Aalto University
School of Science

Master's programme in Computer, Communication, and Information Sciences

Serverless Cloud Computing: A Comparative Analysis of Performance, Cost, and Developer Experiences in Container-Level Services

Md. Foyzur Rahman

Master's Thesis
2023

@2023 Md. Foyzur Rahman

Aalto University
School of Science

Author Md. Foyzur Rahman

Title Serverless Cloud Computing: A Comparative Analysis of Performance, Cost, and Developer Experiences in Container-Level Services

Degree programme Computer, Communication, and Information Sciences

Major Security and Cloud Computing

Supervisor Matti Siekkinen, PhD

Advisor Matti Siekkinen, PhD

Date 31 July 2023 **Number of pages** 111+1 **Language** English

Abstract

Serverless cloud computing is a subset of cloud computing considerably adopted to build modern web applications, while the underlying server and infrastructure management duties are abstracted from customers to the cloud vendors. In serverless computing, customers must pay for the runtime consumed by their services, but they are exempt from paying for the idle time. Prior to serverless containers, customers needed to provision, scale, and manage servers, which was a bottleneck for rapidly growing customer-facing applications where latency and scaling were a concern.

The viability of adopting a serverless platform for a web application regarding performance, cost, and developer experiences is studied in this thesis. Three serverless container-level services are employed in this study from AWS and GCP. The services include GCP Cloud Run, GKE AutoPilot, and AWS EKS with AWS Fargate. Platform as a Service (PaaS) underpins the former, and Container as a Service (CaaS) the remainder. A single-page web application was created to perform incremental and spike load tests on those services to assess the performance differences. Furthermore, the cost differences are compared and analyzed. Lastly, the final element considered while evaluating the developer experiences is the complexity of using the services during the project implementation.

Based on the results of this research, it was determined that PaaS-based solutions are a high-performing, affordable alternative for CaaS-based solutions in circumstances where high levels of traffic are periodically anticipated, but sporadic latency is never a concern. Given that this study has limitations, the author recommends additional research to strengthen it.

Keywords Cloud computing, Serverless cloud computing, Platform as a Service, Container as a Service, AWS Fargate, AWS EKS, GCP Cloud Run, GKE AutoPilot

Preface

First and foremost, I would like to express my gratitude to Matti Siekkinen, PhD for his guidance, tremendous support, and thesis supervision. I am grateful to my maternal uncle Engineer Abdur Rahman, who has been a great mentor and inspiration. I want to thank my friends and family for their encouragement and support in completing the master's degree. I dedicate this thesis to my daughter Faira Rahman who is almost three years old.

I want to thank my professors, teaching assistants, and student service coordinators, especially Anu Kuusela, for her incredible support during my tenure at Aalto University.

Helsinki, Finland 05.07.2023

Md. Foyzur Rahman

Contents

Abbreviations

API	Application Programming Interface
REST API	Representational State Transfer API
IaaS	Infrastructure as a Service
CaaS	Container as a Service
PaaS	Platform as a Service
DbaaS	Database as a Service
SaaS	Software as a Service
FaaS	Function as a Service
UTS	Unix Time Sharing
CI	Continuous Integration
CD	Continuous Delivery
AWS	Amazon Web Services
VMM	Virtual Machine Monitor
CLI	Command Line Interface
IoT	Internet of Things
IPC	Interprocess communication namespace
PID	Process namespace
OS	Operating System
I/O	Input Output
PITR	Point-in-time Recovery
NIST	National Institute of Standards and Technology
RDS	Relational Database Systems
EBS	Elastic Block Storage
ECS	Elastic Container Service
SQL	Structured Query Language
SLAs	Service Level Agreements
MVC	Model View Controller
SOA	Service Oriented Architecture
GCP	Google Cloud Platform
ACID	Atomicity Consistency Isolation Durability
DNS	Domain Name System
UML	User Mode Linux
CRUD	Create Read Update and Delete
HTML	HyperText Markup Language
MVT	Model View Template
SSD	Solid State Drive
NoSQL	Non-SQL (No Structured Query Language)
URL	Uniform Resource
Gbps	Gigabits per second
Mbps	Megabits per second
GCP	Google Cloud Platform
YAML	Yet Another Markup Language
HTTPS	Hypertext Transfer Protocol Secure
HTTP	Hypertext Transfer Protocol

List of Tables

List of Figures

1 Introduction

Cloud computing has gained widespread acceptance and adopted cloud computing paradigms, particularly Infrastructure-as-a-Service (IaaS), with the offerings of on-demand resources based on a pay-as-you-go pricing model [36]. The main reason for the rising adaptation of cloud computing in the IT industry is its pay-as-you-go pricing model, which allows customers to only pay for the resources they leverage [54]. Furthermore, they can lease as many on-demand resources as needed during a sudden traffic surge without incurring additional up-front costs. This has enabled customers to transform their running workloads from on-premise to a public cloud provider, where they offload the burden of infrastructure management, such as hardware management, to cloud provider [1]. However, customers are still responsible for configuring the elasticity, including auto-scaling to handle a sudden spike in the incoming requests . Customers are billed for the resources they allocate rather than the actual consumption. Studies showed a substantial gap between cloud users' actual usage and the resources they allotted and paid for [2]. Historically, customers needed to rent or purchase dedicated servers to run application workloads in a private cloud [1]. This was a costly solution in most circumstances and required a substantial investment because customers needed to hire experts to operate and maintain the underlying infrastructure. When more computational power is needed, they must purchase more licenses and hardware, then install and configure them before they can be used, lengthening lead times and affecting productivity. Customers can now use a public cloud service provider to provide a complex infrastructure in minutes.

Commercial cloud computing solutions delegate server and infrastructure maintenance to cloud service providers, allowing customers to focus on building the business logic rather than worrying about the underlying infrastructure. This increases the likelihood of a faster time to market and increased revenue. Service providers are constantly working on adding new features to a current service model or developing one that will allow them to take on more duties. The serverless cloud computing paradigm, which promises to relieve customers of infrastructure management responsibilities, is one of the newest service model additions [4]. Customers only pay for the processed requests, not server management or scalability. Although the phrase "serverless" hints at that there are no servers running, it actually implies that server administration has been decoupled from users [1, 79]. In addition, customers do not need to know how many servers are required to handle a given quantity of requests. Cloud service providers make on-demand scaling decisions and automatically scale from zero to infinite. Vendors only charge customers for resource consumption during runtime. Serverless computing's cost model appeals to workloads that only need to run occasionally because serverless scales to zero unless otherwise stated in the configuration and eliminates the need to pay for any idle servers [1].

Container-as-a-Service (CaaS), a contemporary cloud service paradigm based on container virtualization, is an emerging serverless cloud computing paradigm subset. Vendors offer fully managed and serverless container platforms under the CaaS model to their customers; for example, AWS Elastic Kubernetes Service (AWS EKS) and GCP Google Kubernetes Service (GCP GKE) is a managed container service

offerings. On the other hand, this thesis utilized the GCP GKE AutoPilot and AWS EKS with AWS Fargate, which are part of serverless container service offerings. One of the main discrepancies between a managed and serverless container service offering is that customers need to create, manage, and apply security patches to nodes to run the application workloads in managed container service offerings in contrast to the serverless container platform. In serverless container service offerings, the customers deploy the containerized application in a serverless cloud environment where containers are provisioned and executed in a restricted environment's specification. Previously, virtual machines (VMs) were used to emulate physical hardware resources that run on their own independent Operating System (OS) on top of a hypervisor (a virtual machine monitor, VMM) [20]. Vendors manage VMs and their underlying host machines. They are responsible for keeping the underlying hardware up-to-date and providing VMs to host on-demand containers as per necessities. Serverless CaaS enables developers to build and deploy highly elastic containerized applications utilizing their preferred programming language or frameworks, for instance, Golang, Node.js, Java, .NET, and others [8].

Platform-as-a-Service (PaaS), a serverless computing paradigm, allows developers to bring their code and deploy it leaving the underlying VMs management and scaling responsibilities to the vendors [3]. In PaaS, customers' applications run and execute inside the containers, with less control over operations and the overall infrastructure. When a container no longer receives any process requests for a brief period, it is scaled to zero, and customers are not billed for it, even while the underlying VM continues to function in the background and is hidden from customers' view [5]. GCP Cloud Run used in this study falls into PaaS, which is a serverless container service offering.

Serverless computing has proven useful and economically attractive to host workloads for web applications, REST APIs backend, back-office administration, lightweight data transformation, and many more. Studies showed the most prominent uses for serverless computing include hosting web applications and the REST APIs backend [24]. This research studied the viability of serverless container service on CaaS and PaaS platforms across two public cloud providers (AWS, GCP) by developing and deploying a containerized web application.

The PaaS and CaaS service models are discussed in section 2.4, and at the end of section 2.4.6, a summary is presented to highlight the main differences among different service models based on the shared responsibilities model between the vendors and customers.

1.1 Problem Statement

This thesis calls the fully managed Kubernetes service a serverfull Kubernetes for convenience. In a serverfull Kubernetes cluster in any public cloud, a minimum quantity of VMs must always be running to serve any incoming traffic, even if there is none for a while. Scaling to zero nodes off pick hour is possible manually or by automation. Nevertheless, this has to be done by the customers. Additionally, it could take a while to scale a VM (its size and family class may impact the scaling behavior) amid a sudden traffic surge. In the worst-case situation, some nodes might become stuck

when auto-scaling. The author has experienced such inconsistent behavior of serverfull Kubernetes in Microsoft Azure at work. Moreover, based on the author's experience, in many cases, a virtual machine may take five to fifteen minutes (sometimes over twenty minutes) while scaling out a node. In such a case, as the necessary resources, including CPU and memory are unavailable to all nodes simultaneously, the kube-scheduler can no longer schedule additional Pods to worker nodes. This eventually results in a higher response time for the page (sometimes website may freeze briefly), which has an adverse effect on the user experience. This behavior can be a performance bottleneck for the customer-facing application if the higher latency is not affordable. Furthermore, customers must pay the machine's per-hour bill even though the entire serverfull Kubernetes cluster is idle. Upgrading the Kubernetes cluster, applying the security patching to the nodes, rebooting the nodes periodically, updating the node's configuration, tightening the security configuration, optimizing the resource utilization, finding the correct number for scaling the on-demand nodes and a bunches of other operational duties falls into the customers' table in the serverfull Kubernetes. As a consequence, serverfull Kubernetes gradually accrues more operational overhead, and often customers are left with no choice but to run the bare minimum number of cluster nodes necessary to handle incoming requests. This forces customers to pay the additional fees without fully utilising the resources.

1.2 Motivation

Serverless cloud computing is evolving and being used widely, a considerable number of research papers were published on its use-cases, benefits, and drawbacks in data analysis, web applications backend, Internet of Things (IoT) applications [32, 33, 34]. According to our research at Aalto University, at least one thesis evaluated the language-specific runtime performance in FaaS on AWS and GCP. Furthermore, the study analyzed the performance, cost differences, and application development and deployment experiences between the FaaS and PaaS, where GCP App Engine was used for the PaaS service [4]. Nonetheless, studies on the performance, cost analysis, and convenience of the use of AWS EKS with AWS Fargate and GCP Cloud Run for hosting web applications are still insufficient [15, 25, 26, 27, 29, 30]. This is because GCP Cloud Run and AWS EKS with AWS Fargate compatibility, released in the fourth quarter of 2019, are relatively new additions to the serverless product fleet. On the other hand, GCP GKE Autopilot was released in the first quarter of 2021 [76]. Therefore, the author is motivated to study the container-level serverless cloud computing services to compare the differences in performance, cost, development, and deployment experience.

Studies show that AWS is the global market leader, and its market share was about 47.8% in the year 2018 [25, 61]. AWS was the dominant cloud service provider in the fourth quarter of 2019, according to the research [28], with a market share of 32% and revenue of $9.8 billion. Microsoft Azure came in second with 17% and $5.3 billion in revenue, while Google Cloud placed third with only a 6% market share and $1.8 billion in revenue.

The author encountered unpleasant user experiences with serverless Kubernetes in

Azure. This is why serverless Kubernetes in Microsoft Azure cloud is not considered to utilize in this study. The reason we call it serverless Kubernetes is that public cloud vendors manage and maintain the control and data plane of the Kubernetes cluster. Customers can not see the data plane nodes because they're concealed. IBM Cloud was scoped out of this research since IBM Cloud is not as popular as Google Cloud or AWS. Due to the mentioned reason, the author choose to conduct the research study in AWS and GCP as they are the two big players based on market share and popularity. .

1.3 Scope and Goal

The rationale mentioned in the previous sections drove us to develop and deploy a web application on serverless CaaS and PaaS platforms in AWS and GCP to evaluate their differences in performance, cost, and the complexity of using the services.

A lightweight single-page web application was developed to wrap up the research work. However, due to time constraints, developing a full-blown complete web application was outside the scope. Three services from two separate public cloud service providers were chosen to conduct the research, which included GCP Cloud Run, GCP GKE AutoPilot, and AWS EKS with AWS Fargate. The latter two fall into the category of CaaS service models. On the other hand, the former is part of PaaS service model offerings. All three services in this study are part of serverless cloud computing, as server management is abstracted from the customers' duties. Customers only need to take care of the application workload, while shifting the rest of duties to the vendors.

The research's primary goal is to examine if adopting a serverless computing platform would be viable in terms of performance and cost variations, focusing on container-level services, including CaaS and PaaS service models. In addition, the author includes a summary of the research's findings on how simple it is for a developer to build and deploy a web application to each service. The developer experiences are presented based on the author's experience during the research. Being said that, the purpose of this study is to provide research answers to the subsequent questions:

RQ1 *Is a serverless CaaS and PaaS solution on the public cloud platforms for a containerized web application a viable approach regarding cost and performance?*

RQ2 *How does developing and deploying a web application on a serverless CaaS and PaaS solution influence the development experience?*

1.4 Main Contribution

The main contribution is mentioned below:

- Evaluating the core features of the existing offerings among GCP Cloud Run, GCP GKE AutoPilot, and AWS EKS with AWS Fargate.
- Partial implementation of setting up the cloud infrastructure via Infrastructure-as-Code (IaC) to host the web application on the serverless computing platform.

- Partial implementation of automated DevOps pipeline to deploy services automatically. Due to time constraints, the workload and scope were too vast, so IaC and the automated DevOps pipeline did not cover automating all services.
- Wrote the load tests scripts
- A single page web application is developed
- In-depth analysis of performance and cost including the summary of the author's development experiences

1.5 Thesis Structure

These five chapters have three main sections: background study of the research, project setup and implementation, and performance and cost analysis. Chapters 2 and 3 are part of the research background section, whereas chapters 4 and 5 fall into the implementation and evaluation sections, respectively. Chapter 2 covers the relevant research concepts and technologies, for instance, virtualization, containers, Docker, Kubernetes, Continuous Integration and Continuous Deployment (CI/CD), various public cloud service models, and monolith and microservice architecture, including their use cases, benefits, and drawbacks. However, chapter 3 discusses serverless cloud computing, including its use cases, advantages, and disadvantages. The implementation is presented in chapter 4. Chapter 5 evaluates the in-depth performance, cost analysis, and ease of use of those services from a developer's point of view. Finally, chapter 6 concludes this thesis with some remarks and reflections and presents future work.

2 Background

This chapter introduces two key components of cloud computing: virtualization and containerization technology, cloud computing service models and technologies. In addition, two most commonly used software design patterns are introduced. The virtualization and containerization technologies that have fostered today's cloud computing are discussed in section 2.1. The architecture and components of Docker are briefly discussed in section 2.2. Kubernetes is presented in section 2.3. Several aspects of cloud computing are discussed in section 2.4. Continuous Integration and Continuous Delivery (CI/CD) are discussed in section 2.5. Finally, section 2.6 reviewed and analyzed the two widely used software architecture patterns.

2.1 Virtualization Technologies

Virtualization is the foundation of cloud computing. It allows the creation of a simulated computing environment by virtualizing the hardware resources of an underlying host machine. This allows a single physical server or host machine to partition into several VMs, also known as guest machines, on which each can interact independently while sharing its host machine's underlying resources. Moreover, each VM is platform-independent; each VM acts as an independent computer, but in reality, it is just a tiny portion of the host machine. When an application runs on top of a completely isolated VM, it is not connected to its host machine, which sits below it. Since virtualization creates multiple virtual resources from its host machine, it efficiently utilizes hardware resources, reducing the overall cost associated with infrastructure, maintenance, and energy consumption. Additionally, it enables better scalability, portability of workloads, simpler-to-manage IT resources, and increased performance. It helps cloud providers to boost the agility to create new cloud resources for their customers, which mainly drives the cloud providers' economics [6]. This section presents two virtualization technologies and highlights their main discrepancies.

