226ns    med=423ns    max=57.15ms p(90)=657ns    p(95)=764ns
     http_req_connecting............: avg=7.01µs   min=0s       med=0s       max=15.5ms  p(90)=0s       p(95)=0s
     http_req_duration..............: avg=1.38s    min=1s       med=1.32s    max=15.01s  p(90)=1.57s    p(95)=1.7s
       { expected_response:true }...: avg=1.38s    min=1s       med=1.32s    max=11.79s  p(90)=1.57s    p(95)=1.7s
     http_req_failed................: 0.00%  ✓ 3         ✗ 44810
     http_req_receiving.............: avg=124.46µs min=27.31µs  med=103µs    max=21.6ms  p(90)=173.96µs p(95)=232.22µs
     http_req_sending...............: avg=143.69µs min=44.85µs  med=123.63µs max=28.09ms p(90)=174.09µs p(95)=227.37µs
     http_req_tls_handshaking.......: avg=13.9µs   min=0s       med=0s       max=16.21ms p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=1.38s    min=1s       med=1.32s    max=15.01s  p(90)=1.57s    p(95)=1.7s
     http_reqs......................: 44813  49.569581/s
     iteration_duration.............: avg=1.88s    min=1.5s     med=1.83s    max=15.52s  p(90)=2.07s    p(95)=2.2s
     iterations.....................: 44813  49.569581/s
     vus............................: 190    min=26      max=190
     vus_max........................: 190    min=26      max=190
```

B.4 Constant Load Test - Google Kubernetes Engine



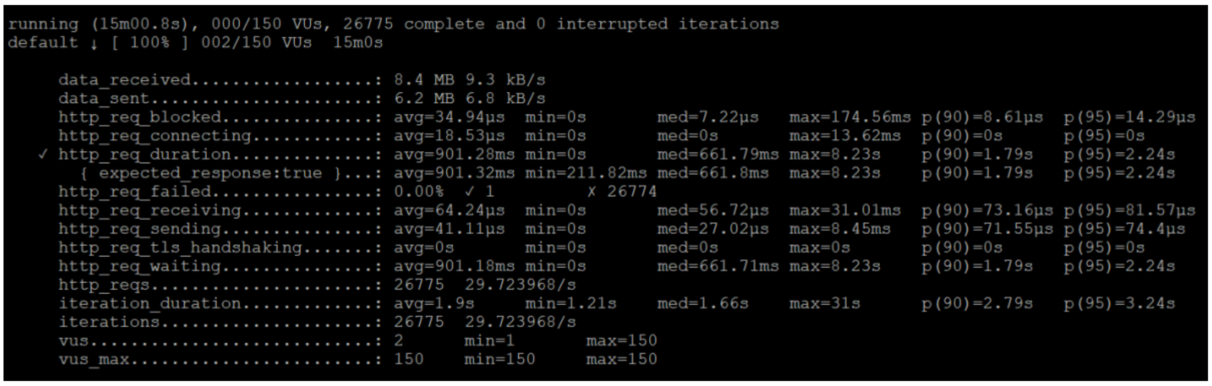
```
unning (15m06.8s), 000/200 VUs, 33753 complete and 0 interrupted iterations
ontacts ✓ [ 100% ] 000/200 VUs  15m0s  50 iters/s

     data_received..................: 9.2 MB 10 kB/s
     data_sent......................: 6.1 MB 6.7 kB/s
     dropped_iterations.............: 11248  12.404233/s
     http_req_blocked...............: avg=97.51µs min=4.15µs   med=7.55µs  max=19.36ms p(90)=8.48µs  p(95)=10.47µs
     http_req_connecting............: avg=87.85µs min=0s       med=0s      max=19.26ms p(90)=0s      p(95)=0s
     http_req_duration..............: avg=4.59s   min=13.15ms  med=4.74s   max=1m0s    p(90)=6.23s   p(95)=6.53s
       { expected_response:true }...: avg=4.57s   min=797.07ms med=4.74s   max=27.12s  p(90)=6.23s   p(95)=6.53s
     http_req_failed................: 0.03%  ✓ 13        ✗ 33740
     http_req_receiving.............: avg=65.99µs min=0s       med=52.65µs max=41.91ms p(90)=71.62µs p(95)=121.14µs
     http_req_sending...............: avg=39.65µs min=11.7µs   med=28.09µs max=32.84ms p(90)=66.95µs p(95)=114.75µs
     http_req_tls_handshaking.......: avg=0s      min=0s       med=0s      max=0s      p(90)=0s      p(95)=0s
     http_req_waiting...............: avg=4.59s   min=13.12ms  med=4.74s   max=1m0s    p(90)=6.23s   p(95)=6.53s
     http_reqs......................: 33753  37.222624/s
     iteration_duration.............: avg=5.09s   min=513.84ms med=5.24s   max=1m0s    p(90)=6.73s   p(95)=7.03s
     iterations.....................: 33753  37.222624/s
     vus............................: 200    min=27      max=200
     vus_max........................: 200    min=27      max=200
```

B.5 Random Load Test – AWS Lambda



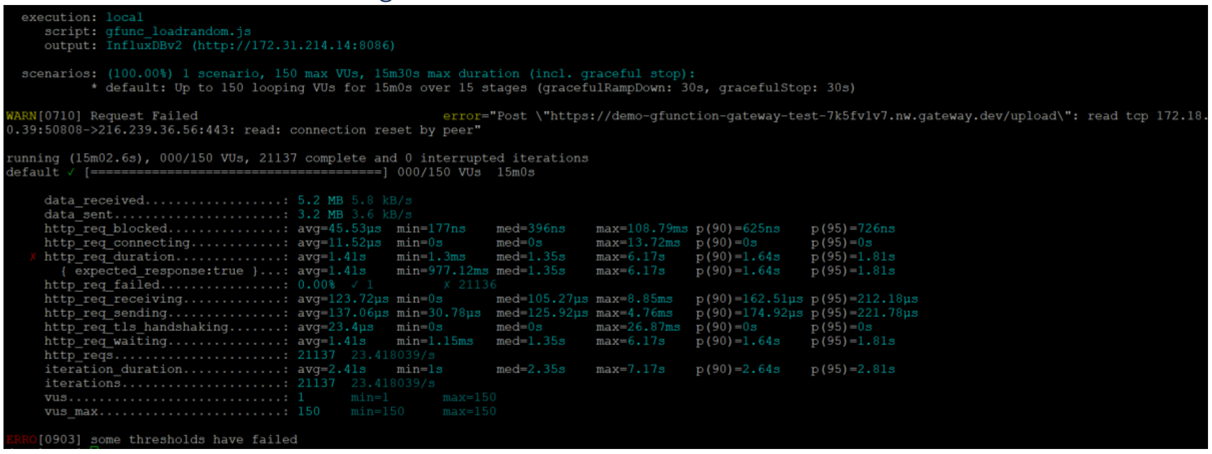
```
          /\      |‾‾| /‾‾/   /‾‾/
     /\  /  \     |  |/  /   /  /
    /  \/    \    |     (   /   ‾‾\
   /          \   |  |\  \ |  (‾)  |
  / __________ \  |__| \__\ \_____/ .io

  execution: local
     script: lambda_loadrandom.js
     output: InfluxDBv2 (http://172.31.214.14:8086)

  scenarios: (100.00%) 1 scenario, 150 max VUs, 15m30s max duration (incl. graceful stop):
           * default: Up to 150 looping VUs for 15m0s over 15 stages (gracefulRampDown: 30s, gracefulStop: 30s)


running (15m00.0s), 000/150 VUs, 20526 complete and 0 interrupted iterations
default ✓ [======================================] 000/150 VUs  15m0s