2.1.1 Hypervisor based virtualization

A hypervisor is a small piece of software that can create, coordinate and run VMs [20]. It is an interface between the VM and its underlying commodity hardware. It ensures that each VM has access to its host machine's resources. Furthermore, impinging on each guest machine's resources, including memory, storage, and CPU, ensures that none of the VMs interfere. Virtualization also has some security challenges; for instance, if a hypervisor is compromised, an attacker can potentially own and control all the VMs since a hypervisor allows each of the VMs to communicate with each other. On the other hand, it offers some security benefits; for example, when a VM is infected with malicious programs, it can be quickly recovered to a particular point in time, usually by the previously taken snapshots. If no snapshots are available for Point-in-Time Recovery (PITR), the VMs can be deleted and recreated. There are two distinct types of hypervisors [20, 22, 149]:

- **Type 1 or bare metal or native hypervisor**: With bare-metal hypervisor technology, the host machine's OS and the VMs use the same hypervisor. It runs directly on top of the hardware, controlling the resources of the underlying host machine.

  Figure 1 shows that Type 1 has no OS in between. Type 1 hypervisor is installed directly on the underlying physical hardware. Connecting to an external monitor during the bootup displays a Command Line Interface (CLI) prompt-like screen where hardware and network-related details can be seen. Typically, VMs are created and managed by connecting to the hypervisor from another machine. The management console must be set up from another device to do that. Some commercial and open source examples are VMWare vSphere with ESX/ESXi, Microsoft Hyper-V, Citrix Hypervisor, Oracle VM, and KVM (Kernel-Based Virtual Machine).

  A bare-metal hypervisor controls and manages the guest machines. It has less latency but better performance since there is no host OS overhead between accessing the underlying physical hardware resources in contrast to Type 2. In the 1960s, IBM developed native hypervisor technology that included the CP/CMS OS, a mainframe-based product, and test software called SIMMON.

- **Type 2 or hosted hypervisor**: Type 2 runs on a orthodox OS. With this technology, a VM runs as a process on the host by creating an abstracted layer from the host machine. The guest OS is required to access the underlying hardware resources via the host machine's OS. This adds to the extra latency, thus leading to slower performance in the end.

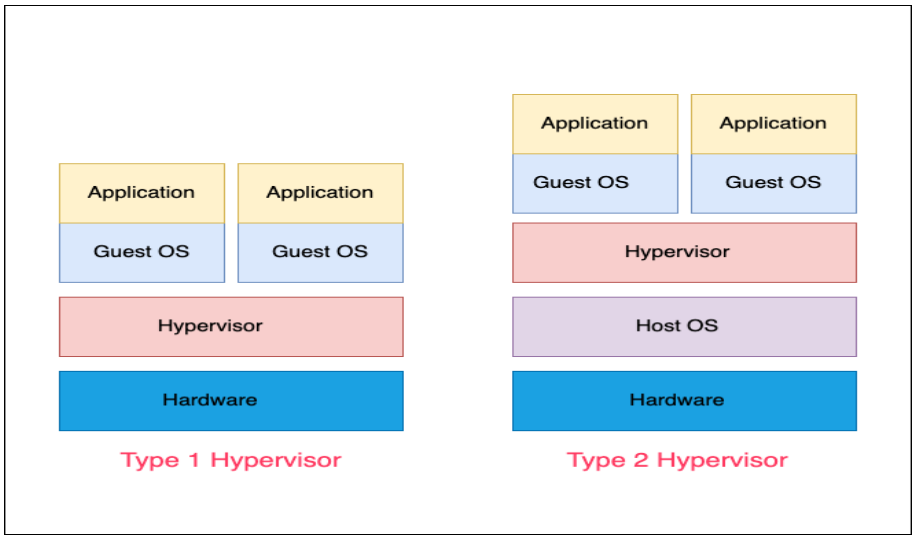

Figure 1: Type 1 and Type 2 Hypervisor

Figure 1 shows that the hypervisor runs on the host machine's OS. Type 2 examples are Oracle VM virtual box, VMWare Fusion, Windows PC, and Parallels Desktop. For example, users only need to install the Oracle VM virtual

box on their machines to create and manage VMs with different OSs. They can, for instance, import and export appliances, clone and take snapshots of a VM.

Virtualization in cloud computing provides excellent portability and is commonly used to share VM snapshots across user accounts, between host machines, and as custom base images in auto-scaling. This allows cloud providers to run numerous VMs on the same physical host machine, allowing them to make efficient and cost-effective use of the hardware resources. Virtualization technology abstracts hardware from end-users, allowing them to focus on providing the computing resources they require, such as storage, networking, servers, and database instances. End users can adjust the machine configuration with just a few clicks, which makes the user experience extremely satisfactory. This is why it has become a game-changing technology in today's cloud computing. Resource provisioning would be highly laborious without the virtualization technology due to the underlying hardware changes.

Category	Type 1	Type 2
Virtualization	Hardware virtualization	OS Virtualization
Security	More secure	Less Secure
Performance	No OS overhead and higher performance	Comparatively slower
System Dependency	Directly access the underlying hardware, including VMs hosts	Directly not allowed to access host hardware and its resources
Operation	Guest OS and applications run on the hypervisor	Runs on the host OS as an application
Scalability	Better Scalability	Not so much due to the reliance on the host OS
Management	A separate machine is required to administer VMs and control the underlying hardware	From one machine VMs are administered
Speed	Faster	Slower

Table 1: A comparison between the Type 1 and Type 2 Hypervisor

The advantages and disadvantages of Type 1 and Type 2 hypervisors are compared in Table 1. Large enterprises choose Type 1 hypervisors in cloud computing because they are more secure, require no middleware, and are independent of the host OS. Users often need to install additional toolkits for Type 2 hypervisors in order to copy and paste files from the host to the guest machine and vice versa. Type 2 must go through the host machine's hardware and resources for all operations, imposing some latencies without utilizing full use of resources [149]. In developer environments, when many OS variations are required, Type 2 is ideal.

2.1.2 Container based virtualization

In cloud computing and microservices, containers have become the de facto industry standard. The Container technology era began with Docker back in 2013 [21]. We discuss docker in section 2.2. A container image includes everything required to run an application, for example, packaging the application source codes and associated settings, along with its dependencies, and clear instructions in a Dockerfile on what processes the container must run when it is orchestrated. A containerized application must always run the same regardless of its underlying hardware. Each container runs on its namespace and is separated from other container namespaces within the same host machine, while all containers share the same kernel. The kernel manages namespaces, and therefore it knows which namespaces are assigned to which process while exposing one process for a container to the host machine. Furthermore, it ensures the process can access resources only on its namespace during the Application Programming Interface (API) call.

Containers do not use a hypervisor, so containers are faster. Resource provisioning is significantly quicker, unlike hypervisor-based virtualization. However, container

images are slim and small; for example, an alpine base image is 5MB in size (compressed size is about 2.28MB) [91]. Technically, containers would require fewer VMs and OS to handle more applications than Hypervisor-based virtualization. A fully virtualized machine usually takes over a minute to boot because it includes an entire copy of the OS, libraries, binaries, and other necessary dependencies to run the application. Containers start in just milliseconds to seconds since it does not start the entire copy of the OS. Also, unlike containers, virtualized hardware resources need to be preallocated, including memory, storage, and CPU, while creating a VM. This allows running many containers on a single OS while isolating the container file systems and processes from each other. This would be practically impossible within the given equal resource limits when we run separate copies of the full OS on their VMs. Unlike VMs, it is significantly faster to test and deploy a particular application on different OS using containers. Containers, while fast and lightweight, have one key disadvantage over VMs: they are restricted to the host OS.

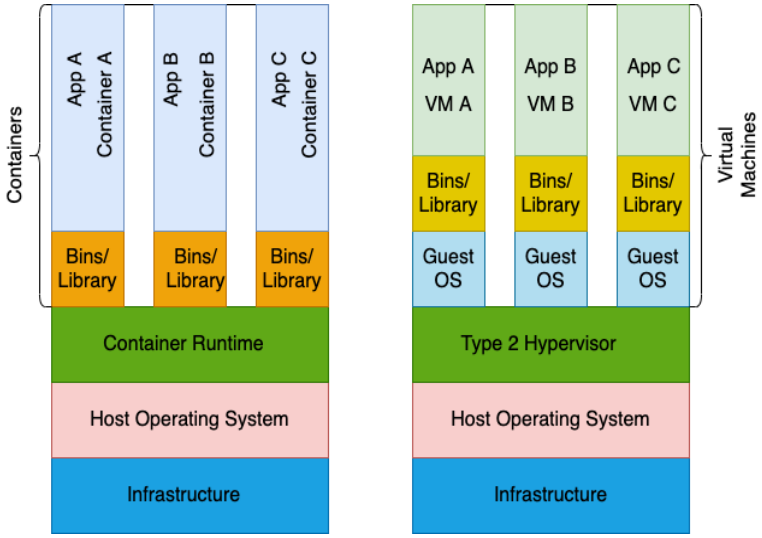

Figure 2: Container and Hypervisor based architecture.

Figure 2 shows the architectural differences between containers and VMs. Containers run on the host OS, while the VM runs on its guest OS.

Container process isolation and resource allocation happen at the OS level. The Linux kernel has native support for process isolation and limiting resource usage using namespaces and groups (control groups), respectively. Namespaces are responsible for partitioning the kernel resources, allowing each service to access just the resources associated with its namespace. CGroups manage and control the granular resource usage (CPU, memory, Network I/O, or access to the file system) to a particular process or a set of processes. There are several sorts of namespaces in the Linux kernel, each with its own set of features [14, 84].

- **User namespace:** Each user has a unique UID, and a user can be assigned to a group or multiple groups. Each group has its group ID. A process can run in its

user space with root privileges, while others can run processes without a root in its user namespace.

- **Process namespace (PID):** Assigns PIDs for a set of processes in each user namespace. Each PID is independent of other PIDs that reside in other namespaces. The first process is assigned to PID 1, becoming the parent process in a namespace. In addition to this, all its child processes get the subsequent PIDs.
- **Network namespace:** Isolates, virtualizes, and manages each component of the network stack- firewall rules, socket listing, private IP address, routing table, and network interfaces.
- **Mount namespace:** Responsible for controlling and isolating mount points in a user namespace for running containers. Processes running in a particular mount namespace can mount and unmount the file systems. This ensures each container is running.
- **Interprocess communication namespace (IPC):** Isolates specific IPC resources such as Portable Operating System Interface (POSIX) message queues and System V IPC resources. When we create objects in an IPC namespace, they can not be seen by other processes outside that namespace, allowing only visibility to all member processes.
- **Unix time-sharing namespace (UTS):** Provides isolation of host and domain names for containers in a single system.
- **Cgroups:** Allows processes into organized hierarchical groups to limit, prioritize and monitor the system's resource usage, including Network bandwidth, Disk, CPU, Memory, and devices.
- **Time Namespace:** Allows containers to see different system clock times within the same host in a way akin to UTS.

Container technology has outclassed the performance compared to hypervisor-based virtualization technology. According to studies, containers have a lower Quality of Service (QoS) in terms of storage transaction speed [58]. Knowing that containers share the same kernel as the host machine, posing many security risks [43]. Container processes, for example, can access the host machine's file systems via mount points when running as a root user or in "privileged" mode. Additionally, container processes have essentially identical privilege rights to the running processes on their host machines. Container runtimes require root user privileges to allow functionality, including port binding, networking, and mounting file systems. The direct access to the host machine's kernel substantially increases the attack surface. Attackers were able to execute arbitrary code, obtain privileges, and alter memory via various kernel flaws, according to the report [83]. As a result, an attacker who can exploit kernel issues inside containers can also exploit them on the host machine. Conversely, on a VM, the process is a little more difficult and time-consuming because attackers must exploit the

VM Kernel and the VM Hypervisor. Studies show that [43] running a container inside a VM can minimize the attack surface. Before breaking into the host machine systems, an attacker must exploit the VM kernel, the hypervisor, and the host machine's kernel [50]. When attackers undertake Denial-of-Service (DoS) and Privilege escalation attacks on a host machine where multiple containers are running inside VM, attackers can gain full control of a subset of containers while the remaining containers remain still under the full control of legitimate users [50]. Container runtime engines and containers can be run in non-root users to reduce conceivable security flaws[89].

Although, in serverless cloud computing, container resources are pre-allocated before containers are even orchestrated. However, if container resource limits are not pre-allocated before containers are provisioned, running containers will consume the resources available on the host when more computing resources are necessary. Since the host kernel is shared with containers; therefore this could pose issues in multi-tenancy architecture because the containers and resource isolation are less efficacious than VMs.

2.2 Docker

Docker is an open-source software project makes it easier to swiftly develop, test, package, ship, and run applications across multiple platforms quickly [11]. Developers can use Docker to package the entire application, including binaries, runtime, and all dependencies, into a single Docker image. Besides that, DevOps teams can control infrastructure correspondingly; they manage the applications with docker. The following are some of the most common Docker use cases:

- Run applications from developers' local physical or VMs to hybrid, public, private, or community clouds.
- Fast, consistent delivery of applications using CI/CD workflows.
- Scalable and responsive deployment.
- Using the same hardware to run huge workloads.

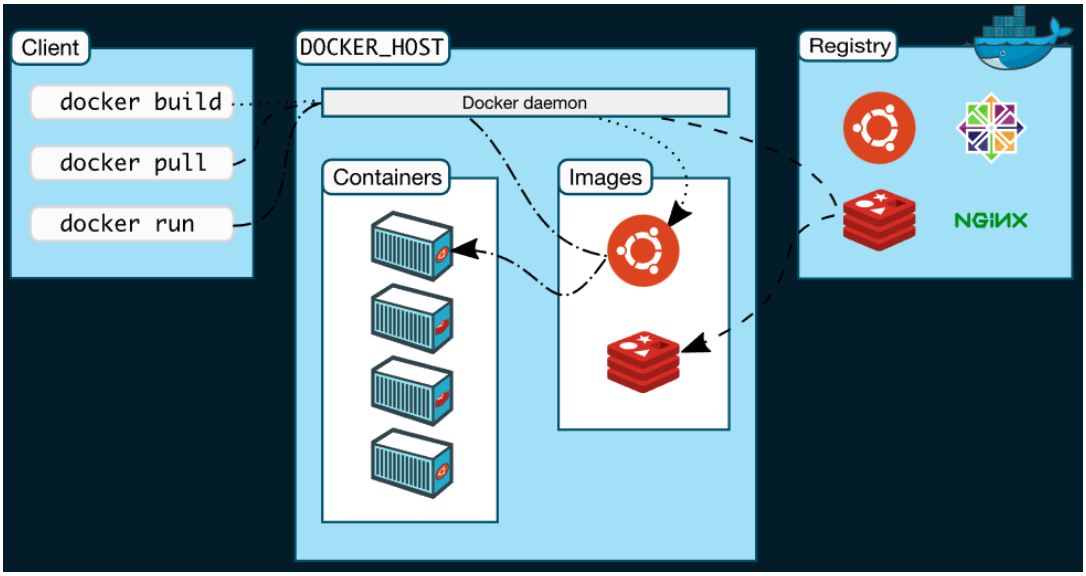

Figure 3: Docker Architecture [101]

Figure 3 illustrates the architecture of docker based on the client-server architecture model. Docker comprises a docker engine in which docker or docker-compose, a CLI client, works as a client and dockerd, a long-running daemon process, acts as a server [101]. The Docker client talks with the Docker daemon over a UNIX socket or network interface via the Representational State Transfer API (REST API) [101]. Users interact with the docker engine executing docker commands. Briefly, we present various docker components below [7, 10, 11, 49, 101, 102].

- **Docker Daemon:** Docker daemon handles the API calls and manages docker objects, including images, containers, volumes, and networks. Users can not interact directly with the Docker daemon.

- **Docker Client:** Users interact with Docker daemon invoking Docker commands to create and remove images, containers, networks, and volumes. For example, when a user issues a Docker command, for instance, docker pull, the Docker client immediately sends the command to the docker daemon, which executes it. A remote connection between a Docker client and a daemon is also conceivable. Typically, they reside on the same machine.

- **Docker Registry:** A Docker registry stores and manages docker images. By default, images are fetched from the Docker Hub, a public registry, unless a separate container registry is specified. Commercial examples of container registries include DockerHub, Quay, Amazon Elastic Container Registry, Alibaba Container Registry, Google Container Registry, and GitLab Container Registry.

- **Docker Images:** A Docker image is a read-only template that can be built, providing step-by-step instructions in a Dockerfile. However, a Dockerfile is written in text format, which typically resides at the project's root and contains instructions about commands that a user could run in the CLI using a Docker client. Docker executes each instruction sequentially in a Dockerfile. Every instruction given in a Dockerfile creates a layer in the Docker image.

- **Docker Container:** Users can administer running or stopped containers. Users can also attach a persistence data volume to a container while running a container on a separate private network if necessary or even create new images based on the existing container state.

- **Docker Compose:** A docker-compose allows users to run multi-container applications. For example, docker object instructions are written in *compose.yml* file to orchestrate multiple containers such as frontend, backend, and database containers to run an application.

2.3 Kubernetes

Containers are the core technology of serverless cloud computing. Google has been running applications on containers since it introduced container management tools more than a decade ago. Google first developed Borg for their internal use to run containerized applications, including Google Search, Gmail, and Google Compute Engine. It was built for managing batch jobs and long-running services. Borg was a robust tool that offered service discovery, auto-scaling, load balancing, quota management, and VM lifecycle management. Omega was developed to improve the Borg's ecosystem, which was a descendant of the Borg, but they had to re-architecture from the ground up. Google used Borg and Omega for their internal services. After Borg and Omega, Kubernetes was the third container orchestration tool developed by Google. A container orchestration tool help to orchestrate a batch of containers and streamline their life cycles. Most container orchestration tools provide more or less similar features from a user point of view. Some popular container orchestration tools are Kubernetes, Docker Swarm, OpenShift, Marathon, and Apache Mesos [44]. In this

thesis, the instance or container instance refers to the replica in the Kubernetes context. The author might call it replica, instance, or container instance in the Kubernetes context throughout the thesis for convenience.

Google created Kubernetes, a technology for managing and orchestrating containers [12]. It enables automated deployment, auto-scaling and self-healing, automated rolling out, rollbacks a deployment to a previous release with zero downtime. After introducing Kubernetes, its interest grew exponentially among developers. More importantly, this growing business interest led Google to build its global public cloud infrastructure. Kubernetes was built based on Borg to ease deploying and managing distributed complex systems while providing a great developer experience to write and run containerized applications within a cluster [48].

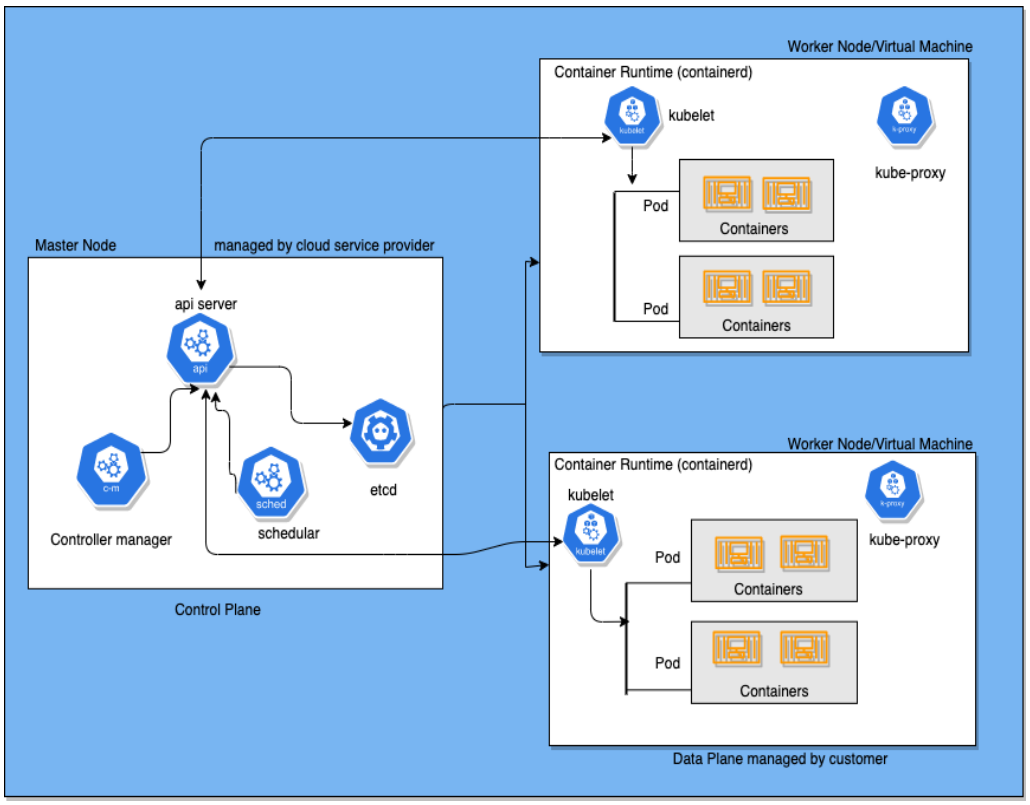

Figure 4: A managed Kubernetes cluster architecture

A Kubernetes cluster comprises one or more VMs, often three. Within a cluster, these VMs are known as nodes. Each node within the cluster shares computing resources, for example, storage. A single node can serve as the master and worker nodes in a cluster. As a result, the single-node cluster handles all workloads. If a node dies, the cluster and any services or application workload running on that cluster also die. One node serves as the master node and the others as worker nodes when a cluster comprises more than one node. The master node periodically checks the worker nodes' health. If it discovers any unhealthy nodes in the cluster, no new application workloads are scheduled on that node. It also determines which cluster worker nodes should be used for which application workloads. The cluster's nodes can communicate with one

another. Workloads for applications, such as containerized apps deployed as Pods, are hosted on worker nodes [9].

Kubectl is a command-line interface (CLI) client, and operators use it to communicate with the Kubernetes cluster, mainly through the master node. Kube-apiserver sits on the master node and receives commands sent to the cluster using kubectl. Then kube-apiserver passes the requests to the kube-controller-manager, which is in charge of handling worker nodes operations. Kubelet, which resides on the worker nodes, eventually receives commands from the master node.

Figure 4 shows a severfull Kubernetes cluster diagram based on the client-server architecture. The Kubernetes cluster diagram is divided into the control plane (also known as the master node) and the data plane (known as the worker node or nodes). Each of the worker nodes has its own Linux/Windows environment. Containers live within Pods, and Pods live within the nodes. The master node runs all the essential processes , including Kube API Server, Kube Scheduler, and Kube Controller Manager, which are required to manage and run a cluster. Additionally, the master node uses a key-value data store called etcd which is hosted by the master node. In the public cloud, a serverfull Kubernetes cluster is divided into two parts based on the operational duties:

- Control planes are managed by cloud providers, which orchestrate the computing resources for an application along with all other Kubernetes services. Vendors' responsibility is to keep the master node up and running always.
- Data plane nodes run the application workloads, and customers' duty to configure, apply security patches, and keep the nodes up to date. At least one cluster node, typically a VM that runs the container runtime and the Kubernetes node components, is needed to run the application workloads and supporting services.

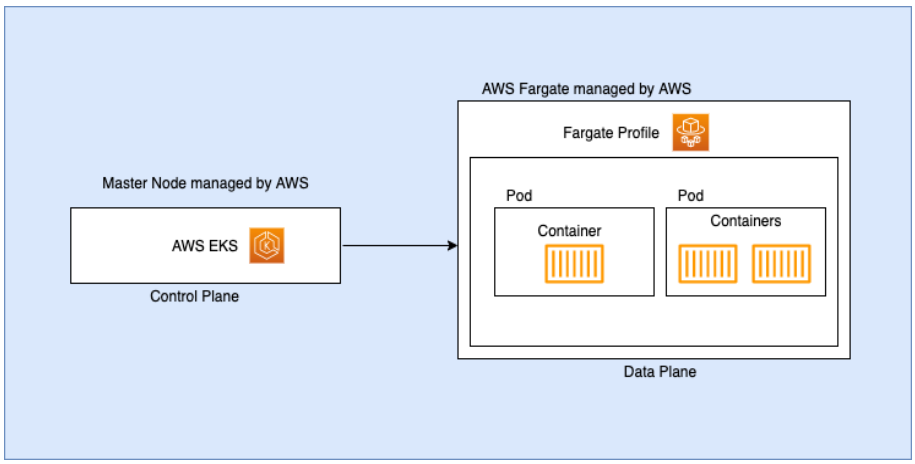

Figure 5: Managed AWS EKS running workloads on serverless AWS Fargate

Figure 5 shows the master node running on AWS EKS service and serverless Kubernetes Pods running on AWS Fargate, with AWS managing, provisioning, scaling, and maintaining the underlying master and Fargate nodes. On the right-hand side, it

can be seen that customers only need to create a Fargate profile and configure Pods on which Fargate profile their application workloads must run. Figures 4 and 5 differ primarily in terms of operational responsibilities and pricing approach, where Figure 5 delegates node administration and configuration to the cloud service provider. We go through AWS EKS with AWS Fargate in section 3.5 in depth. The following are some of the core components of Kubernetes [9, 12, 14, 103, 104].

- **Pod** consists of one or more containers. Containers reside within the Pods. Containers are co-located and co-scheduled within a Pod and tightly coupled. When a Pod dies, all the containers within a pod die as well. Before the application containers startup, each Pod runs the init containers. Each container in a pod has access to the same resources. Containers within the same pod communicate using localhost, but containers in different Pods use the pod IP address. Kubernetes considers Pods to be single-functioning units scaled according to the configuration. Pods live briefly and can be evicted anytime due to resource unavailability and the configuration specified on the *deployment.yaml* file. By default, the kube-scheduler does not schedule any application deployment Pods to a master node due to security concerns.

- **Kubectl** is a CLI tool that allows users to interact with Kubernetes' API server via kubectl commands.

- **DaemonSet** is responsible for running at least one single Pod in some or all of the nodes. Pods are added to nodes when some workloads run on the cluster. Depending on the settings, when a node is removed from a cluster, running Pods are either terminated or transferred to another node within the cluster. If a DaemonSet is removed, the Pods it spawned are cleaned up.

- **Worker Nodes** are commonly known as nodes. A Kubernetes cluster consists of at least one node that serves as both the master and worker node. However, a single-node Kubernetes cluster is often used for local application development using Minikube or similar tools. Clusters must have more than one node in any production-grade cluster environment. Worker nodes are used to host Pods. Node provides all of the services required to run Pods.

- **ReplicaSet** guarantees at any given time the specified number of desired copies of an identical pod running

- **ReplicationController** assures that a certain number of homogeneous Pods are always operational at a given time. It also guarantees that the number of specified Pods in the specification is met by terminating extra Pods or launching more Pods.

- **Deployment** ensures that only a certain amount of Pods are down while they are being updated, thanks to deployment. It ensures that at least 75% of the appropriate Pods are up by default (25 percent max unavailable). Pods, Containers, and ReplicaSets are all specified in Deployments. The deployment

specification specifies the intended state, and the deployment controller gradually advances the current state in the direction of the intended state. One of the most significant Deployment use cases is to roll back to an earlier version of the application deployment.

- **Service** is an abstraction that exposes the application operating on Pods or a set of Pods. Each pod gets assigned to an IP address while a set of Pods are assigned to a single Domain Name System (DNS) name, allowing them to load balance across them.

- **Horizontal Pod Autoscaler (HPA)** adjusts the number of deployment Pods, replica sets, or replication controllers. This enables an application to expand to fulfill rising demand and scale when resources aren't required, freeing up nodes for other uses. The HPA adjusts an application's resource utilization based on the predefined configuration of the minimum and the maximum number of replicas and the metric value of a target CPU or memory percentage utilization specified in the Kubernetes manifest file. The HPA does not apply to objects that cannot be scaled, such as DaemonSets.

- **Vertical Pod Autoscaler (VPA)** automatically scales the CPU and memory reservations for existing Kubernetes Pods. This increases resource utilization while freeing CPU and memory for other Pods.

- **Cluster Autoscaler (CA)** helps to adjust (scale in and scale out based on demands) the number of nodes within a cluster.

- **Pod Disruption Budget (PDB)** helps prevent Pods from being removed and the application from becoming completely non-functional.

- **Add-ons** enhance Kubernetes functionality. It implements cluster functionalities using Kubernetes resources such as DaemonSet, and Deployment. The duty of an add-ons manager is to create and manage add-ons. DNS (serves DNS records inside cluster), Web UI (Web-based User Interface for monitoring and troubleshooting any running apps in the cluster), Cluster-level logging (responsible for storing container logs), and so on are examples of Kubernetes add-ons.

Kubernetes comprises several objects that collectively compose the control plane [9]. Some of the important components of the control plane are briefly discussed below [9, 12, 14, 103, 104].

- **Etcd** is the primary datastore Kubernetes uses to avoid race conditions and networking by storing distributed cluster data such as metadata, configuration, and state data. In a nutshell, it stores and replicates the state within a Kubernetes cluster. It is a distributed key-value data store.

- **Kube-apiserver** exposes the Kubernetes API. Its main responsibility is to scale cluster instances horizontally.

- **Kube Scheduler** assigns newly created Pods to a node on which they will operate. Before a pod is assigned to a node, the scheduler considers a variety of parameters, including affinity and anti-affinity setups, deadlines, resource requirements, and inter-workload interference, among others.
- **Kube Controller Manager** handles controller operations. Although they are contained in a single binary, the controller processes run as a single process. Contrarily, every controller process is a unique process. A few controllers are as follows:
  - **Node Controller**: The main duty is to observe and respond when a node goes down.
  - **Endpoints Controller**: Creates the endpoints objects, including hooking Pods and services.
  - **Service Account and Token Controller**: Creates the default accounts and API access token for new Kubernetes cluster namespaces.
  - **Job Controller**: Notices job objects and creates Pods to complete tasks.
- **Cloud Controller Manager** was first released as an alpha release in Kubernetes version 1.6. Cloud providers can use this component to run cloud-specific code alongside the Kubernetes controller. This also allows a cluster to interface with the cloud provider's API while separating components that deal with the cloud platform from those that only interact with the cluster.

Node components of Kubernetes run on every node, provide the Kubernetes runtime environment, and maintain all running Pods. Some of the core node components are outlined below [9, 12, 14, 103].

- **Kubelet** is a cluster agent runs on each node in the cluster to ensure containers run within Pods.
- **Container Runtime** is software responsible for running containers. Container runtime includes containerd, docker, CRI-O, and so on.
- **Kube Proxy** runs on every single node within the Kuberntes cluster and responsible for creating and managing networking rules (IP tables) on the nodes. This allows Pods to communicate over the network inside and outside the cluster from network sessions.

Responsibility	Serverfull Kubernetes Cluster	Serverless Kubernetes
Control Plane	Vendor managed	Vendor managed
Data Plane configuration and maintenance includes OS upgrade, security configuration	Customer managed	Vendor managed
Node auto-scaling configuration	Customer managed (requires calculating the capacity to house the workload)	Vendor managed
Pod autoscaling configuration	Customer managed	Customer managed
Pricing model	Compute resources are billed per second with a one-minute minimum usage	Customers never pay for the node's computing, but the Pods runtime consumption of resources (virtual CPU, memory) billed by seconds
Monitoring	Customers monitor their nodes resource utilization and adjust the configuration accordingly to accommodate workload	Vendor managed (auto health monitoring, autohealing

Table 2: Notable comparison between managed Kubernetes and serverless Kubernetes

Based on the shared responsibility model, serverfull and serverless Kubernetes can be distinct based on the responsibilities in Table 2. In contrast to serverfull Kubernetes, Table 2 demonstrates that customers still have responsibilities for Pod Configuration, security, and data management in serverless Kubernetes. After a cluster is created, there is no option to halt it; customers can either delete it or create a new one. On the other hand, in serverfull Kubernetes, customers can shut down the clusters any time they want. Both serverless and serverfull Kubernetes serve as CaaS, which is discussed in section 2.4.2.

2.4 Cloud Computing

Cloud computing is a pay-as-you-go, on-demand delivery of computing resources accessed over the internet [19]. Customers can consume resources, including data storage, servers, networking capabilities, software development tools, and services, but they do not own, buy or maintain any physical data centers or servers. There is not yet a widely accepted definition of cloud computing [53]. As mentioned in numerous publications, many have presented their own [17, 18]. The National Institute of

Standards and Technology (NIST), which is based in the United States of America and develops standards, recommendations, and guidelines, provided the definition of cloud computing that has been most frequently cited by scholars [53]. The cloud computing is a "model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction," according to NIST [66].