     data_received..................: 11 MB  12 kB/s
     data_sent......................: 2.4 MB 2.6 kB/s
     http_req_blocked...............: avg=251.55µs min=177ns   med=393ns    max=48.51ms p(90)=625ns    p(95)=732ns
     http_req_connecting............: avg=106.39µs min=0s      med=0s       max=25.19ms p(90)=0s       p(95)=0s
   ✗ http_req_duration..............: avg=1.48s    min=1.03s   med=1.44s    max=5.07s   p(90)=1.63s    p(95)=1.7s
       { expected_response:true }...: avg=1.48s    min=1.03s   med=1.44s    max=5.07s   p(90)=1.63s    p(95)=1.7s
     http_req_failed................: 0.00%  ✓ 0          ✗ 20526
     http_req_receiving.............: avg=101.91µs min=21.79µs med=71.8µs   max=20.53ms p(90)=131.04µs p(95)=176.77µs
     http_req_sending...............: avg=142.66µs min=36.67µs med=125.91µs max=15.88ms p(90)=178.92µs p(95)=228.67µs
     http_req_tls_handshaking.......: avg=140.9µs  min=0s      med=0s       max=34.19ms p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=1.48s    min=1.03s   med=1.44s    max=5.07s   p(90)=1.63s    p(95)=1.7s
     http_reqs......................: 20526  22.805409/s
     iteration_duration.............: avg=2.48s    min=2.03s   med=2.44s    max=6.07s   p(90)=2.63s    p(95)=2.7s
     iterations.....................: 20526  22.805409/s
     vus............................: 2      min=1        max=150
     vus_max........................: 150    min=150      max=150

ERRO[0901] some thresholds have failed
```

B.6 Random Load Test – AWS ECS



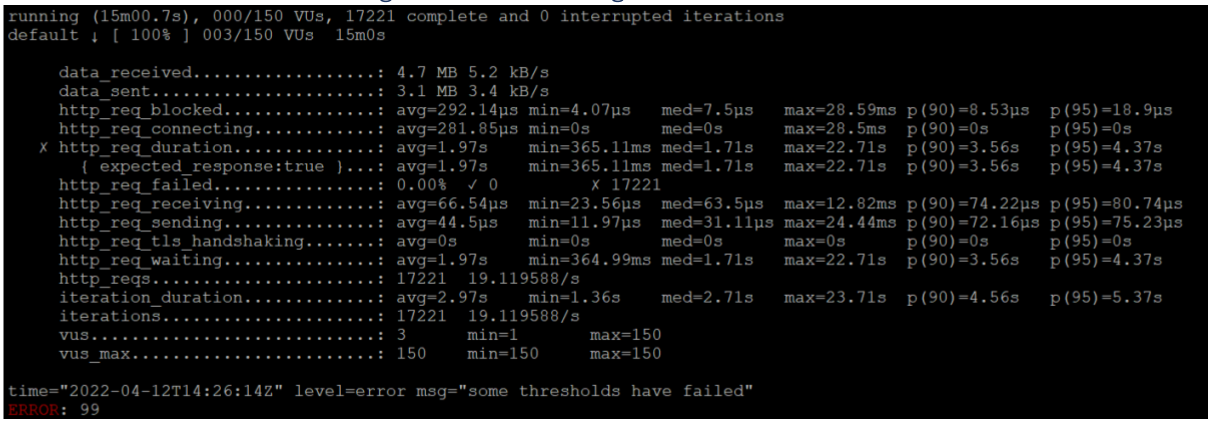
```
running (15m00.8s), 000/150 VUs, 26775 complete and 0 interrupted iterations
default ↓ [ 100% ] 002/150 VUs  15m0s

     data_received..................: 8.4 MB 9.3 kB/s
     data_sent......................: 6.2 MB 6.8 kB/s
     http_req_blocked...............: avg=34.94µs  min=0s       med=7.22µs   max=174.56ms p(90)=8.61µs  p(95)=14.29µs
     http_req_connecting............: avg=18.53µs  min=0s       med=0s       max=13.62ms  p(90)=0s      p(95)=0s
   ✓ http_req_duration..............: avg=901.28ms min=0s       med=661.79ms max=8.23s    p(90)=1.79s   p(95)=2.24s
       { expected_response:true }...: avg=901.32ms min=211.82ms med=661.8ms  max=8.23s    p(90)=1.79s   p(95)=2.24s
     http_req_failed................: 0.00%  ✓ 1         ✗ 26774
     http_req_receiving.............: avg=64.24µs  min=0s       med=56.72µs  max=31.01ms  p(90)=73.16µs p(95)=81.57µs
     http_req_sending...............: avg=41.11µs  min=0s       med=27.02µs  max=8.45ms   p(90)=71.55µs p(95)=74.4µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s      p(95)=0s
     http_req_waiting...............: avg=901.18ms min=0s       med=661.71ms max=8.23s    p(90)=1.79s   p(95)=2.24s
     http_reqs......................: 26775  29.723968/s
     iteration_duration.............: avg=1.9s     min=1.21s    med=1.66s    max=31s      p(90)=2.79s   p(95)=3.24s
     iterations.....................: 26775  29.723968/s
     vus............................: 2      min=1       max=150
     vus_max........................: 150    min=150     max=150
```

B.7 Random Load Test - Google Function



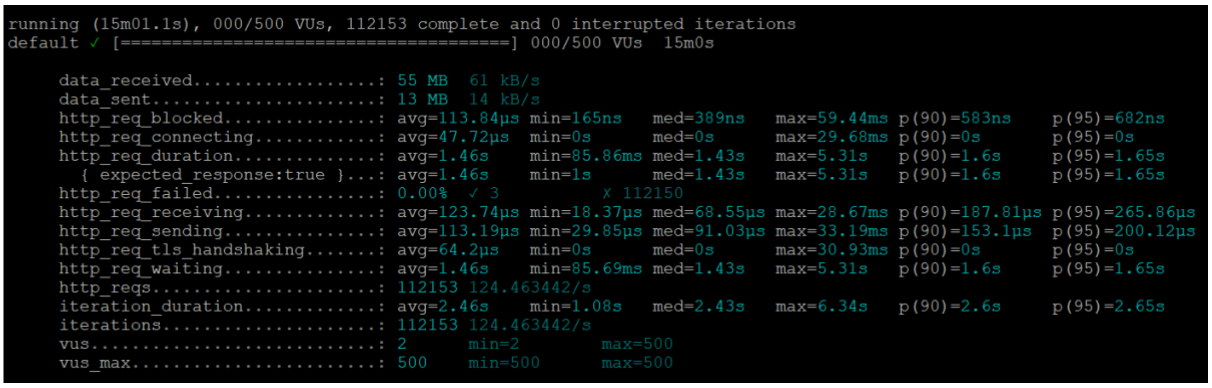
```
  execution: local
     script: gfunc_loadrandom.js
     output: InfluxDBv2 (http://172.31.214.14:8086)

  scenarios: (100.00%) 1 scenario, 150 max VUs, 15m30s max duration (incl. graceful stop):
           * default: Up to 150 looping VUs for 15m0s over 15 stages (gracefulRampDown: 30s, gracefulStop: 30s)