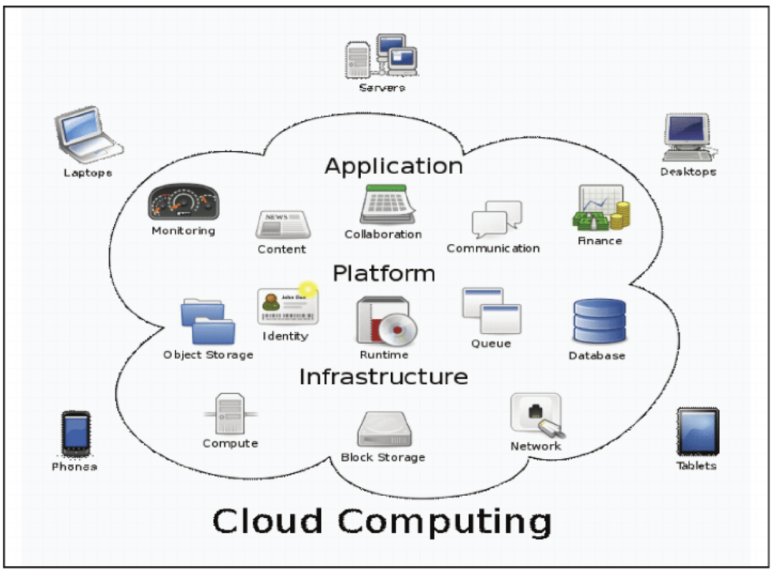

Figure 6: A conceptual architecture of cloud computing [47]

Figure 6 shows that cloud computing comprises a wide range of services, including its underlying physical hardware and networking capabilities that build the infrastructure. For example, a storage service provides block or file-based services, identity, and access management services; a data service provides column or record-based services; and a compute service provides all the computational services [47].

Cloud computing gives consumers access to a wide range of technology, allowing them to construct nearly anything on top of public or hybrid clouds. They enable faster provisioning of large-scale on-demand resources. New versions of an application can be deployed in a matter of minutes. Users are granted different levels of control depending on the service models. For example, Users do not have access to the OS when using PaaS, but they do with Infrastructure as a Service (IaaS).

Figure 7 shows different cloud computing service models. Light Grey, Green, and yellow colored rectangle boxes represent vendor- and customer-managed responsibility, respectively. Cloud service providers are responsible for securing physical data centers across different regions. They are also responsible for keeping the underlying hardware resources secure, up-to-date, and healthy without causing any service interruptions and for the availability of the computing resources at all times, including during peak demand, so that customers can conduct business without service downtime. Cloud service providers and customers work together in a shared responsibility model to ensure security and adherence to cloud computing compliance standards and

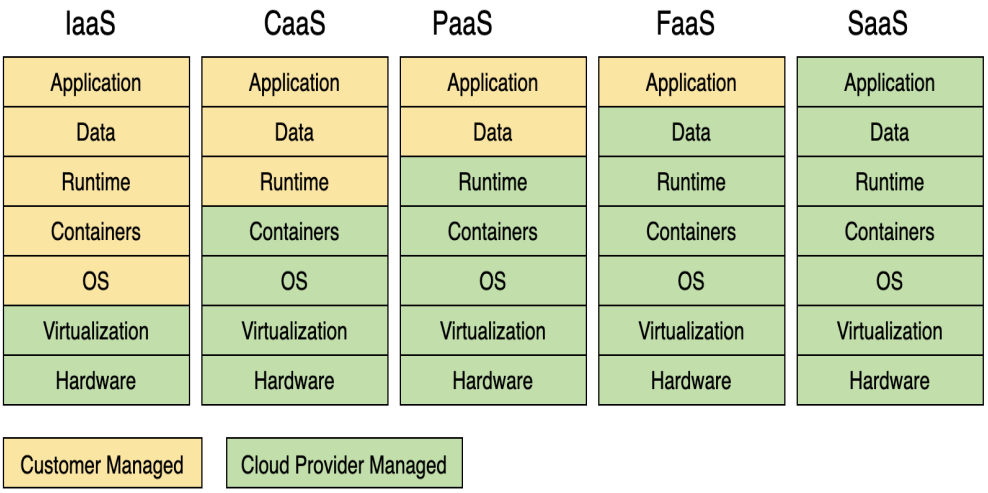

Figure 7: Different types of cloud computing service model

regulations. Typically, cloud service providers' responsibility is the security of the cloud, while customers' responsibility is the security of the cloud [97]. The whole cloud infrastructure worldwide and all the services offered and run on top of it are protected by cloud service providers, including hardware, facilities, software, and computing resources. On the other hand, customers' responsibilities differ depending on whatever services they utilize [97].

Public cloud providers frequently struggle to keep all their services up and operating worldwide within their vast infrastructure. The authors [47] presented the total hours of service outages for various cloud providers that occurred from 2009 to 2011, which directly interrupted a large number of cloud-hosted services around the world. Amazon Elastic Block Storage (AWS EBS) services outages happened in 2011, 2012, and 2013 and affected a wide range of enterprises [47, 73, 74]. Outages of Amazon EBS, ECS, and RDS services occurred in 2021 and affected Gitlab, Reddit, Xero, and Imgur, among other unknown business sites [70, 71, 98]. However, Amazon's competitors, Google and Azure Cloud, have also failed to maintain 100% availability for all of their services [71, 72]. The report showed that various cloud providers failed to maintain 100% uptime for their service offerings in 2020 [100]. Furthermore, the Hong Kong data center of Alibaba Cloud was nearly 12 hours down in 2015 [78]. This is why it is recommended to design and implement fault-tolerant systems for production sites to maintain service availability in the event of outages and, if possible, automatically failover to a different zone or region, although this is costly. The following sections briefly present different service deployment models in cloud computing.

2.4.1 Infrastructure as a Service

Infrastructure as a Service is a cloud computing service offering on-demand, pay-as-you-go basis, computing, networking, and storage resources to consumers [36]. Typically, cloud computing service providers manage the underlying physical infras-

tructure while clients install the necessary tools, including the OSs, binaries, and middleware to host their programs or applications. VMs run on shared hardware resources consumed by many customers. Nevertheless, running VMs on dedicated hardware is also an option for customers, whereby at the hardware level, dedicated VMs that belong to other accounts or subscriptions are physically isolated. Also, they might share the underlying hardware with other VMs within the same account or subscriptions not dedicated to VMs. In addition to dedicated VMs, most cloud service providers provide dedicated hosts, which allow customers to build VMs on physical servers dedicated solely to them. This enables customers to employ server-bound licenses while adhering to all compliance and regulatory obligations. Dedicated hosts also provide more control and visibility. Customers use the same physical server over time and control how VMs are placed on top of it [36, 54].

Many cloud providers represent IaaS as a general-purpose computing resource with the limitless capability and ability to scale resources, in a matter of minutes, a large, diverse array of workloads, particularly in case of a sudden spike of workloads. IaaS is most commonly used for data analytics, data storage, business applications, data backup and recovery, data warehouse workloads, and dev and staging environments as well [138].

An IaaS pricing strategy, generally based on the consumption basis, offers granularity which means customers are charged for the resources they use. However, consumers willing to use cloud providers' services for long-term commitment contract terms- typically from one to three years can get discounts off the regular prices. In addition, many cloud providers offer unused computing resources at a discount for spot instances (VMs or servers). Spot instances/virtual servers stop, hibernate, or terminate when cloud providers need to reclaim the computing capacity or the hourly spot price rate exceeds a customer's hourly spot price request [105].

In contrast to traditional computing, IaaS is often more efficient regarding resource consumption, resulting in significant cost savings. Spinning new compute resources can be time-consuming because each VM is bundled with its OS, adding extra overhead. Although IaaS abstracts away the management of low-level hardware components, developers must still maintain VMs, runtimes, binaries, tool security patches, backups, and OS systems. Customers can preserve control of their servers with IaaS. On the other hand, customers cannot retain root or superuser access rights in a fully managed RDS database instance. Managing and maintaining many backend fleet servers in a large enterprise necessitates many dedicated workforces. Amazon Elastic Compute Cloud, Alibaba Elastic Compute Service, Google Compute Engine, IBM Cloud Compute, and Oracle Cloud Infrastructure- Compute are examples of IaaS services. Auto-scaling may take time to fire up a new VM with the OSs on it during periods of heavy traffic demand, which is one of the significant drawbacks [138].

2.4.2 Container as a Service

Container as a Service is an efficient and convenient way to deploy software efficiently and run containerized applications on its own user spaces without managing servers still being isolated from each other and their host machines. Containerized applications

run independently with minimal dependencies. Containerization helps developers to concentrate on developing business applications rather than hurdling with orchestrating and managing the infrastructure. Developers only need to build the application images to run on different environments, including Linux or Windows, and container service providers automatically enable the operation at any unprecedented scale. These features make containerization applications more portable. Testing applications across several platforms are cost-efficient. This makes it easier to maintain applications at scale and enables developers to build and release applications faster without compromising security. Containers can be easily deployed and run across private, public, and hybrid clouds due to their portability. Cloud providers usually use container orchestration and management tools to provision and manage enterprise-level containerized applications using Kubernetes, OpenShift, Docker Swarm, and Borg. Some commercial CaaS services are GCP GKE, GCP GKE AutoPilot, and AWS EKS. Besides hosting applications, CaaS can help with CI/CD to find defects early in the development process and boost project release velocity without fiddling with the CI/CD infrastructure [107].

2.4.3 Platform as a Service

Cloud service providers have dramatically changed how applications are built and run in cloud computing. In the PaaS model, developers can easily and quickly build and deploy apps using cloud vendor-supported programming languages or bring their own programming languages runtimes and frameworks in a fully managed platform without ever having to deal with the nuisance of constructing and maintaining the underlying infrastructure [139]. This also includes auto-scaling the computing resources from zero to planet-scale during sudden traffic spikes without concern about below or under-provisioning the computing resources. PaaS eliminates the need to manage OSs and hardware instead of IaaS. PaaS also includes monitoring and server management that includes OSs, runtime environments, middleware, database management systems, language stacks, backups, security patches, and upgrades whenever required to ensure customer services are secured, up-to-date, and working correctly. With PaaS, API development and management, Internet of Things (IoT), agile development, and DevOps, cloud-native development is simple since it offers built-in frameworks for cloud-native app development, supports for CI/CD automation, and a wide range of popular programming languages, tools, and application environments [34, 35, 139]. The primary benefits of PaaS are the faster time to market business applications, easily scalable in minutes, almost zero server management for the business apps, almost low to no risk while testing and adopting new technologies, leverage of SSL/TLS certificates without incurring any additional cost by default and traffic splitting and application versioning. In the PaaS computing pricing model, customers only pay for the consumed resources on a pay-as-you-go basis. Examples of PaaS are Google Cloud App Engine, Amazon Beanstalk, IBM Cloud Foundry and Code Engine, Heroku, Azure App Service, and Alibaba Simple Application Server. Overall, PaaS is designed to eliminate the cost of managing and buying the underlying physical hardware management, middleware, software licenses, development tools, and database systems

needed for development. Customers only need to build and manage their applications while cloud providers manage everything else that sits below the applications [35, 54].

2.4.4 Database as a Service

Database as a service (DBaaS) is a fully managed cloud computing service for hosting data using a preferred database engine[99]. Customers can use cloud databases with almost no administrative effort using this model [53]. Customers can set up the database backup window to take incremental backups regularly. They can use the maintenance window to apply any minor patches they choose. Besides cloud services, they can manage databases and database users, monitor performance-related indicators on the dashboard, and integrate databases with on-premise applications. Cloud service providers offer multiple levels of security for the managed database instances, including network isolation and data encryption at rest and in transit using SSL [99]. With DBaaS, cloud service providers handle the underlying infrastructure, which includes the data storage, database software, licenses, any needed software, data replication, automatic failover, minor updates for keeping the service up-to-date and secured as part of a subscription or pay-as-you-go fee-based pricing strategy[99].

Besides upfront purchases, customers are charged on a pay-as-you-go basis [99]. They can purchase the database instance upfront to commit to long-term contract terms ranging from one to three years with a discount on regular prices. In the DBaaS model, customers must pay for the underlying platform if the database instance spans multiple zones or regions. Examples of commercial DBaaS include Google Cloud SQL, Azure SQL Database, AWS Aurora, Alibaba ApsaraDB RDS, IBM Cloud Databases for MongoDB, and AWS DynamoDB. The data stored on the cloud database relies entirely on the service provider, albeit there is a risk of data unavailability if the service goes down [55]. With DBaaS, customers can not fully control the underlying database instance; they are not granted the superuser or root user privileges in contrast to an on-premise database server or self-hosted database instance on the cloud where customers have complete control over the database instance with much administrative work. Some cloud providers allow customers to update the database instance configurations with higher specifications, for example, adding additional memory and CPU, if required to handle load after launching the application at a later stage without provisioning a new database instance which offloads a considerable amount of database administration work. Cloud providers also provide service level agreements (SLAs) for database instance availability of 99.95% [109, 110]. Cloud providers handle the data storage growth incrementally overtime when necessary. Customers can seamlessly upgrade to the latest supported major database engine version in just a few clicks.

2.4.5 Software as a Service

Software as a Service (SaaS) offers cloud-hosted, on-demand, ready-to-use complete software applications, typically web-based applications, mobile apps, or desktop clients, where customers are charged for a pay-as-you-go basis or a monthly or annual

subscription fee [92]. As shown in Figure 7, the stack's top layer is SaaS. In the SaaS service model, customers receive a fully functional application that requires them to set up some application-specific settings and control user management [54]. However, SaaS vendor providers develop and maintain the applications, their deployments, and underlying infrastructures, everything about the service delivery, for instance, virtual servers, data storage, middleware, networking, OSs, binaries, and networking [54]. The account creation and management, granting access to customers, provide automatic upgrades and security patches to the software are managed by the SaaS vendors, usually invisible to customers. Depending on the type of application, some may need some customization work to set up before they can be used, including ServiceNow or Office 365. The vendors also provide SLAs to meet the agreed-upon service availability, performance, and security. Customers can also purchase on-demand resources at additional cost, adding more application users. SaaS can run on multi-tenant architecture in which one software instance can serve many users. The main advantage of using SaaS is dramatically lower upfront costs, easy application access over the internet, offline functionality, seamless integration, rapid deployment, on-demand scalability, upgrading business solutions quickly and easily, and forecasting the total cost of ownership (TCO) with greater accuracy. Examples of SaaS are Zoom, office365, DropBox, Slack, Oracle Netsuite, and Hubspot. According to a Gartner analysis, SaaS solutions sales will expand rapidly from US$ 270 billion in 2020 to over US$ 332 billion in 2021, with an estimated annual growth rate of over 23 percent [77]. SaaS innovative solutions will drive the growth of artificial intelligence (AI), autonomous IT management, blockchain, machine learning (ML), chatbots, augmented reality (AR), virtual reality (VR), and digital assistants to the Internet of Things (IoT) [92]. Some SaaS vendors migrate their on-premises business solutions to the cloud without enabling the full advantages of the SaaS cloud delivery model; call it SaaS. This type of SaaS model has some limitations, including the slow upgrades process, significant support bills, disconnected systems, high IT operational costs, performance issues- all of which prevent agility in business and, reduce innovations, and eventually lead to poor user experiences.

2.4.6 Function as a Service

Serverless cloud computing is a subset of cloud computing, while *Function as a Service* (FaaS) is a subset of serverless cloud computing. The event-driven serverless computing paradigm, or FaaS, focuses on enabling developers to create, deploy, and run microservices applications as functions without maintaining the underlying infrastructure [57, 123]. Application code or containers only run under the FaaS model in response to events or requests [79].

FaaS platforms execute a small piece of code in response to an event after users deploy code to a serverless platform. An HTTP request, a message in a highly available (HA) distributed asynchronous message queue service, or a single database query can all be considered an event. Functions are executed in FaaS in a stateless ephemeral container, and a single function's lifetime can be milliseconds. Because FaaS functions are stateless, state information is not guaranteed between function invocations [79].

FaaS platforms, for instance, GCP Cloud Functions, Azure Cloud Functions, and AWS Lambda, deploy, monitor, auto-scale, route traffic, aggregate logs, and manage cloud functions. FaaS, on the other hand, cannot entirely abstract away all operational logic from FaaS users. FaaS users can still update the Memory and CPU configurations of the container [57].

The Faas model ensures high service availability across zonal availability in each geographic region or multiple regional deployments without incurring additional costs. FaaS supports seamless integration with cloud services, including backend databases, application load balancers (ALBs), and data caching. Developers can upload application source code as a zip file or bring the application container image [157]. With Faas, microservices application code can execute in parallel, and each event is triggered separately, scaling precisely in milliseconds to less than seconds, from a few requests to virtually at any scale per second. Developers can optimize the compute resources, configuring the proper memory size to reduce the code execution time. Moreover, developers, if needed, may also keep the functions initialized and hyper-ready by letting the code execute concurrently to respond in double digits milliseconds. FaaS can be used for latency-sensitive backend API services, data processing, batch jobs or scheduled jobs, dynamic web applications, mobile backends, IoT backend, and machine learning [157].

In summary, distinguishing between different cloud computing service models might be challenging. DBaaS, PaaS, and SaaS could be serverless since servers are invisible to customers and not the customer's responsibility to manage servers compared to FaaS. CaaS can fall into serverless and fully manage based, for example, in GCP GKE clusters (fully managed based), customers manage the worker nodes, while in GKE AutoPilot (serverless, meaning that customers have no access to the cluster's node), the vendor manages them. However, the main difference between these service models is the shared responsibility, mainly operational accountability, between vendors and customers. Unlike IaaS platforms, serverless architecture offers substantial compromises in control, cost, and flexibility. With IaaS, consumers pay for the VMs and other resources needed to run applications from when they are provisioned until they are explicitly decommissioned, unlike a serverless platform. Modularizing an application forces developers to consider the cost of their code instead of scalability, and high availability, which have historically been the main areas of development efforts. PaaS, like FaaS, abstracts the server management from customers, but it differs from FaaS because it deploys functions as a deployment unit. In SaaS, vendors are ultimately responsible for securing their platform, including physical and application security, which vendors handle. Vendors are responsible for their data security. They do not own customer data and are not liable for how customers use the applications. As a result, they must ensure that malicious data is not exfiltrated, unintended exposure is avoided, and that malware is not introduced. In contrast to IaaS, CaaS uses containers as its primary resource rather than VMs. CaaS is an extension of IaaS because customers build, run, and manage applications on containers without contriving to build and maintain the infrastructure or platform. However, they yet require to write their application code and manage data. In CaaS, vendors deliver the container engines, orchestration, and the underlying compute to

clients as a service, allowing customers to control runtime and scale. Conversely, PaaS provides the platform to build, execute, and administer their applications while shifting the server management and scaling responsibilities to the vendors. Another example would be, after deploying a web application service successfully in PaaS, vendors provide a domain name with SSL/TLS certificates activated, which vendors manage. Customers never have to pay for the domain name or SSL/TLS certificates. A custom domain mapping feature is available from the service itself. In CaaS, depending on the vendors, customers are given either an IP address to access the service publicly or a domain name without the SSL/TLS certificates, which means that they must manage and buy their own SSL/TLS certificates, including a registered domain name to point the service endpoint to their registered domain. For instance, GCP Cloud Run provides a custom domain name with TLS certificates; however, GKE AutoPilot only provides the public IP address when an Ingress service is created. However, AWS EKS provides a custom domain name without SSL/TLS certificates after successfully creating an Ingress service with an ALB. In short, PaaS is commonly termed an application platform, while on the other hand, CaaS is termed a container platform. In PaaS and CaaS, data and application security is still the customers' duty.

The services used in the thesis fall into CaaS and PaaS models. For example, GCP Cloud Run, discussed in section 3.4, falls into a PaaS model, while AWS Fargate, AWS EKS, and GCP GKE, GKE AutoPilot fall into the CaaS model. The CaaS services are discussed in section 3.5, 3.6. The relational database services (GCP Cloud SQL and AWS RDS) utilized in this thesis fall into the DBaaS, discussed in section 3.4.

2.5 Continuous Integration and Continuous Delivery

CI/CD is a collection of processes to automate software development's builds, tests, and deployment stages. CI/CD automation reduces delivery times, offers a faster feedback loop for the software development teams, increases visibility, and enhances the stability of software with faster release velocity throughout the software development life cycle (SDLC), which leads to better customer satisfaction and product quality [60]. CI/CD consists of three steps: (i) Continuous Integration, (ii) Continuous Delivery, and (iii) Continuous Deployment. In CI/CD, continuous integration is abbreviated as CI, whereas continuous delivery and continuous deployment is abbreviated as CD [124]. Some popular CI/CD tools are Jenkins, CircleCI, Gitlab, Spinnaker, GoCD, Google Cloud Platform (GCP) CodeBuild, Amazon Web Service (AWS) CodeBuild, CodeDeploy, and CodePipeline [124]. The key concept of CI/CD is to make small changes to the codebase, build, test, and roll out a new production release faster and frequently to respond quickly to constantly changing business requirements. This section briefly discusses the main concept of CI/CD.

In Figure 8, the blue color represents that when a developer commits code changes to a version control system, a CI pipeline is triggered to build and test the application code. The green color indicates that build artifacts are stored in an artifact repository such as JFrog Artifactory, Nexus, or in a cloud container registry and storage service, including Amazon Simple Storage Service (AWS S3) or Google Cloud Storage. The yellow hue denotes the deployment of application code to a test or pre-production

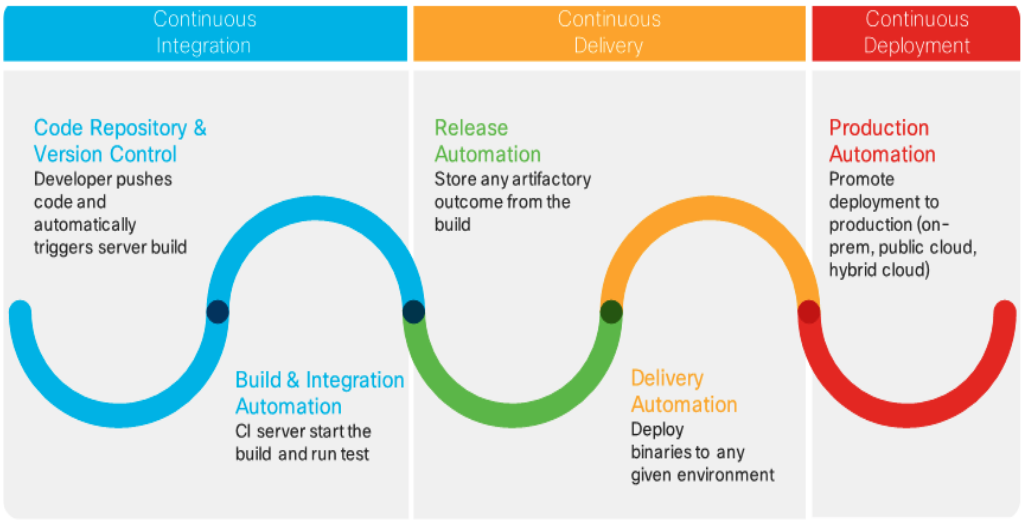

Figure 8: A typical CI/CD process [121]

environment for integration and performance testing. The red color means the code is deployed to the production environments using an automated pipeline. The deployment of new versions of code to pre-production or production environments can be a manual or automated operation, depending on the organization.

2.5.1 Continuous Integration

Continuous Integration (CI) is an automation process in software development to merge (integrate) application code changes and updates from various team members in a shared repository. When developers make changes to their application code before merging it to the main branch, an automatically triggered CI pipeline typically validates those changes by building the application. In addition, the CI pipeline runs different levels of automated tests, typically license checking, static code analysis, and unit and integration tests, to ensure the changes do not break the application. In most cases, teams configure the minimum threshold for the code coverage for the repository. If the new changes do not satisfy the code quality, the pipeline status is shown as failed, which means the code is not ready to merge into the main branch. This entails testing everything from functions and classes to the various modules that comprise the application. Continuous Integration helps to detect bugs at the early stage of software development in each iteration. For example, if the CI pipeline discovers any application code that breaks the entire or part of the application, CI makes it easy to detect those bugs at the early stages so that developers can get them fixed quickly and frequently [124].

2.5.2 Continuous Delivery

Continuous Delivery (CD) ensures the code base is always ready for production deployment. The CD is the intermediate phase of a release pipeline, which begins

with CI introduced in the previous section and finishes with Continuous Deployment described in the next section. During this phase, integration, performance, and functional testing are performed against applications in a production-like environment. In software development, various teams work with pre-production environments in addition to production environments, where CD aids in the effective use of automation to quickly push code to various environments [124].

2.5.3 Continuous Deployment

Continuous Deployment (CD) is the process of repeatedly building, testing, configuring and delivering software into production environments or end-users hands in a timely, efficient, and secure manner. The expanded form of continuous delivery, known as continuous deployment, automates software release from one version to another, with the option of adding a manual approval step to comply with regulatory or other control requirements[60]. Typically, continuous deployment supports the blue-green and canary deployment strategies, two of the most popular deployment strategies for minimizing risks and application downtime. The blue-green deployment strategy runs two identical copies of the production environment; a new version (green) and the existing (blue) version. This uses the load balancing approach to reroute 100% of traffic from the existing (blue) one to the new (green) one without any perceptible change for end-users. The blue environment will be pulled down once the deployment is thoroughly tested in green. If any incidents are discovered during monitoring, all traffic is redirected to the blue environment, which is still operational [23]. On the other hand, canary deployment uses a similar approach to blue-green deployment but adopts a slightly different method. Canary deployment is a strategy for releasing software in small, incremental steps to a small group of users or servers. The loadbalancer distributes a tiny fraction of users traffic to test the new version of the application, which is delivered to a relatively small percentage of Canary servers. The changes are rolled out to the other servers once the roll out has proven successful in a subset of the environment. When a deployment fails, the rest of the servers are unaffected, allowing the canary deployment more control over the application's downtime [125].

To summarize, in CI/CD, the CD refers to continuous delivery rather than Ccntinuous deployment [124]. The significant distinction is that code updates are available to end-users once all essential tests are passed in continuous deployment. In contrast, automation pauses when developers push to production in continuous delivery. The primary duties of CI/CD is summarized below [124, 128]:

- **Continuous integration** automates builds, tests, and integrates code changes in a shared repository.
- **Continuous delivery** automates code changes to production-like environments.
- **Continuous deployment** automates the deployment of new software versions to production environments.

2.6 Software Architecture Patterns

Software architecture patterns aid in defining the properties and behavior of everything from individual components to the overall system. This defines how a group of components work together to form the complete system, what role each component should serve, and how they should communicate. Business boundaries clearly define the scope of a well-designed software architecture pattern. The software architectural pattern help in the development of a robust system that is easy to integrate with other systems and reuse and replace parts of components as required. A well-defined design pattern can drive productivity and efficiency. In serverless cloud computing, microservice architecture demands architectural changes to implement Function-as-a-Service (FaaS) to leverage the benefits fully. Each microservice must be lightweight to maximize resource efficiency. The monolith and microservice architectures are briefly discussed in this section.

2.6.1 Monolith Architecture

A typical monolith application has a three-tiered architecture: a user interface, a backend that handles business logic, and a traditional database that stores user data. Usually, these layers are owned by different teams [39]. The entire application uses one type of database engine and one programming language to write the backend business logic. A monolithic application's processes are also closely connected, operated from a single binary, and run as a single service [39]. Within the application, components have interfaces and can communicate via interfaces, function calls, and method invocations.

A modularized monolith can assist organizations if each module's binding context is adequately defined and parallelism can be achieved. For example, Shopify [94], an online e-commerce site, demonstrated how it transformed from a typical monolith to a modularized monolith architecture. They achieved this by extensively refactoring their code base, allowing for code reuse, identifying dependencies, and isolating them through decoupling and enforcing the boundaries for each module.

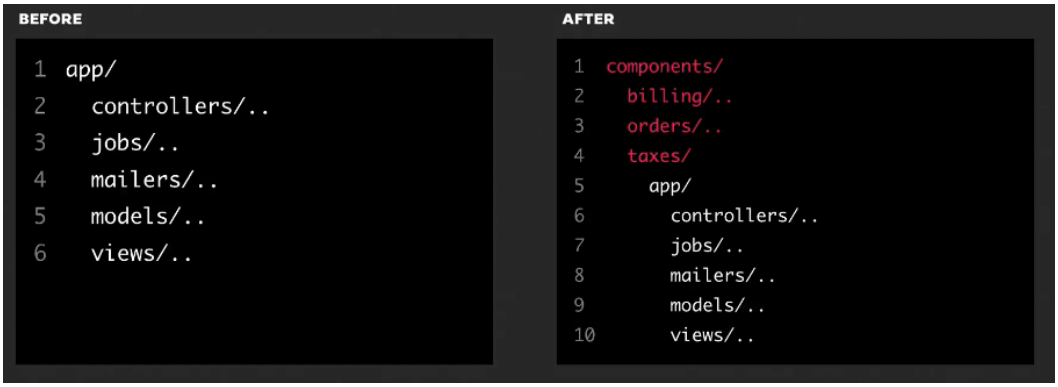

Figure 9: Modular monolith architecture codebase example at shopify [94]

Figure 9 displays the codebase of Shopify before and after it was restructured. On

the left-hand side, it can be seen that a large monolith codebase was created using the Model-View-Controller (MVC) design pattern, where the whole application's business logic was placed under the controller's folder [94]. The right-hand side shows that the entire application was split into smaller components or modules, with each component having its controllers to handle the business logic. Several processes are highly dependent on each other, which increases the likelihood of single-process failure. Neither a single module can be deployed separately nor executed independently. Nonetheless, many applications start with comparatively smaller codebases at the early stage with a single, lightweight executable binary entity, but over time, adding or refactoring business logic becomes more complex. Adding a new technology stack, for example, a new framework and programming languages or a database engine becomes very difficult. This complexity becomes a barrier to experimenting with new innovative ideas. A minor change in the whole application must be deployed, and it directly affects the entire application, and changes need to be fully deployed to production. Eventually, this slows the release frequency, increasing the cost to roll out a new feature to production and stay competitive with its competitor. In addition, this adds the risk for the application's availability and the scalability cost since deploying the whole application requires powerful containers or VMs [93, 95]. Some possible drawbacks of the monolith are listed below [41, 88]:

- **Scaling**

  Components or sub-components (modules or sub-modules) can only be scaled by scaling the entire application. Every monolith application must often utilize the computing resources correctly because a subset of the application may experience more traffic and others not- due to the end user's interest. Customers on e-commerce sites, for example, are more likely to peruse things than to buy them. Because parts of the system cannot be scaled out or scaled in, this results in resource underutilization.

- **Testing**

  Testing becomes more difficult and expensive as the code base expands over time. Typically, each tiny change must pass regression tests to guarantee that nothing fails before being deployed to production, which can take a long time for an extensive monolith application.

- **Deployment limitations**

  The monolith application works as a single unit in the form of a combination of all its modules, which means every change needs to deploy the entire application. Modules never work independently, so if something goes wrong that requires the application to restart, usually restarting time is longer in contrast to microservices. Technically, this means all the users are affected during a restart, and their sessions are terminated. Running a monolith application in containers can avoid application downtime in such situations. Overall, deployment duration increases as the code base increases, as does the container image size.

- **Technology restrictions**

  Upgrading technology stacks is always challenging; it increases risk and costs because all components, including the database engine, must use the same technology stack. In most cases, this necessitates rewriting the entire application using new frameworks or programming languages, which is costly and time-consuming. This hinders the company's ability to apply new tools and innovative ideas to stay ahead of the competition. New tools must be compatible with the application, which is typically a challenging operation due to the changes that must be made in the sub-modules.

- **Dependency management**

  With a monolith, components are interdependent, and changing and upgrading libraries might break other dependent components. Maintaining multiple versions of a single tool or framework adds extra overhead and increases the cost of keeping them up-to-date whenever a new security patch is released.

- **Maintainability**

  Large monolithic applications become more complex to maintain over time, resulting in fewer releases. This has a direct impact on the productivity of teams. Developers leave organizations often, raising concerns about increasing the technical debt, and new developers finding their way around a massive codebase might take a long time. Finally, new problem patches take a long time to implement because of the code's complexity and reliance on other libraries.