WARN[0710] Request Failed                                error="Post \"https://demo-gfunction-gateway-test-7k5fv1v7.nw.gateway.dev/upload\": read tcp 172.18.
0.39:50808->216.239.36.56:443: read: connection reset by peer"

running (15m02.6s), 000/150 VUs, 21137 complete and 0 interrupted iterations
default ✓ [======================================] 000/150 VUs  15m0s

     data_received..................: 5.2 MB 5.8 kB/s
     data_sent......................: 3.2 MB 3.6 kB/s
     http_req_blocked...............: avg=45.53µs  min=177ns    med=396ns    max=108.79ms p(90)=625ns    p(95)=726ns
     http_req_connecting............: avg=11.52µs  min=0s       med=0s       max=13.72ms  p(90)=0s       p(95)=0s
   ✗ http_req_duration..............: avg=1.41s    min=1.3ms    med=1.35s    max=6.17s    p(90)=1.64s    p(95)=1.81s
       { expected_response:true }...: avg=1.41s    min=977.12ms med=1.35s    max=6.17s    p(90)=1.64s    p(95)=1.81s
     http_req_failed................: 0.00%  ✓ 1          ✗ 21136
     http_req_receiving.............: avg=123.72µs min=0s       med=105.27µs max=8.85ms   p(90)=162.51µs p(95)=212.18µs
     http_req_sending...............: avg=137.06µs min=30.78µs  med=125.92µs max=4.76ms   p(90)=174.92µs p(95)=221.78µs
     http_req_tls_handshaking.......: avg=23.4µs   min=0s       med=0s       max=26.87ms  p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=1.41s    min=1.15ms   med=1.35s    max=6.17s    p(90)=1.64s    p(95)=1.81s
     http_reqs......................: 21137  23.418039/s
     iteration_duration.............: avg=2.41s    min=1s       med=2.35s    max=7.17s    p(90)=2.64s    p(95)=2.81s
     iterations.....................: 21137  23.418039/s
     vus............................: 1      min=1        max=150
     vus_max........................: 150    min=150      max=150

ERRO[0903] some thresholds have failed
```

B.8 Random Load Test - Google Kubernetes Engine



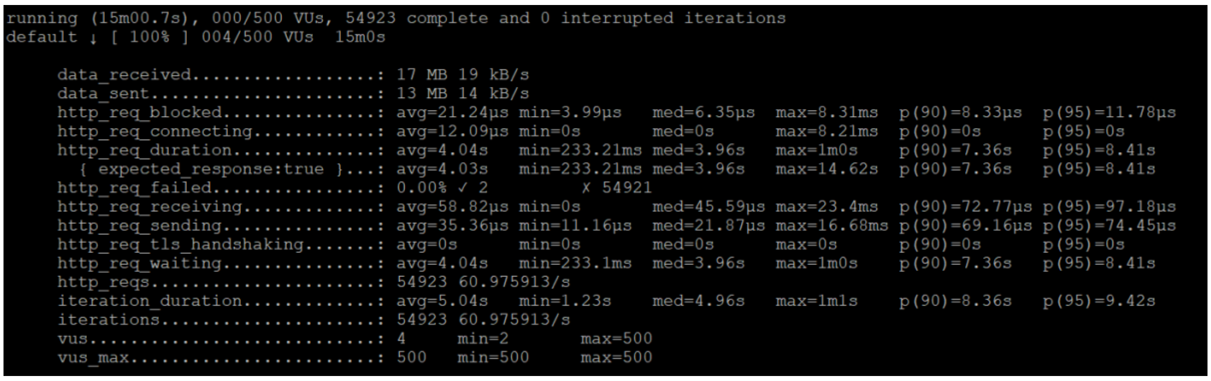
```
running (15m00.7s), 000/150 VUs, 17221 complete and 0 interrupted iterations
default ↓ [ 100% ] 003/150 VUs  15m0s

     data_received..................: 4.7 MB 5.2 kB/s
     data_sent......................: 3.1 MB 3.4 kB/s
     http_req_blocked...............: avg=292.14µs min=4.07µs   med=7.5µs    max=28.59ms p(90)=8.53µs  p(95)=18.9µs
     http_req_connecting............: avg=281.85µs min=0s       med=0s       max=28.5ms  p(90)=0s      p(95)=0s
   ✗ http_req_duration..............: avg=1.97s    min=365.11ms med=1.71s    max=22.71s  p(90)=3.56s   p(95)=4.37s
       { expected_response:true }...: avg=1.97s    min=365.11ms med=1.71s    max=22.71s  p(90)=3.56s   p(95)=4.37s
     http_req_failed................: 0.00%  ✓ 0          ✗ 17221
     http_req_receiving.............: avg=66.54µs  min=23.56µs  med=63.5µs   max=12.82ms p(90)=74.22µs p(95)=80.74µs
     http_req_sending...............: avg=44.5µs   min=11.97µs  med=31.11µs  max=24.44ms p(90)=72.16µs p(95)=75.23µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s      p(90)=0s      p(95)=0s
     http_req_waiting...............: avg=1.97s    min=364.99ms med=1.71s    max=22.71s  p(90)=3.56s   p(95)=4.37s
     http_reqs......................: 17221  19.119588/s
     iteration_duration.............: avg=2.97s    min=1.36s    med=2.71s    max=23.71s  p(90)=4.56s   p(95)=5.37s
     iterations.....................: 17221  19.119588/s
     vus............................: 3      min=1        max=150
     vus_max........................: 150    min=150      max=150

time="2022-04-12T14:26:14Z" level=error msg="some thresholds have failed"
ERROR: 99
```

B.9 Stress Incremental Load Test - AWS Lambda



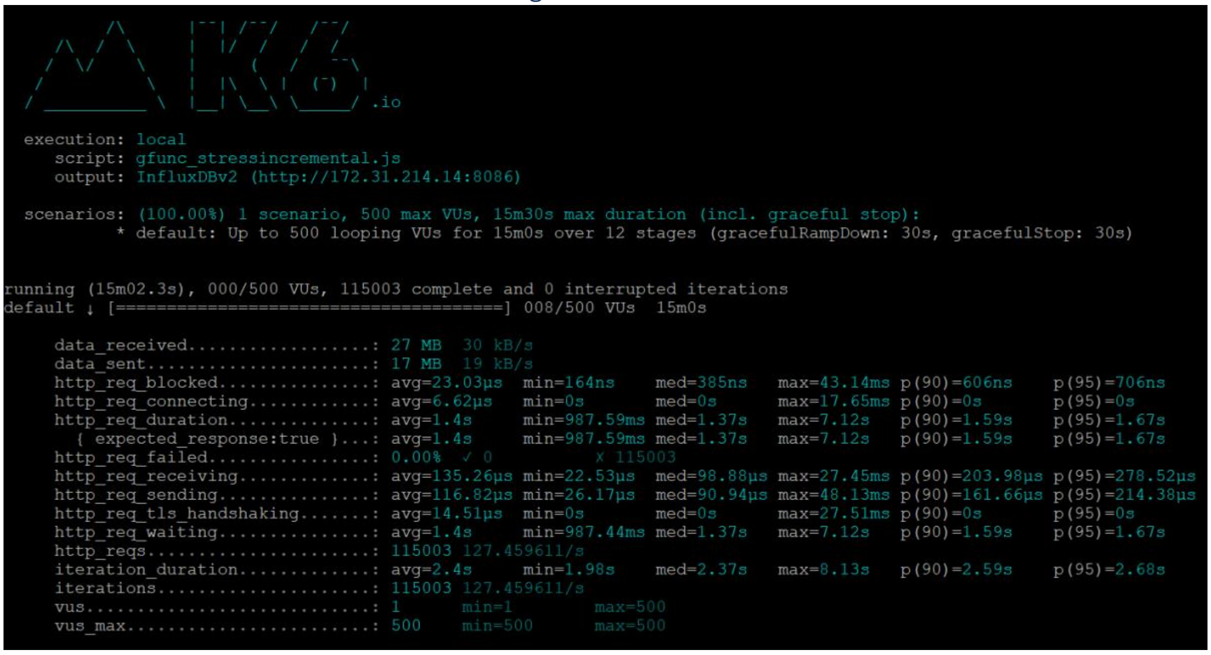
```
running (15m01.1s), 000/500 VUs, 112153 complete and 0 interrupted iterations
default ✓ [======================================] 000/500 VUs  15m0s