A monolithic architectural pattern can work well in some circumstances, despite the fact that it always has limitations. In monoliths, components are not distributed, unlike microservices. Components can communicate directly among themselves as they reside within the same application. Unlike microservices, they do not need to traverse the internet to communicate with other components. Testing, debugging, and log tracing are much more straightforward than distributed microservices. The following section discusses the pros and cons of microservices. However, container monolith applications can empower many benefits because cloud providers offer powerful container specifications or VMs that can rapidly auto-scale and deploy applications. Updating a container image is faster and network efficient. Container boot time duration is smaller and speeds up the release rollout process. Cloud providers

offer container orchestration and management tools, simplifying container life cycle management. By fully leveraging the container orchestration tool, deploying a newer version to production and rolling it out to a previous version using some CI/CD pipeline became remarkably easy. Monolith architecture can scale horizontally [85]. So this architecture is unable to scale when data volume increases. If one application runs multiple instances, each instance must access the data, which uses caching while increasing the I/O traffic and memory consumption. Studies suggested that monolith can be a good choice for organizations if the below requirements apply to them [45]:

- Team is smaller
- Relatively smaller codebase
- Communication is rather fast and easy
- Geographically, teams are located close to each other

2.6.2 Microservice Architecture

Microservices is a software architectural pattern in which the entire application is segregated into smaller units of independent service components. These service components are developed, owned, and maintained by self-contained teams, and they communicate with each other over the well-defined lightweight REST APIs, event streaming, and message brokers. Each service component is finely grained with its life cycles, independently deployable, autonomous, and loosely coupled, which collaborate in distributed systems. Furthermore, each service can be built on its technology stack and programming languages, including database systems. Individual microservices are designed to serve one business logic, and the end users or system can access via the APIs. Organizations that have adopted the microservices approach can launch new products and add new functionality or features without touching the entire application, increasing the software release velocity, reliability, resilience, and safety. In addition, individual microservices scale independently, reducing the overall cost and waste associated with the entire application life cycle.

The authors stated that, overwhelmingly, the organizations adopted microservices approaches to software development to work (WoW) with extensive software systems in which scaling was a significant factor [42]. They had a great interest in improving changeability, independent microservices manageability, and simplified testability but particularly replaceability of each microservice in the entire application instead of finding a universal process. The authors mentioned the successful microservices transformation case of Gilt, an online retailer in the fashion industry, which started with a monolithic web application developed in Ruby on Rails in 2007 [40]. By 2009, their application could not scale in response to serving on-demand traffic. To overcome this bottleneck, they split out the entire application into smaller, independently deployable units by which they were able to deal with the sudden spiked traffic. Today, they run over 450 microservices in their single application, on which each microservice runs on multiple separate VMs. Some large organizations and many more have already

transitioned from monolith to microservices architecture, such as Samsung, Uber, Netflix, Amazon, Github, SoundCloud, and LinkedIn, as their existing large monolith codebase limited the rapid feature development, deployment and maintenance, which became a considerable overhead [47, 68, 69, 75].
According to an AWS whitepaper [96], microservices are not a new concept, over a decade old concept, in software development and a combination of multiple concepts, which are listed below:

- Service Oriented Architecture (SOA)
- Agile Software Development
- API or Event Driven Design
- DevOps

In many cases, microservices follow a footprint of Twelve-Factor-App's design pattern. In cloud computing, in an advanced serverless microservice architecture, an application can be built using thousands of microservices with almost zero administration, which reduces the overall cost without worrying about provisioning, configuring, and monitoring the underlying infrastructure, lessening the architectural burden, and simplifies the deployment. Often the greater complexity comes with handling inter-service communication, auto service health status, and replacement of unhealthy services but not the services themselves [96]. Some services offer inter-service discovery and communication so that services can talk to each other in distributed systems, for example, AWS ECS, AWS App Mesh, and Alibaba Cloud Service Mesh (ASM). In distributed microservices, data management becomes challenging and has some trade-off consistency about performance. Typically, business transactions in a distributed application architecture span multiple microservices wherein a single ACID (Atomicity, Consistency, Isolation, Durability) transaction might end up with partial execution. To handle such scenarios, AWS commonly leverages the distributed saga pattern to redo partially processed transactions. Conversely, a saga orchestrates a botched transition by orchestrating compensating transactions to reverse the changes of preceding transactions [96]. Commercial examples of such technologies are AWS Step Functions, GCP Workflows, Alibaba Serverless Workflows, and so on. Microservices architecture on public cloud platforms allow development teams to take advantage of event-driven architecture patterns which complement cloud-native application development.

- **Technological freedom**

  An oversized system comprises many microservices where many, if not each, are built using different technologies. This brings much flexibility to choosing the right tool without sticking to a more standardized technology stack system-wide. When a particular stack or microservices performance hinders the business value, organizations can use a completely different technology stack to achieve the performance boost. Technology is growing faster; therefore, to stay ahead

of competitors choosing the right tool can help boost performance, better user experiences, reliability, software security, and less time to roll out the product to markets. Trying out and adopting new tools or technology might bring multiple degrees of risks and overheads. On top of that, teams associated with developing, testing, and deploying microservices need coaching and training; in typical cases, the learning curve is significantly steep. For example, if needed, teams can implement a data analysis service component using Python instead of Julia or R programming languages. Moreover, each service can have its database engine, where one could benefit more from a document and Memcached database and the other from an RDS and Redis. Often we have experienced organizations sharing the same backend database for all their microservices for a single application. Still, the preference is to keep one database for a single microservice.

- **Resilience**

  In a microservices architecture, if one or more components or sub-components of a system stop working, the rest of the system can still work, unlike in monolith architecture. It is possible to run a monolith system on different machines to eliminate the chance of failure. Contrarily, we can leverage microservices to create fault-tolerant systems, and guarantee high availability regarding networks, VMs, and container failures.

- **Flexible scaling**

  Users deploy and run smaller systems units independently with microservices, allowing us to run services on less powerful containers, VMs, or hardware. This allows organizations to optimize costs efficiently and effectively while provisioning services at scale on demand. This ensures that they are not provisioning containers or VMs with extra CPU, memory, and storage resources; in a situation where it is unnecessary. One of the most significant advantages of scaling in microservices is that it only scales part of the systems based on the traffic pattern without scaling the rest of the system. On the other hand, a monolith architecture scales the entire application, which requires powerful computing resources and is costly.

- **Composability**

  With a microservices system, composability reduces the overall system development time, allowing compound benefits of function reusability over time by outside parties or new systems. A Monolith system limits composability because it offers a single interface to be consumed from the outside.

- **Organizational alignment**

  A common problem with managing large codebases across different teams in different geographical locations brings extra challenges, risks, and overhead. It is easier for smaller teams co-located to work on a smaller codebase while being more productive. With a microservices approach, teams work in smaller code bases of a system. Typically, small cross-functional agile teamwork often leads

to better team throughput. Co-located teams, if needed, can switch ownership to different services between teams.

- **Optimizing for replaceability**

  In a large organization, the code bases of a monolith system grow over time, which usually introduces dependencies. Nonetheless, team members leave the company as knowledge gaps widen, and no one wants to touch the codebases to refactor to improve the logic. The codebase for a microservice is supposed to be smaller, so the hurdles to modifying or rewriting the entire codebase are low.

- **Ease of partial deployment**

  In practice, monolith system deployment introduces significant impacts and high risk and sometimes requires service downtime if the platform lacks support. Ideally, the entire application must be rolled out to production for a new revision release if a single line of code changes the entire code base, which is expensive. In a microservices architecture, changes are only deployed to its associated services without touching the rest of the system, which reduces the cost and deployment time. If a service fails after deployment, it can be rolled back to the previously deployed version faster, in typical cases, in just a few minutes. According to the author, a German company, mainly working in digital media named Haufe-Lexware, was able to reduce the platform downtime to thirty minutes from five days by automating the problematic areas of the deployment process during a transition period from monolith to their microservices journey [39].

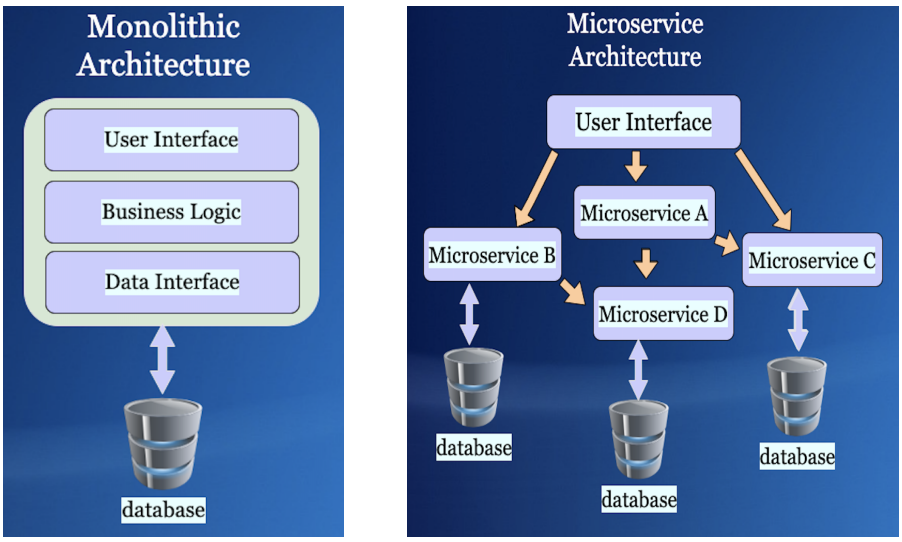

Figure 10: Monolithic and Microservice Architecture

Figure 10 illustrates monolithic and microservices architecture differences. It can be seen that the monolith has three layers: a frontend, a backend layer (the business

logic layer), and a data layer [45]. The data layer communicates with the persistent backend database. However, each microservice can have, optionally, its database engine, which means developers can change any table structures and database schema within a single microservice without depending on other microservices.

The author stated that there are two different kinds of challenges to tackle while adopting microservices; one of them is the technical challenges, and the other one is the organizational challenges [40]. A microservice architecture that is too fine-grained can cause performance degradation because of the added network latency.

Reusing code in distributed microservices can be challenging because each service has its repository. The options are to copy code, push the shared library for standard functionality into a service, or split libraries. On the other hand, monolith simplifies code reusability itself. Monitoring microservices is more complex as opposed to the monolith. Developers and DevOps teams must choose the right tool to aggregate monitoring metrics logs from each distributed microservices for the infrastructure and the application logs. DevOps teams also need to deal with tracing information across multiple microservices [39]. When a client requests a service, it generates a correlation ID (also called request ID) [87]. This correlation ID must be passed into the message to all downstream services that handle the request. Microservices run multiple copies of each service instance, including the read replicas of the database instance. This means DevOps teams have more services to maintain. Introducing different programming languages and frameworks (Polyglot) for different microservices adds additional costs in development and maintenance. Distributed services run over the network independently to ensure minimal latency, but fast and reliable network connections are required. On top of that, data security at rest and in transit (data encryption) is essential to avoid any security breaches across distributed services. Function-as-a-service microservice patterns can not avoid code duplication. In microservices, systems testing is complex compared to monolithic applications. Some organizations might end up implementing monolith in distributed systems in the name of microservices; often, organizations need to define a well-defined bound context for each service. Many cloud providers offer DevOps solutions that can integrate into other cloud services, abstracting the complexity of setting up a CI/CD pipeline and helping organizations have a faster deployment process for monolith and microservices architecture.

Feature	Monolith	Microservice
Partial Deployment	Not possible. The entire application has to be deployed	Possible. Each microservice can be deployed independently, regardless of the entire application
Release Frequency	When the codebase grows, release frequency reduces, but in the initial stage, it is faster	High release cadency but can be slower in the initial stage
Technical debt in codebase	As the codebase grows, so do the technical knowledge gaps	Minimizes technical debt.
Resilience	If a single error occurs, the application stops working	If one service stops working, the rest of the services work
Performance	No communication overhead, but existing technology might underperform due to tools limitations	Several tools boost performance, but communication adds overhead because of their distributed nature.
Refactoring codebase	Refactoring is time-consuming and expensive due to the tightly coupled codebase. One small change can break other parts of the codebase	Refactoring is easier since the codebase follows the Single Responsibility Principle (SRP)
Choosing the right tool	Hard to replace an existing programming language or framework without a big rewriting as the codebase grows	Changing to a different programming language or framework is relatively easy because of the small codebase
Scalability	Part of the application cannot be scaled. The entire application has to be scaled	Scalability can be performed on a microservice basis
Monitoring	Easy to trace calls and monitor from one place	Different services need to be aggregated in a centralized monitoring system
Integration with external systems	Difficult.	Much easier
Testability	Easy	More complex
DevOps Skills	Some basic levels of DevOps skills are required	Good DevOps skills are required

Table 3: Comparison between the monolith and microservice architecture

Table 3 shows the pros and cons of the monolith and microservice architecture styles. Based on the data in Table 3, we can draw the conclusion that a larger codebase is better suited for the microservice architecture. On the other hand, monolith might be a better fit for a smaller codebase if the codebase remains that way for many years [45]. The following chapter briefly covers the pros, limitations, and application scenarios of serverless computing. We also recapitulate the container-level services utilized in this study.

3 Serverless Cloud Computing

This chapter introduces serverless cloud computing and some of its widespread use cases, benefits, and drawbacks. We cover the GCP Cloud Run's core values and functionalities in section 3.4. Section 3.5 discusses the AWS EKS service with AWS Fargate. GKE AutoPilot was covered in section 3.6. Furthermore, at the end of section 3.6 we highlighted and contrasted the features of AWS Fargate, GCP GKE AutoPilot, and GCP Cloud Run exploring their pricing policies.

Defining serverless succinctly is tricky in a concise manner because the definition may coincide with the definitions of CaaS, PaaS, DaaS, and SaaS [38]. Based on the various degrees of developers' duty and authority over the infrastructure, serverless computing can be succinctly explained. The term *serverless computing* (commonly known as *serverless*) refers to a cloud computing execution paradigm in which end users simply delegate the operational responsibilities to a cloud service provider, for instance, the underlying computing capacity, high availability, creating on demand resources, elasticity, other resources [38]. Serverless does not imply that no servers are running; instead, it defines customers' experience with such servers, as servers are not visible to host their application code [79].

The underlying backend infrastructure, maintenance of servers or infrastructure, and any operational tasks like provision, deployment, schedule, scale, backup data, and applying security fixes are the responsibility of cloud service providers. Without worrying about managing servers, serverless allows developers to spend more time optimizing and developing the application code. Because serverless requires no extensive knowledge of the underlying infrastructure, developers can undertake operational tasks in smaller businesses. However, they may set up, configure, and manage the automated CI/CD pipeline as part of the operational tasks.

With serverless, the cloud service provider's responsibility is to guarantee that on-demand computing resources are accessible at any scale to run customer applications or respond to events. They also need to ensure that auto-scaling scales resources up or down in response to the sudden spike or decreases in incoming traffic surges.

Customers must pay for the consumed runtime resources while running the application runtime never pay for idle computing capacity. However, when the application receives no incoming requests for a brief spell, they need to scale resources to zero [126].

In conclusion, serverless technology frees businesses from the burden of managing servers, allowing them to focus on developing interesting, creative ideas, creating cutting-edge applications at a reasonable cost, and releasing them frequently to the market sooner and at a more affordable price, resulting in increased turnover, and enhancing customer happiness [67].

The author wants to clarify that cold start and cold start instances refer to the same thing, while warm instances as warm start instances in this thesis. This is used for the convenience of addressing them the same way.

3.1 History

In 2006, AWS launched the beta version of EC2, a service offering for VMs [153]. Soon after, in 2008, Google launched its first product, Google App Engine (GAE) [154]. GAE is a fully managed PaaS-based service offering that abstracts away the underlying server management for the users [59, 155]. As a result, Google became the first cloud provider to allow developers to run code on their platform without having to manage servers. With AWS Lambda, AWS launched the first serverless offering in 2014, making them the industry's first serverless computing platform. In 2016, after the introduction of AWS Lambda, serverless computing began to gain popularity in the software industry as a way to construct modern web applications. Following AWS, several cloud service providers entered the serverless computing field with their offerings. Initially, serverless computing offerings were confined to FaaS. Serverless is expanding its application spectrum with serverless offerings from cloud vendors. Microsoft Azure Cloud Functions, GCP Cloud Functions, GCP Cloud Run, AWS Fargate, and IBM Cloud Functions are examples of commercial serverless computing solutions. Apart from commercial serverless offerings, there are a few popular open-source serverless solutions for running workloads in private clouds, such as Kubeless, Fission, Knative, and OpenFaaS [79].

3.2 Advantages and Limitations

The most important perks and limitations of using serverless architecture are introduced in this section. The following are some of the most commonly cited advantages of serverless computing [4, 38, 61, 127].

- **Efficient resource utilization**

  Resource consumption is more efficient because container lifetimes are often shorter in serverless platforms. There is no need to reserve computing resources for a specific consumer when there are no active requests because resources are allocated on demand. On the other hand, service providers share computing capacity between consumers and switch container instances or workloads inside the distributed cluster. On the converse, research reveals that a large portion of computing resources is underutilized [80]. For example, resources are allocated as needed to handle tens of thousands of inbound requests over some time. Furthermore, resources are assigned elsewhere when no active requests are processed. In traditional IT infrastructure, this would be a substantial overhead because the ALB might need to be capable of handling tens of thousands of concurrent connections with a small number of servers. A standby ALB could always operate in typical IT infrastructure, even if there is no incoming traffic to provide automatic failover capabilities, which wastes significant computing power. Serverless computing can potentially make computing "greener" by reducing the amount of computing power wasted on idle capacity.

- **No compute time cost**

One of the notable monetary advantages of serverless computing is that customers never pay for running computing resources, such as containers or VMs, when the application is idle or not running. Furthermore, no costs are involved in provisioning, creating, and terminating new containers to scale the application, allowing users to scale up and down from zero to thousands of containers and vice versa with minimal labor and expense. There are no costs associated with serverless database services if the database is inactive.

- **Effortless auto-scaling within seconds**

  Applications hosted on serverless platforms can scale and heal automatically within seconds without user input. Users no longer need to consider how many application instances must run or how to handle load balancing when adjusting application instances. In the event of a spike in inbound traffic, the application automatically scales up instances required to cope with the increased demand.

- **Zero server Ops**

  Serverless abstracts server management from the users, allowing them to concentrate on adding more business logic and tightening the application security.

- **Minimal operational costs**

  An application employing serverless architecture minimizes operational costs because there are no servers to manage. Furthermore, in smaller businesses, developers can handle operational tasks combined with their development tasks without the help of a specific team of system operation engineers.

- **Fast to market and stay competitive**

  Using a serverless microservices architecture with rich, intuitive features at more affordable prices, organizations can construct and market faster customer-oriented services focusing on innovation and customer satisfaction, resulting in increased revenue and enduring client relationships. This is because releasing a new application does not include uploading source code to servers or configuring the backend. In a matter of minutes, a new revision of the application can be rolled out if a application new functionality added or updated. Developers can update code all at once, albeit doing so is unnecessary. This enables businesses to adapt to remain viable and provide the same level and QoS as their competitors.

- **Cheaper alternative**

  An application using serverless architecture provisioning resources on-demand and consumers paying for a pay-as-you-go basis, when appropriately managed, can reduce overheads significantly in any organization. Serverless offloads infrastructure management and reduces the workload of system operations engineers, allowing them to increase productivity and streamline operations without spending much money upfront. According to the author's findings, in 41% of use cases, the cost is more important than the performance. Compared

to performance-focused use cases, cost-focused use cases are twice as famous (23 percent). Cost and performance are equally important in 15% of the use cases. Finally, for 22% of use cases, the trade-off remains unknown [46].

Because serverless computing has many advantages, businesses are adopting it for many applications. However, there are also some applications where serverless is not a good fit or where technological and commercial trade-offs must be made. The drawbacks of employing serverless computing are outlined below [4, 38, 61, 127]:

- **Introduces security concerns**

  With serverless computing, resources are shared between consumers (or multi-tenancy), meaning that code from other consumers is executed on a single server at a given period. Unless multi-tenant servers are configured correctly, data exposure is risky.

- **Challenges in monitoring and application debugging**

  Debugging is tricky as the backend processes are invisible to developers, and the application is split into functions.

- **Not ideal for long-running batch process**

  The cost of hosting an application with long processes in a serverless infrastructure may be higher than in a traditional one because vendors bill by the time code runs. Therefore, running long-running batch jobs in a serverless architecture is not recommended because the cost could be too high.

- **Performance may be affected due to startup latency**

  With no constant traffic pattern, serverless apps will not always run. In such cases, serverless applications need a *cold start* which means that a container instance must be created to process the request, as no instances are available to do so. As a result of the cold start, the application's performance may suffer, and the end-user may encounter some latency. A delay of milliseconds to seconds is expected during a cold start since it has to create on-demand resources and initialize the application code in newly created containers. Even while the delay would be insignificant in long-running batch activities, this directly impacts the user experience, which may not be acceptable for time-sensitive applications such as financial trading applications, mobile apps, or the application backend. The pattern of incoming requests determines cold starts; for example, applications with consistent incoming requests are unlikely to experience cold starts. On the other hand, cold starts are more likely to occur when traffic patterns are irregular or shift significantly.

- **Vendor lock-in**

  The more backend services a vendor is given access to, the more reliant the application becomes on them. It may be more difficult to switch vendors after adopting a serverless architecture with just one provider, especially if they each have slightly different capabilities and processes.

- **Lack of uniformity, guidelines, best practices, and documentation** Because serverless is still a novel idea, shortage of expertise and knowledge may raise hidden issues structuring the source code and repository management. However, as more research is undertaken and serverless computing becomes more common, the situation is improving.

  Aside from the disadvantages mentioned above, an IBM serverless study revealed certain hurdles in using or implementing a serverless [67]. When asked what obstacles their organisations faced in adopting or expanding serverless computing, more than half of the users highlighted a number of issues, including [67]:

  - Security concerns
  - Complexity of serverless architecture
  - Difficulty predicting performance in production environments
  - When developing serverless applications, there is uncertainty about the time and expenses associated.
  - The cost of running applications with long-running processes is higher.
  - Determining which applications will profit from serverless frameworks is challenging.

  Furthermore, when non-users were asked about their barriers to serverless adoption, about 25% of them concluded the following [67]:

  - Insufficient advocates in the organization
  - Uncertainty over the adoption process is involved regarding time and costs
  - Necessary security requirements are not in place
  - Learning curve is quite steep and complex for serverless architecture
  - Difficulty in managing, sharing, and securing data
  - Lack of serverless expertise in the organization
  - Moving applications from development and testing to production is difficult.
  - There is no clear mechanism to calculate Return on Investment (ROI) or track specific advantages.

In conclusion, the IBM survey found that despite its shortcomings and challenges, users and non-users of serverless architectures concur that the technology has much to offer. Additionally, both groups believe that serverless application development will advance substantially during the next two years in their respective businesses [67].

3.3 Use cases

Serverless is well-suited for a wide range of workloads. Serverless and traditional systems are interchangeable in terms of functionality. Other non-functional variables influencing when to utilize serverless include the level of control over operations necessary, cost, performance, and the nature of the application workload. However, serverless is suitable for the below-listed workloads [37]:

- Unpredictable variance in scaling due to sporadic or irregular inbound traffic patterns.
- Asynchronous and concurrency is a requirement
- Stateless and ephemeral workload with shorter startup times.
- The ability to boost developers' productivity and release new software versions multiple times daily.
- Microservice workloads that execute a series of steps
- Scheduled batch jobs require intense parallel computation, IO, or network access.
- Mobile backend and web apps with HTTP REST APIs
- The ability to respond to messages and scale up or down as needed, for instance, Sensor data for IoT.
- Stream processing workloads

The workload types listed below for serverless computing might not be appropriate:

- Tasks that run for a long time and consume many resources. AWS Lambda, for example, allows users to allocate up to 10GB of memory and six vCPUs per container. GCP Cloud Run, on the other hand, limits each container to four vCPUs.
- If latency in seconds is not accepted due to cold starts.

The first example would be image and video manipulation to boost performance for any service. For example, when users upload images or videos to any web service, a serverless function is triggered to resize images, provide dynamic thumbnails, and adjust video transcoding for any target devices. Additionally, when users upload artifacts to cloud storage, a FaaS function can help perform a security scan to detect malicious objects. Other use cases include when users upload HR data to some cloud storage. A web application with erratic incoming traffic request patterns is another example. In a serverfull Kubernetes cluster, for example, worker nodes (VMs) run continuously (unless otherwise defined) even if no incoming traffic is received, implying that users are charged for the computing resources they consume. Users do not need to provide worker nodes in serverless Kubernetes clusters to deploy application

Pods, meaning they only need to pay for the time; pods consume computing resources. Cloud service providers handle provisioning, maintaining, and administering the needed nodes (VMs) within the clusters to allocate Pods on demand. The third example would be to create cloud-based data processing apps that transform and store unstructured data as it arrives. From Google Cloud sources, transformations can be triggered. An event is triggered and delivered to a Cloud Run service when a *.csv* file is generated. The data is extracted, structured, and stored in a BigQuery table [136].

IBM's and Insights MDI's survey reports showed that more than 1200 IT experts from mid to large-sized businesses, including developers and executives, participated [67]. According to the report, 37% of users utilize serverless for customer relationship management systems. This is nearly twice as much as they used on the supply chain (20%) and just over 1% more than they used on data analytics and business intelligence (36%). Serverless was used by 30% of users for databases and HR applications, while finance stood at 31%, just 1% higher [67].

3.4 GCP Cloud Run

GCP Cloud Run is a serverless container platform allowing developers to develop and deploy containerized applications at any scale without managing servers. Developers can write applications in any language of their choice. Although they can bring any libraries, dependencies, or OSs with the applications, they can also bring their binaries. Cloud Run provides incoming traffic management capability when deploying a new application version. It allows developers to gradually release a new application revision, divide the traffic between revisions, and even roll back to an earlier iteration. Cloud Run delivers a stable Hypertext Transfer Protocol Secure (HTTPS) Uniform Resource Locator (URL) after the initial deployment of a service. Each Cloud Run service has a secure HTTPS URL. Furthermore, it manages and controls Transport Layer Security (TLS) and automatically renews the certificate before it expires [136].

Users can choose a particular region but not the zones to deploy a service using Cloud Run. A region may consist of one or more zone. Data and traffic are load-balanced automatically across zones as necessary within a region. Each zone has a scheduler to provide auto-scaling capabilities. In case of a zonal failure or outage, a scheduler will allow the extra computing capacity in-zone as it is aware of the load received by the other zones. Data plane operations are unaffected by zonal failures since Cloud Run stores data and the deployed container regionally. In the case of a zonal failure, traffic is automatically redirected to the other zones. Because Cloud Run does not replicate data across regions, in the event of a regional outage, the Cloud Run service will be unavailable until the problem is rectified [132].

The developer workflow for Google Cloud Run may be broken down into three simple steps. First, developers write the application in their preferred language. The application must have an HTTP server to start the application. Second, the application must be built and packaged as a docker container image. The container image is then deployed to Cloud Run as the third and final step. Cloud run creates a secure invokable HTTPS service endpoint and a unique resource name [52].

The GA of the second-generation execution environment was made available by

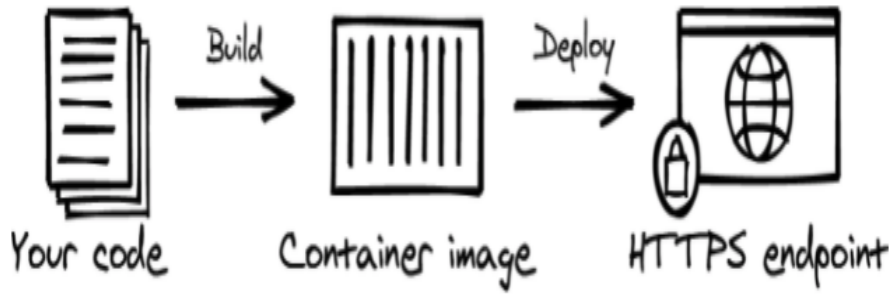

Figure 11: GCP Cloud Run developer workflow [52]

Cloud Run on December 6, 2022 [86]. By default, Cloud Run services run in a first-generation execution environment for the cloud run services. It emulates most OS calls but not all and allows for faster cold start times. However, users can change to a second-generation execution environment if they want. Nevertheless, the jobs that run on the second generation can not be changed for the cloud-run jobs [135].

After assessing the advantages and disadvantages between the first and second-generation execution environments, we presented them below as highlights [134]:

- We can use less than 512 MiB of memory, but a second-generation execution environment requires at least 512 MiB.
- We anticipate a massive cold start due to infrequent traffic, necessitating frequent scaling out from zero. Second-generation execution environments are faster under steady load but take longer too cold start than first-generation execution environments.
- The second generation performs better and supports a network file system than the first.

The key advantage of adopting the first generation over the second is the faster cold start time, which reduces container provisioning time while dealing with burst traffic. The author extensively researched GCP Cloud Run to comprehend its salient characteristics fully. The following is a summary of the key feature findings:

- **Concurrency**: Developers can change the concurrency configuration. A container instance can receive up to eighty concurrent requests simultaneously by default. In case of high resource utilization, for example, when a container CPU is heavily used, Cloud Run may not send requests to a container instance to avoid performance degradation. Developers can also limit concurrency by configuring one request to be received at a time by a container instance in the following situations where [114]:
  - An application is CPU or memory-intensive, where each request consumes the available CPU or memory.

– An application cannot handle multiple requests concurrently, for example, if it depends on the global state where multiple requests can not be shared.

Because many containers must be created to manage the abrupt traffic surges, setting concurrency to one will likely harm the application's scaling performance.

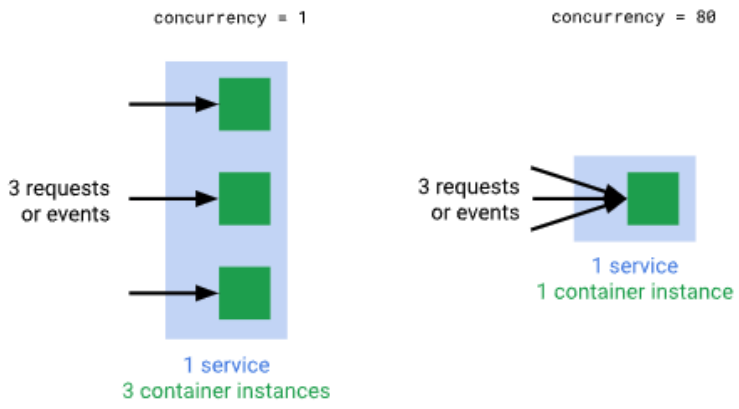

Figure 12: Comparison between different level of concurrency [114]

The left-hand side of figure 12 shows that when concurrency is set to one, each request is handled by one container instance. On the right-hand side of the figure, it displays that one container instance can handle three simultaneous incoming requests. This figure shows how concurrency settings are crucial in dealing with incoming requests.

A case study demonstrated how concurrency settings to different numbers for the same Cloud Run application affected the total number of container instances needed overtime to handle requests.

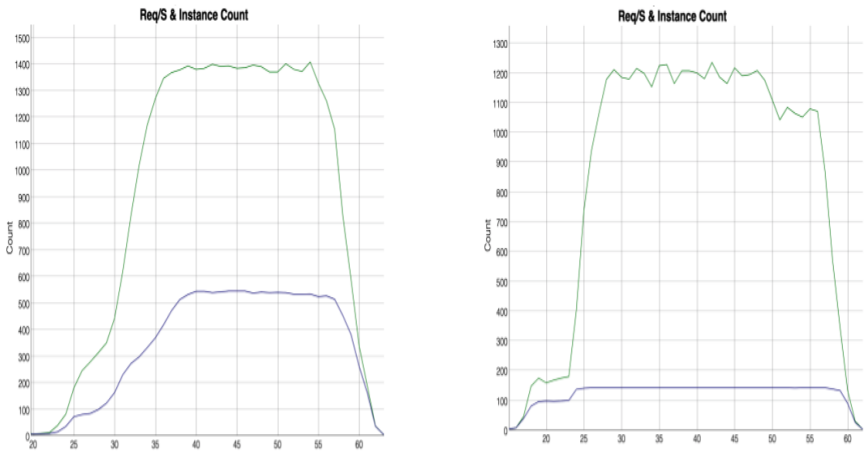

Figure 13: Comparison between different concurrency settings [114]

In figure 13, the green line represents total requests over time, while the blue line shows the total number of container instances needed to handle the requests. To experiment, 400 clients made three incoming requests per second to an application running in GCP Cloud Run, in this case, study [114]. We can see that when concurrency was set to a maximum of one, hundreds of containers were needed to handle requests. When concurrency was set to eighty; however, comparatively fewer container instances were required to handle the same volume of incoming requests, as shown in the right hand.