     data_received..................: 55 MB  61 kB/s
     data_sent......................: 13 MB  14 kB/s
     http_req_blocked...............: avg=113.84µs min=165ns   med=389ns   max=59.44ms p(90)=583ns    p(95)=682ns
     http_req_connecting............: avg=47.72µs  min=0s      med=0s      max=29.68ms p(90)=0s       p(95)=0s
     http_req_duration..............: avg=1.46s    min=85.86ms med=1.43s   max=5.31s   p(90)=1.6s     p(95)=1.65s
       { expected_response:true }...: avg=1.46s    min=1s      med=1.43s   max=5.31s   p(90)=1.6s     p(95)=1.65s
     http_req_failed................: 0.00%  ✓ 3          ✗ 112150
     http_req_receiving.............: avg=123.74µs min=18.37µs med=68.55µs max=28.67ms p(90)=187.81µs p(95)=265.86µs
     http_req_sending...............: avg=113.19µs min=29.85µs med=91.03µs max=33.19ms p(90)=153.1µs  p(95)=200.12µs
     http_req_tls_handshaking.......: avg=64.2µs   min=0s      med=0s      max=30.93ms p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=1.46s    min=85.69ms med=1.43s   max=5.31s   p(90)=1.6s     p(95)=1.65s
     http_reqs......................: 112153 124.463442/s
     iteration_duration.............: avg=2.46s    min=1.08s   med=2.43s   max=6.34s   p(90)=2.6s     p(95)=2.65s
     iterations.....................: 112153 124.463442/s
     vus............................: 2      min=2        max=500
     vus_max........................: 500    min=500      max=500
```

B.10 Stress Incremental Load Test - AWS ECS



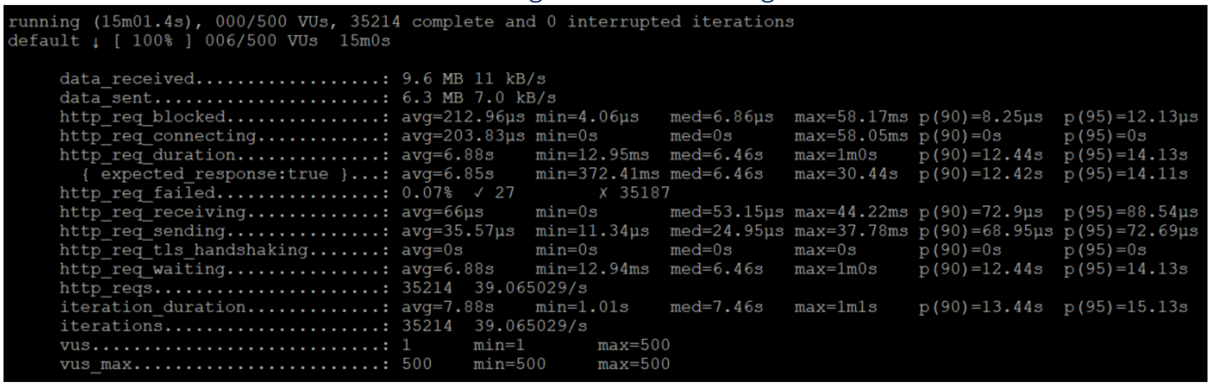
```
running (15m00.7s), 000/500 VUs, 54923 complete and 0 interrupted iterations
default ↓ [ 100% ] 004/500 VUs  15m0s

     data_received..................: 17 MB 19 kB/s
     data_sent......................: 13 MB 14 kB/s
     http_req_blocked...............: avg=21.24µs min=3.99µs   med=6.35µs  max=8.31ms  p(90)=8.33µs  p(95)=11.78µs
     http_req_connecting............: avg=12.09µs min=0s       med=0s      max=8.21ms  p(90)=0s      p(95)=0s
     http_req_duration..............: avg=4.04s   min=233.21ms med=3.96s   max=1m0s    p(90)=7.36s   p(95)=8.41s
       { expected_response:true }...: avg=4.03s   min=233.21ms med=3.96s   max=14.62s  p(90)=7.36s   p(95)=8.41s
     http_req_failed................: 0.00% ✓ 2         ✗ 54921
     http_req_receiving.............: avg=58.82µs min=0s       med=45.59µs max=23.4ms  p(90)=72.77µs p(95)=97.18µs
     http_req_sending...............: avg=35.36µs min=11.16µs  med=21.87µs max=16.68ms p(90)=69.16µs p(95)=74.45µs
     http_req_tls_handshaking.......: avg=0s      min=0s       med=0s      max=0s      p(90)=0s      p(95)=0s
     http_req_waiting...............: avg=4.04s   min=233.1ms  med=3.96s   max=1m0s    p(90)=7.36s   p(95)=8.41s
     http_reqs......................: 54923 60.975913/s
     iteration_duration.............: avg=5.04s   min=1.23s    med=4.96s   max=1m1s    p(90)=8.36s   p(95)=9.42s
     iterations.....................: 54923 60.975913/s
     vus............................: 4     min=2       max=500
     vus_max........................: 500   min=500     max=500
```

B.11 Stress Incremental Load Test - Google Function



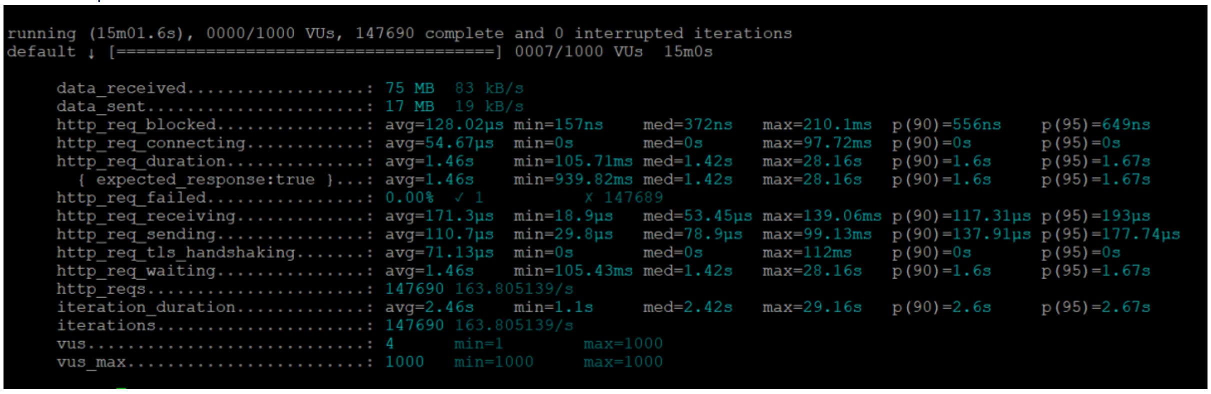
```
          /\      |‾‾| /‾‾/   /‾‾/
     /\  /  \     |  |/  /   /  /
    /  \/    \    |     (   /   ‾‾\
   /          \   |  |\  \ |  (‾)  |
  / __________ \  |__| \__\ \_____/ .io

  execution: local
     script: gfunc_stressincremental.js
     output: InfluxDBv2 (http://172.31.214.14:8086)

  scenarios: (100.00%) 1 scenario, 500 max VUs, 15m30s max duration (incl. graceful stop):
           * default: Up to 500 looping VUs for 15m0s over 12 stages (gracefulRampDown: 30s, gracefulStop: 30s)


running (15m02.3s), 000/500 VUs, 115003 complete and 0 interrupted iterations
default ↓ [======================================] 008/500 VUs  15m0s

     data_received..................: 27 MB  30 kB/s
     data_sent......................: 17 MB  19 kB/s
     http_req_blocked...............: avg=23.03µs  min=164ns    med=385ns   max=43.14ms p(90)=606ns    p(95)=706ns
     http_req_connecting............: avg=6.62µs   min=0s       med=0s      max=17.65ms p(90)=0s       p(95)=0s
     http_req_duration..............: avg=1.4s     min=987.59ms med=1.37s   max=7.12s   p(90)=1.59s    p(95)=1.67s
       { expected_response:true }...: avg=1.4s     min=987.59ms med=1.37s   max=7.12s   p(90)=1.59s    p(95)=1.67s
     http_req_failed................: 0.00%  ✓ 0          ✗ 115003
     http_req_receiving.............: avg=135.26µs min=22.53µs  med=98.88µs max=27.45ms p(90)=203.98µs p(95)=278.52µs
     http_req_sending...............: avg=116.82µs min=26.17µs  med=90.94µs max=48.13ms p(90)=161.66µs p(95)=214.38µs
     http_req_tls_handshaking.......: avg=14.51µs  min=0s       med=0s      max=27.51ms p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=1.4s     min=987.44ms med=1.37s   max=7.12s   p(90)=1.59s    p(95)=1.67s
     http_reqs......................: 115003 127.459611/s
     iteration_duration.............: avg=2.4s     min=1.98s    med=2.37s   max=8.13s   p(90)=2.59s    p(95)=2.68s
     iterations.....................: 115003 127.459611/s
     vus............................: 1      min=1        max=500
     vus_max........................: 500    min=500      max=500
```

B.12 Stress Incremental Load Test - Google Kubernetes Engine



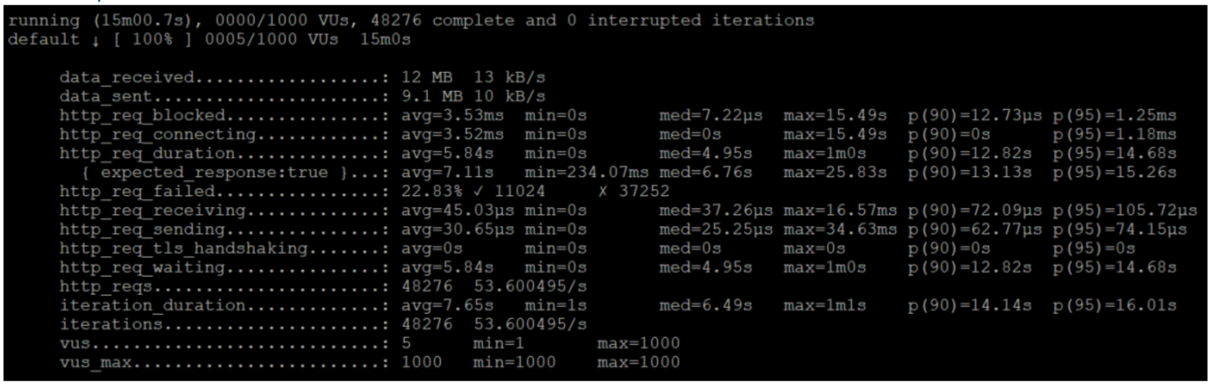
```
running (15m01.4s), 000/500 VUs, 35214 complete and 0 interrupted iterations
default ↓ [ 100% ] 006/500 VUs  15m0s

     data_received..................: 9.6 MB 11 kB/s
     data_sent......................: 6.3 MB 7.0 kB/s
     http_req_blocked...............: avg=212.96µs min=4.06µs   med=6.86µs  max=58.17ms p(90)=8.25µs  p(95)=12.13µs
     http_req_connecting............: avg=203.83µs min=0s       med=0s      max=58.05ms p(90)=0s      p(95)=0s
     http_req_duration..............: avg=6.88s    min=12.95ms  med=6.46s   max=1m0s    p(90)=12.44s  p(95)=14.13s
       { expected_response:true }...: avg=6.85s    min=372.41ms med=6.46s   max=30.44s  p(90)=12.42s  p(95)=14.11s
     http_req_failed................: 0.07%  ✓ 27       ✗ 35187
     http_req_receiving.............: avg=66µs     min=0s       med=53.15µs max=44.22ms p(90)=72.9µs  p(95)=88.54µs
     http_req_sending...............: avg=35.57µs  min=11.34µs  med=24.95µs max=37.78ms p(90)=68.95µs p(95)=72.69µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s      max=0s      p(90)=0s      p(95)=0s
     http_req_waiting...............: avg=6.88s    min=12.94ms  med=6.46s   max=1m0s    p(90)=12.44s  p(95)=14.13s
     http_reqs......................: 35214  39.065029/s
     iteration_duration.............: avg=7.88s    min=1.01s    med=7.46s   max=1m1s    p(90)=13.44s  p(95)=15.13s
     iterations.....................: 35214  39.065029/s
     vus............................: 1      min=1      max=500
     vus_max........................: 500    min=500    max=500
```

B.13 Spike Load Test – AWS Lambda



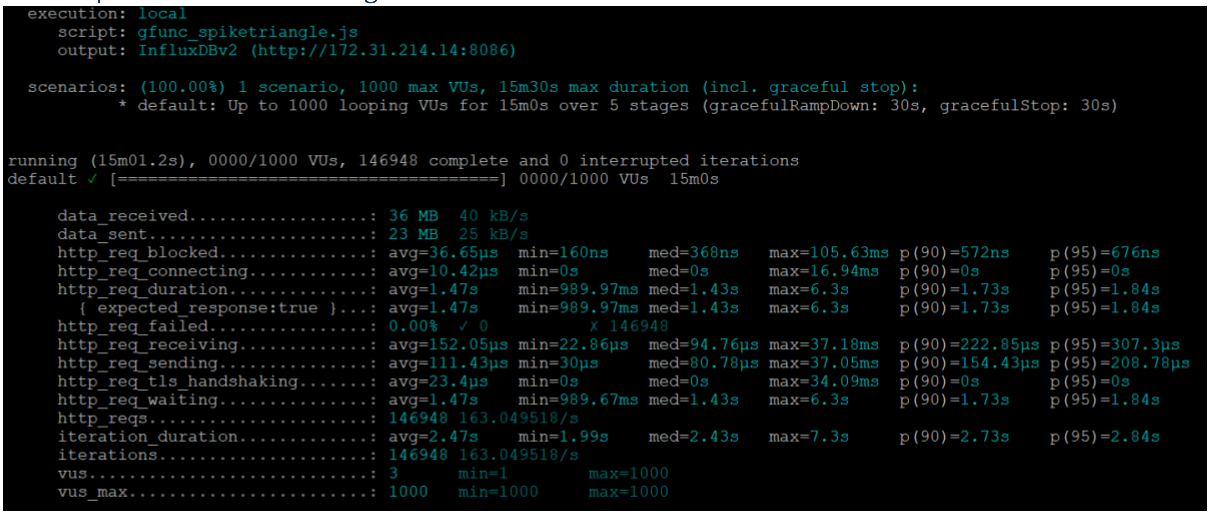
```
running (15m01.6s), 0000/1000 VUs, 147690 complete and 0 interrupted iterations
default ↓ [======================================] 0007/1000 VUs  15m0s

     data_received..................: 75 MB  83 kB/s
     data_sent......................: 17 MB  19 kB/s
     http_req_blocked...............: avg=128.02µs min=157ns    med=372ns   max=210.1ms  p(90)=556ns    p(95)=649ns
     http_req_connecting............: avg=54.67µs  min=0s       med=0s      max=97.72ms  p(90)=0s       p(95)=0s
     http_req_duration..............: avg=1.46s    min=105.71ms med=1.42s   max=28.16s   p(90)=1.6s     p(95)=1.67s
       { expected_response:true }...: avg=1.46s    min=939.82ms med=1.42s   max=28.16s   p(90)=1.6s     p(95)=1.67s
     http_req_failed................: 0.00%   ✓ 1          ✗ 147689
     http_req_receiving.............: avg=171.3µs  min=18.9µs   med=53.45µs max=139.06ms p(90)=117.31µs p(95)=193µs
     http_req_sending...............: avg=110.7µs  min=29.8µs   med=78.9µs  max=99.13ms  p(90)=137.91µs p(95)=177.74µs
     http_req_tls_handshaking.......: avg=71.13µs  min=0s       med=0s      max=112ms    p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=1.46s    min=105.43ms med=1.42s   max=28.16s   p(90)=1.6s     p(95)=1.67s
     http_reqs......................: 147690 163.805139/s
     iteration_duration.............: avg=2.46s    min=1.1s     med=2.42s   max=29.16s   p(90)=2.6s     p(95)=2.67s
     iterations.....................: 147690 163.805139/s
     vus............................: 4      min=1        max=1000
     vus_max........................: 1000   min=1000     max=1000
```

B.14 Spike Load Test – AWS ECS



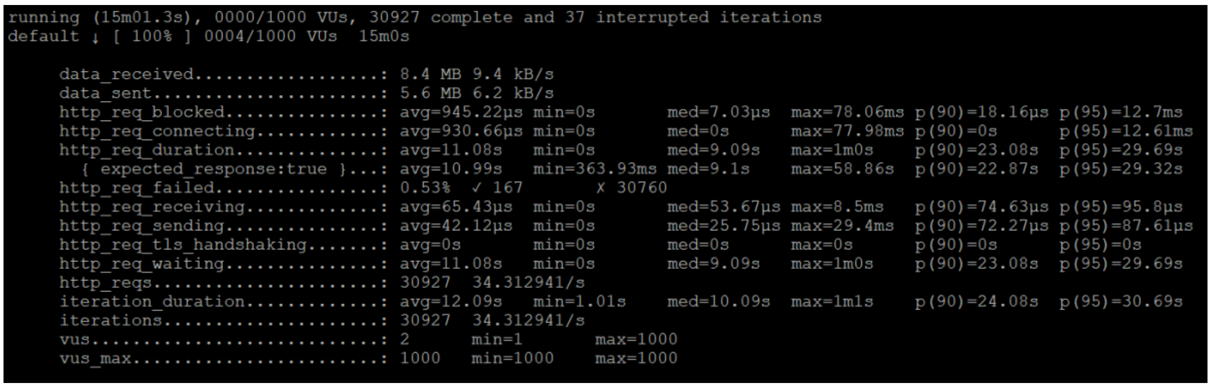
```
running (15m00.7s), 0000/1000 VUs, 48276 complete and 0 interrupted iterations
default ↓ [ 100% ] 0005/1000 VUs  15m0s

     data_received..................: 12 MB  13 kB/s
     data_sent......................: 9.1 MB 10 kB/s
     http_req_blocked...............: avg=3.53ms   min=0s       med=7.22µs  max=15.49s   p(90)=12.73µs p(95)=1.25ms
     http_req_connecting............: avg=3.52ms   min=0s       med=0s      max=15.49s   p(90)=0s      p(95)=1.18ms
     http_req_duration..............: avg=5.84s    min=0s       med=4.95s   max=1m0s     p(90)=12.82s  p(95)=14.68s
       { expected_response:true }...: avg=7.11s    min=234.07ms med=6.76s   max=25.83s   p(90)=13.13s  p(95)=15.26s
     http_req_failed................: 22.83% ✓ 11024      ✗ 37252
     http_req_receiving.............: avg=45.03µs  min=0s       med=37.26µs max=16.57ms  p(90)=72.09µs p(95)=105.72µs
     http_req_sending...............: avg=30.65µs  min=0s       med=25.25µs max=34.63ms  p(90)=62.77µs p(95)=74.15µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s      max=0s       p(90)=0s      p(95)=0s
     http_req_waiting...............: avg=5.84s    min=0s       med=4.95s   max=1m0s     p(90)=12.82s  p(95)=14.68s
     http_reqs......................: 48276  53.600495/s
     iteration_duration.............: avg=7.65s    min=1s       med=6.49s   max=1m1s     p(90)=14.14s  p(95)=16.01s
     iterations.....................: 48276  53.600495/s
     vus............................: 5      min=1        max=1000
     vus_max........................: 1000   min=1000     max=1000
```

B.15 Spike Load Test – Google Function



```
  execution: local
     script: gfunc_spiketriangle.js
     output: InfluxDBv2 (http://172.31.214.14:8086)

  scenarios: (100.00%) 1 scenario, 1000 max VUs, 15m30s max duration (incl. graceful stop):
           * default: Up to 1000 looping VUs for 15m0s over 5 stages (gracefulRampDown: 30s, gracefulStop: 30s)


running (15m01.2s), 0000/1000 VUs, 146948 complete and 0 interrupted iterations
default ✓ [======================================] 0000/1000 VUs  15m0s

     data_received..................: 36 MB  40 kB/s
     data_sent......................: 23 MB  25 kB/s
     http_req_blocked...............: avg=36.65µs  min=160ns    med=368ns   max=105.63ms p(90)=572ns    p(95)=676ns
     http_req_connecting............: avg=10.42µs  min=0s       med=0s      max=16.94ms  p(90)=0s       p(95)=0s
     http_req_duration..............: avg=1.47s    min=989.97ms med=1.43s   max=6.3s     p(90)=1.73s    p(95)=1.84s
       { expected_response:true }...: avg=1.47s    min=989.97ms med=1.43s   max=6.3s     p(90)=1.73s    p(95)=1.84s
     http_req_failed................: 0.00%   ✓ 0          ✗ 146948
     http_req_receiving.............: avg=152.05µs min=22.86µs  med=94.76µs max=37.18ms  p(90)=222.85µs p(95)=307.3µs
     http_req_sending...............: avg=111.43µs min=30µs     med=80.78µs max=37.05ms  p(90)=154.43µs p(95)=208.78µs
     http_req_tls_handshaking.......: avg=23.4µs   min=0s       med=0s      max=34.09ms  p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=1.47s    min=989.67ms med=1.43s   max=6.3s     p(90)=1.73s    p(95)=1.84s
     http_reqs......................: 146948 163.049518/s
     iteration_duration.............: avg=2.47s    min=1.99s    med=2.43s   max=7.3s     p(90)=2.73s    p(95)=2.84s
     iterations.....................: 146948 163.049518/s
     vus............................: 3      min=1        max=1000
     vus_max........................: 1000   min=1000     max=1000
```

B.16 Spike Load Test – Google Kubernetes Engine



```
running (15m01.3s), 0000/1000 VUs, 30927 complete and 37 interrupted iterations
default ↓ [ 100% ] 0004/1000 VUs  15m0s

     data_received..................: 8.4 MB 9.4 kB/s
     data_sent......................: 5.6 MB 6.2 kB/s
     http_req_blocked...............: avg=945.22µs min=0s       med=7.03µs  max=78.06ms p(90)=18.16µs p(95)=12.7ms
     http_req_connecting............: avg=930.66µs min=0s       med=0s      max=77.98ms p(90)=0s      p(95)=12.61ms
     http_req_duration..............: avg=11.08s   min=0s       med=9.09s   max=1m0s    p(90)=23.08s  p(95)=29.69s
       { expected_response:true }...: avg=10.99s   min=363.93ms med=9.1s    max=58.86s  p(90)=22.87s  p(95)=29.32s
     http_req_failed................: 0.53%  ✓ 167        ✗ 30760
     http_req_receiving.............: avg=65.43µs  min=0s       med=53.67µs max=8.5ms   p(90)=74.63µs p(95)=95.8µs
     http_req_sending...............: avg=42.12µs  min=0s       med=25.75µs max=29.4ms  p(90)=72.27µs p(95)=87.61µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s      max=0s      p(90)=0s      p(95)=0s
     http_req_waiting...............: avg=11.08s   min=0s       med=9.09s   max=1m0s    p(90)=23.08s  p(95)=29.69s
     http_reqs......................: 30927  34.312941/s
     iteration_duration.............: avg=12.09s   min=1.01s    med=10.09s  max=1m1s    p(90)=24.08s  p(95)=30.69s
     iterations.....................: 30927  34.312941/s
     vus............................: 2      min=1        max=1000
     vus_max........................: 1000   min=1000     max=1000
```

Appendix C

Application Source Code

C.1 Microservice - AWS ECS

```
'use strict';

const serverless = require('serverless-http');
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const app = express();
const Jimp = require('jimp');
const { v4: uuidv4 } = require("uuid");
const height = 500;
var imageType = "image/png";
require('dotenv').config();
const bucket = process.env.Bucket;
const port = process.env.Port;
const AWS = require('aws-sdk');
const s3 = new AWS.S3();
var params = {
    Bucket: bucket
};

// Cors
app.use(cors());

// Data Parsing
app.use(bodyParser.json({ extended: true }));
app.use(bodyParser.urlencoded({ extended: true }));

// here you send the URL image
app.post('/dev/upload', async (req, res, cb) => {

    let path = req.body.photoUrl; // URL Image

    console.log(path);

    let photoUrl = path;
    let objectId = uuidv4();
    let objectKey = `rz-${height}-${objectId}.`;

    return fetchImage(photoUrl)
        .then(image => {
            imageType= image.getMIME();

            if(imageType == 'image/png'){
                objectKey= objectKey+'png';
            } else if(imageType == 'image/jpg') {
                objectKey= objectKey+'jpg';
            } else if(imageType == 'image/jpeg') {
                objectKey= objectKey+'jpeg';
            }
            image.resize(Jimp.AUTO, height);
            return image.getBufferAsync(image.getMIME());
        })
        .then(resizedBuffer => uploadToS3(resizedBuffer, objectKey,
imageType))
        .then(function (response) {
            console.log(`Image ${objectKey} was upload and resized`);
            res.status(200).json(response);
        })
        .catch(error => console.log(error));

})

/**
 * @param {*} data
 */
function uploadToS3(data, key, imageType ) {
    console.log('uploadToS3');
    return s3
        .putObject({
            Bucket: bucket,
            Key: key,
            Body: data,
            ContentType: imageType
        })
        .promise();
}

/**
 * @param {url}
 * @returns {Promise}
 */
function fetchImage(url) {
    return Jimp.read(url);
}

app.listen(port, () => {
    console.log(`App listening at http://localhost:${port}`)
})
```

C.2 Microservice - Google Kubernetes Engine

```
'use strict';

const serverless = require('serverless-http');
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const app = express();
const Jimp = require('jimp');
const { v4: uuidv4 } = require("uuid");
const height = 500;
var imageType = "image/png";
```

```
require('dotenv').config();
const port = process.env.Port;
const { Storage } = require('@google-cloud/storage');
const bucket = process.env.Bucket;
const project_id = 'robust-resource-12345';//process.env.Bucket;



// Cors
app.use(cors());

// Data Parsing
app.use(bodyParser.json({ extended: true }));
app.use(bodyParser.urlencoded({ extended: true }));

app.get('/', async (req, res) => {
    const healthcheck = {
        uptime: process.uptime(),
        message: 'OK',
        timestamp: Date.now()
    };

    res.status(200).send(healthcheck);
});

app.get('/health', (req, res) => {
    const healthcheck = {
        uptime: process.uptime(),
        message: 'OK',
        timestamp: Date.now()
    };

    res.status(200).send(healthcheck);
});

// here you send the URL image
app.post('/upload', async (req, res, cb) => {

    let path = req.body.photoUrl; // URL Image

    console.log(path);

    let photoUrl = path;
    let objectId = uuidv4();
    let objectKey = `rz-${height}-${objectId}.`;

    return fetchImage(photoUrl)
        .then(image => {
            imageType = image.getMIME();

            if (imageType == 'image/png') {
                objectKey = objectKey + 'png';
            } else if (imageType == 'image/jpg') {
                objectKey = objectKey + 'jpg';
            } else if (imageType == 'image/jpeg') {
                objectKey = objectKey + 'jpeg';
            }
            image.resize(Jimp.AUTO, height);
            return image.getBufferAsync(image.getMIME());
        })
        .then(resizedBuffer => uploadToGoogle(resizedBuffer,
objectKey, imageType))
        .then(function (response) {
            console.log(`Image ${objectKey} was upload and resized`);
            res.status(200).json(response);
        })
        .catch(error => console.log(error));

})

function uploadToGoogle(data, key, imageType) {
    return new Promise(function (resolve, reject) {
        console.log('uploadToS3');
        const storage = new Storage({
            projectId: project_id,
            keyFilename: '826719257276.json'
        });
        const googlebucket = storage.bucket('oslometimagev1');
        const bucketfile = googlebucket.file(key);


        // Uploads the file.
        bucketfile.save(data, { contentType: imageType }).then(() =>
{
            // Success handling...
            console.log('success save');
            resolve("success");
        }).catch(error => {
            console.log('error');
            reject(error);
            // Error handling...
        });
        console.log("returning success");
    });
}

function fetchImage(url) {
    return Jimp.read(url);
}

app.listen(port, () => {
    console.log(`App listening at http://localhost:${port}`)
})
```

C.3 Serverless - AWS Lambda

```
"use strict";
const AWS = require("aws-sdk");
const { v4: uuidv4 } = require("uuid");
const Jimp = require("jimp");

const s3 = new AWS.S3();
const height = 500;
var imageType = "image/png";
const bucket = "oslometx2";

module.exports.handler = (event, context, callback) => {
  let requestBody = JSON.parse(event.body);
  let photoUrl = requestBody.photoUrl;
  let objectId = uuidv4();
  let objectKey = `rz-${height}-${objectId}.`;

  fetchImage(photoUrl)
  .then(image => {
      imageType= image.getMIME();

      if(imageType == 'image/png'){
          objectKey= objectKey+'png';
      } else if(imageType == 'image/jpg') {
          objectKey= objectKey+'jpg';
      } else if(imageType == 'image/jpeg') {
          objectKey= objectKey+'jpeg';
      }
      image.resize(Jimp.AUTO, height);
      return image.getBufferAsync(image.getMIME());
  })
  .then(resizedBuffer => uploadToS3(resizedBuffer, objectKey,
imageType))
  .then(function (response) {
    callback(null, {
      statusCode: 200,
      body: JSON.stringify(response)
    });
  })
  .catch(error => console.log(error));

};
function uploadToS3(data, key) {
  return s3
    .putObject({
      Bucket: bucket,
      Key: key,
      Body: data,
      ContentType: imageType
    })
    .promise();
}
function fetchImage(url) {
  return Jimp.read(url);
}
```

```


```

C.4 Google Cloud Functions

```
const { v4: uuidv4 } = require("uuid");
const Jimp = require("jimp");

const { Storage } = require('@google-cloud/storage');
// Instantiate a storage client
//const storage = new Storage();

const height = 500;
var imageType = "image/png";
const bucket = 'oslometx2';//process.env.Bucket;
const project_id = 'robust-resource-12345';//process.env.Bucket;

exports.uploadHandler = (req, res) => {
  console.log('handler', req.method);
  console.log('imageUrl:', req.body.photoUrl);
  //let requestBody = JSON.parse(req.body);
  let photoUrl = req.body.photoUrl;
  let objectId = uuidv4();
  let objectKey = `rz-${height}-${objectId}.`;


  if(photoUrl.length === 0) {
    return res.status(500).send({
        error: "empty value"
    });
  }

  fetchImage(photoUrl)
    .then(image => {
      imageType = image.getMIME();

      if (imageType == 'image/png') {
        objectKey = objectKey + 'png';
      } else if (imageType == 'image/jpg') {
        objectKey = objectKey + 'jpg';
      } else if (imageType == 'image/jpeg') {
        objectKey = objectKey + 'jpeg';
      }
      image.resize(Jimp.AUTO, height);
      return image.getBufferAsync(image.getMIME());
    })
    .then(resizedBuffer => uploadToGoogle(resizedBuffer, objectKey,
imageType))
    .then(function (response) {
      console.log('response:', response);
      res.status(200).send('Ok');
    })
    .catch(error => console.log(error));
};
```

```

function uploadToGoogle(data, key, imageType) {
  return new Promise(function (resolve, reject) {
    console.log('uploadToGoogle');
    const storage = new Storage({
      projectId: project_id,
      keyFilename: '826719257276.json'
    });
    const googlebucket = storage.bucket('oslometx2');
    const bucketfile = googlebucket.file(key);


    // Uploads the file.
    bucketfile.save(data,{contentType: imageType}).then(() => {
      // Success handling...
      console.log('success save');
      resolve("success");
    }).catch(error => {
      console.log('error');
      reject(error);
      // Error handling...
    });
    console.log("returning success");
  });
}
function fetchImage(url) {
  return Jimp.read(url);
}
```

Appendix D

Serverless & Microservice Deployment

D.1 AWS Lambda

```
service: demo-lambda

custom:
  bucket: oslometx

provider:
  name: aws
  runtime: nodejs12.x
  region: eu-west-2
  stackName: imageUploader
  apiGateway:
    restApiId:
      "Fn::ImportValue": SharedGW-restApiId
    restApiRootResourceId:
      "Fn::ImportValue": SharedGW-rootResourceId
  iamRoleStatements:
    - Effect: "Allow"
      Action:
        - "s3:PutObject"
        - "s3:GetObject"
        - "s3:ListBucket"
        - "s3:DeleteObject"
        - "s3:PutObjectAcl"
        - "s3:CreateObject"
        - "s3:ListObjects"
      Resource:
        - "arn:aws:s3:::${self:custom.bucket}"
        - "arn:aws:s3:::${self:custom.bucket}/*"

functions:
  UploadImage:
    handler: uploadImage.handler
    # The `events` block defines how to trigger the
uploadImage.handler code
    events:
      - http:
          path: upload
          method: post
          cors: true
    environment:
      Bucket: ${self:custom.bucket}

resources:
  Resources:
    StorageBucket:
      Type: "AWS::S3::Bucket"
      Properties:
        BucketName: ${self:custom.bucket}
```

D.2 Google Functions

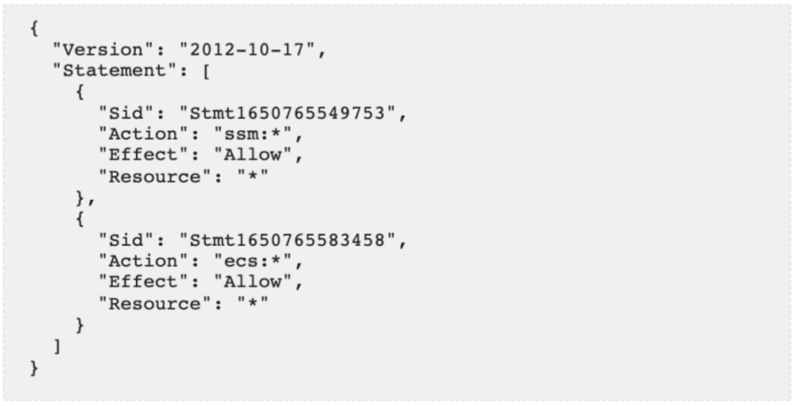
```
service: demo-gfunction

custom:
  scripts:
        commands:
            make-public-function: gcloud functions add-iam-policy-
binding ${self:service}-${self:provider.stage}-${opt:opt.function,
"functionName"} --member="allUsers" --
role="roles/cloudfunctions.invoker" --project=${self:provider.project}
--region=${self:provider.region} | xargs echo

        hooks:
            'after:deploy:deploy': npx sls make-public-function --
stage ${self:provider.stage}

provider:
  name: google
  runtime: nodejs14
  region: europe-west2
  project: robust-resource-12345

plugins:
  - serverless-google-cloudfunctions

functions:
  uploadHandler:
    handler: uploadHandler
    # The `events` block defines how to trigger the
uploadImage.handler code
    events:
      - http: upload

package:
  include:
    - index.js
  excludeDevDependencies: false
```

D.3 Deploy script for Google Function

```
#!/bin/bash
PROJECT_ID=<enter your project id>
API_ID=demo-gfunction-api
CONFIG_ID=gfunction-api-configs
PROJECT_NUMBER=<enter your project num>
REGION=europe-west2
GATEWAY_ID=demo-gfunction-gateway

# Enable required services
gcloud services enable apigateway.googleapis.com
gcloud services enable servicemanagement.googleapis.com
gcloud services enable servicecontrol.googleapis.com
```

```
#serverless deploy
#gcloud functions deploy demo-gfunction-dev-uploadHandler  --
runtime=nodejs14 --trigger-http --allow-unauthenticated
#gcloud functions deploy demo-gfunction-dev-downloadHandler  --
runtime=nodejs14 --trigger-http --allow-unauthenticated

# Create an API
gcloud api-gateway apis create $API_ID --project=$PROJECT_ID

# Describe the API to see the details
gcloud api-gateway apis describe $API_ID
#gcloud beta api-gateway apis create $API_ID --project=$PROJECT_ID

# Create an API config
gcloud api-gateway api-configs create $CONFIG_ID \
  --api=$API_ID --openapi-spec=openapi2-functions.yaml \
  --backend-auth-service-account=$PROJECT_NUMBER-
compute@developer.gserviceaccount.com

# Describe the API config to see the details
gcloud api-gateway api-configs describe $CONFIG_ID \
  --api=$API_ID

# Create a gateway with the API config
gcloud api-gateway gateways create $GATEWAY_ID \
  --api=$API_ID --api-config=$CONFIG_ID \
  --location=$REGION

# Describe the gateway to see the details
gcloud api-gateway gateways describe $GATEWAY_ID \
  --location=$REGION

# Default gateway default host
DEFAULT_HOST="$(gcloud api-gateway gateways describe $GATEWAY_ID --
location=$REGION --format='value(defaultHostname)')"
echo $DEFAULT_HOST
```

D.4 ECS Cluster Ec2 Instance Profile



```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1650765549753",
      "Action": "ssm:*",
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Sid": "Stmt1650765583458",
      "Action": "ecs:*",
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```