- **Container instance security**: A Cloud Run container instance is sandboxed using gVisor [116]. Studies show that gVisor is arguably secure compared to runC (runC is a lightweight container runtime) [51]. GVisor is an open-source project developed in the Go programming language, which incorporates an additional layer of segregation between the host OS and containerized applications [115]. GVisor works as a guest kernel, intercepting application system calls without virtualized hardware translation. User Mode Linux (UML) and gVisor are similar, but UML internally virtualizes the hardware, leaving a fixed resource footprint. GVisor offers a customizable resource footprint depending on threads and memory mappings while reducing the fixed cost of virtualization [115, 116]. This could decrease performance for system call-intensive applications. GVisor currently does not support all system calls for the amd64 architecture, and some syscalls have partial implementations [117].

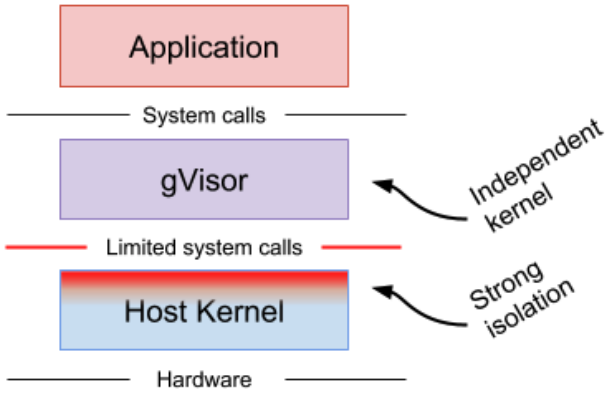

Figure 14: A gVisor sits between the application and the host kernel [115]

- **Container instance auto-scaling and self-healing**: Cloud Run scales container instances by default, depending on the incoming requests. However, developers can specify the number of mins and max instances when configuring auto-scaling. The min-instance and max-instance are both set by default to zero and three, respectively. Users can increase the max-instance number per service basis by editing Quotas. Configuring the max-instance number ensures that the number of container instances must not surpass the upper limit at any given

time during the traffic surge. However, in some cases, Cloud Run may launch slightly more container instances, for a short period, than the given maximum instance value, for example, during rapid traffic surges. Cloud Run will replace a failed container with a new instance if it fails. Cloud Run queues incoming requests for up to ten seconds if there are not enough container instances to fulfill the traffic demand. When no instances become available to handle the queue request during that period, the request fails with an error code of 429 on Cloud Run. By default, a Cloud Run container instance allocates CPU resources only when processing a request, and users are charged per request. Although there are no requests to handle, users can allocate CPU resources for containers until their lifecycle. This would incur costs for the users as long as a container instance exists [118].

Request timeout is set to five minutes by default, but it can be increased to sixty minutes, implying that a response must return within that time frame. The request is completed if a response is not returned within the timeout frame and an HTTP error code of 504 is issued. For example, Cloud Run initiates a new request if a client reconnects due to a client or Cloud Run failures. Cloud Run does not ensure that the client will connect to the same container instance of the service [131].

Cloud Run terminates a container instance if it has not received any traffic for a maximum of 15 minutes. Before termination, the grace period often prevails for up to 15 minutes [118]. This reduces the negative impact of cold starts. A cold start means no operating containers receive inbound requests to invoke functions or API calls. While inbound requests are queued for ten seconds, one or more containers must be created. This adds overhead, primarily latency, which can negatively impact the end-user experience. It is known that large container images can raise security concerns by including more than the code requires. However, the size of the container image has no bearing on the cold start or request processing times, and it does not count against the containers' available memory [119].

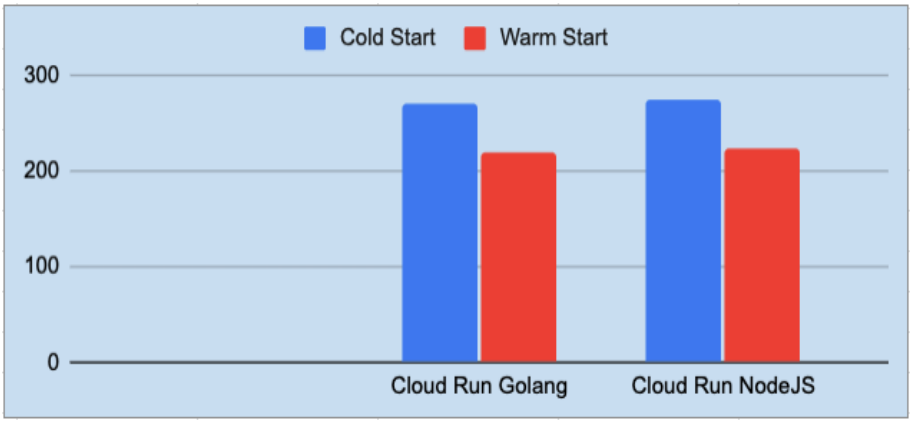

Figure 15: Comparison between Cloud Run's cold start and warm instance

In figure 15, the author demonstrated that container image size has almost no impact on how long it takes to complete a cold start [108]. The cold start duration was almost identical between a Go container image size of 8MB and a NodeJS container image size of 900MB. It is worth noting that the same amount of memory and vCPU resources were allocated for Go and Node.js containers [81]. However, developers can reduce latency by keeping the smallest instances warm. A warm container instance is kept on standby to handle incoming requests, resulting in some consumer expenses, although no requests are handled. A warm start (sometimes refer to hot start) means that it does not recycle prior API calls or function invocations, and it is ready to receive incoming client requests. While cold starts save money, they introduce latency to the request-response process. The following factors, for example, impact the number of container instances scheduled to accommodate a sudden spike in incoming traffic [118]:

  - CPU utilization of existing container instances. If a container instance is 60% utilized, a new one is usually added to the container fleet.
  - Minimum number of container instances
  - Maximum number of container instances
  - Number of concurrency set

- **Pricing models**: There are two different types of pricing policies a customer can enable:

  - **Request based**: Customers pay a per-request fee even though no incoming requests are processed, and the CPU is not allocated.
  - **Instance based**: Customers never pay for a per-request fee; instead, they pay for the entire lifetime of an instance, and the CPU is always allocated.

  This thesis deals with an instance-based pricing model where customers are billed for only the resources they use. Cloud Run charges customers past the free tier usage to the nearest 100 milliseconds only for the allocated CPU and Memory when [120]:

  - The container instance is starting.
  - The container instance is shutting down gracefully, handling the SIGTERM signal. New incoming requests are routed to other container instances when a container instance needs to be shut down, and requests currently being processed are given time to complete. Before being shut down, the container instance receives a SIGTERM signal announcing the start of 10 seconds with a SIGKILL signal.
  - The Container instance processes at least one request or event.

  In addition to that, customers are also charged for the total number of requests as well as for the network egress [120]. However, they are not billed for during

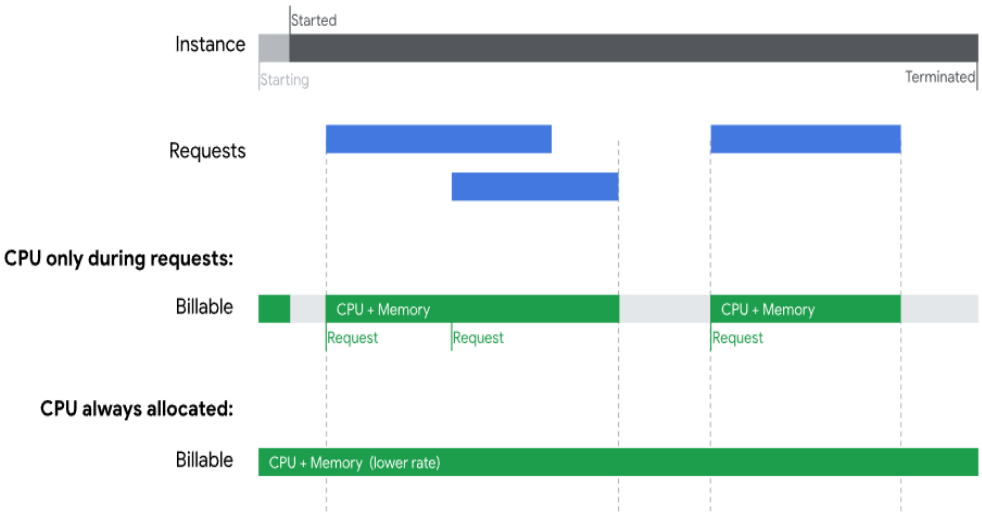

Figure 16: GCP Cloud Run Pricing Model [120]

the container instance initialization and termination phase, as shown in figure 16. They are only charged for the time it takes to complete the request. If CPU resources are always allocated, they are invoiced with a lower rate pricing method for the whole life of each container instance, from when it is started to when it is terminated, with a minimum of one minute. They are not charged if Cloud Run decides to leave the container running when it is not serving requests for some reason.

3.5 AWS Elastic Kubernetes Service with AWS Fargate

The AWS EKS, a serverfull Kubernetes service, helps run, manage, and scale applications in the cloud and on-premises while avoiding a single point of failure (SPOF). It automatically scales up and down applications and runs in a HA architecture across multiple availability zones (AZs), with networking and security integrations. The AWS service provider automatically deploys all security patches to the cluster nodes to safeguard the cluster environment. The etcd persistent layer and the Kubernetes API servers' scalability and availability are guaranteed by AWS for each cluster. A single-tenant Kubernetes control plane is present in each cluster in Amazon EKS. The infrastructure for the control plane is not shared between clusters or AWS accounts [129]. Amazon EKS distributes the Kubernetes control plane across three availability zones (AZs) to offer high availability, and it detects and replaces any control panel nodes that are not functioning properly. At least two API server and three etcd instances are deployed across three Availability Zones on the control plane (AZ) [129]. Amazon EKS offers a Service Level Agreement (SLA) for the API server endpoint availability [122]. Using network policies offered by Amazon VPC (Virtual Private Cloud), an Amazon EKS cluster enforces traffic limits. Unless they are allowed with Kubernetes Role-Based Access Control (RBAC) policies, components of the Amazon EKS control plane do not have the authority to view or receive communication from other clusters

or AWS accounts [129]. A VPC is a virtual network logically isolated from the other VPCs, allowing customers complete control over virtual network configuration, tightening security, and restricting network connectivity to and from the VPC.

Amazon EKS supports both Amazon EC2 and AWS Fargate for running Kubernetes applications. In contrast to self-managed EC2 nodes, AWS Fargate offers serverless computing for containers, enabling customers to run application workloads without managing worker nodes. AWS EKS allows customers to operate OS-dependent applications on the same cluster by supporting Windows nodes. Windows servers run on the node to enable running Windows applications as worker nodes alongside Linux worker nodes. Linux worker nodes run on Linux servers. A container base image runs on a Windows server, so a Windows application can be deployed to a Windows container. This thesis project operates with Linux-based containers, so discussing Windows containers is out of the scope.

AWS Fargate offers on-demand compute capacity for containers running Kubernetes Pods as part of an Amazon EKS cluster. Fargate allows the Kubernetes Pod to run with only the computational capacity users request, and each Pod runs in its isolated VM environment, independent of resource sharing with other Pods. Customers only pay for their Pods, allowing them to boost their app utilization and cost-efficiency with no additional effort. Fargate eliminates server provisioning and management, allowing them to choose and pay for computing resources per application while enhancing security. This removes the underlying nodes' ability to be customized or controlled by the customers. Fargate allows developers to concentrate on designing and building applications. Moreover, GPU workloads are presently not supported on Fargate with AWS EKS. Fargate's SLA guarantees 99.99% of monthly uptime percentage for computing resources [137]. For vCPU and memory, Fargate Pods are invoiced at the standard Fargate rate, and each cluster user's run is billed at the standard AWS EKS rate.

The Amazon EKS on AWS Fargate log router, which streams the container logs to AWS CloudWatch, is built on Fluent Bit. The service AWS CloudWatch is used for storing and monitoring logs [133]. Fluent Bit is a multi-platform open-source project that facilitates log processing and forwarding by collecting data such as metrics and logs from various sources, enriching them with filters, and sending them to multiple destinations. A log router enables users to leverage AWS services for log analytics and storage. Customers do not need to run a sidecar container for Fluent Bit; Amazon does. Customers only need to set up the log router [133].

Fargate allows 20 GB of container storage for each running pod. Pod storage is ephemeral, which means that when a pod terminates, the storage is deleted. Furthermore, pod storage is secured by default by employing Fargate-managed keys with an AES-256 (Advanced Encryption Standard) encryption algorithm to ensure security.

A customer must create and configure at least one Fargate profile in the EKS cluster before scheduling Pods on AWS Fargate. They can specify which Pods should run on Fargate using the Fargate profile. A Fargate profile is immutable, meaning it cannot be altered after creation. The customer must provide a Pod execution role for the Amazon EKS components that execute on the Fargate infrastructure after creating a Fargate

profile. A Pod execution role is added to the RBAC to authorize the Kubernetes cluster, allowing kubelets executing on AWS Fargate to register with the AWS EKS cluster. This permits Fargate infrastructure to appear as AWS EKS cluster nodes. The Identity and Access Management (IAM) permissions associated with the Pod execution role cannot be assumed by the container executing in the Fargate Pod. To grant authorization to the containers in the Fargate Pod to access other AWS services, the IAM roles for service accounts must be used [129].

3.6 GCP GKE AutoPilot

Google handles the cluster management, which includes the nodes, resilience, availability, scaling, security, and other predefined settings, as part of the serverless Kubernetes service offering of GKE AutoPilot. AutoPilot clusters are optimized to run production workloads, and compute resources are provisioned following the instructions in the Kubernetes manifest files. The simplified configuration adheres to GKE's best practices and recommendations for the cluster and workload setup, scalability, and security.

Since GKE must manage the underlying nodes to host the workload Pods, customers are never billed for any unused or underutilized nodes capacity. Furthermore, customers do not pay for system Pods, OS costs, unallocated space, or unscheduled workloads. There is no minimum duration for AutoPilot resources; billing is done in increments of seconds. However, The same flat fee per cluster applies regardless of whether a cluster is single-zone, multi-zone, regional, or Autopilot. If resources are not provided in the Pod specification, GKE AutoPilot optimizes resource usage to lower costs based on the compute instance classes. Each computing instance class contains a set of standard resource requests. GKE AutoPilot includes SLA that provides 99,95% of the control plane's availability and 99,9% for the Pods in multiple zones [146].

To sum up, in both serverless Kubernetes offerings in AWS and GCP, users can either delete or always run a cluster. For example, suppose users must stop the Kubernetes cluster to save money during the quiet period; in that case, that is impossible, unlike a serverfull Kubernetes cluster. The Kubernetes per-hour cluster bill remains valid whether workloads run or not.

AWS EKS and GKE AutoPilot do not offer a console option for connecting to the RDS database from the EKS cluster. Suppose the private RDS database instance is created in a different VPC than the EKS cluster. In that case, a VPC peering is required to communicate with each other.

Before concluding this chapter, we highlight the core feature differences between the services presented in Table 4 based on the discussion from section 3.4, 3.5, 3.6. The fundamental concept of AWS EKS with AWS Fargate and GKE AutoPilot is identical; both services abstract cluster management out of customers' responsibility, allowing them to focus on developing business logic. Both service providers automatically scale nodes to accommodate the cluster's number of Pods. Customers must configure the circumstances under which Pods must be scaled accordingly; vendors are not responsible.

Category	GCP Cloud Run	AWS EKS with Fargate	GCP GKE AutoPilot
Isolation Level	GVisor	Dedicated VM	GVisor
Multi-containers	Can not deploy more than one container at a time	Can deploy multiple containers at a time	Can deploy many containers simultaneously
GPU support	Yes with Anthos	No	Yes
Logging	StackDriver	CloudWatch	CloudWatch
Application Revisions/Versions	Yes	No (but possible with *kubectl rollout* command)	No (but possible with *kubectl rollout* command)
Windows Containers	No	No	Yes
Deployment unit	Container image	Kubernetes Pod	Kubernetes Pod
Deployment Spec	Knative service (Pod)	Kubernetes Pod	Kubernetes Pod
Cluster Pre-Provisioning	Optional with Anthos	Required EKS	No
Scale to zero	Yes	No as they do not support alpha cluster in AWS	Only with alpha cluster Using HPA
Developer Experience	Very happy	Not as happy as GCP Cloud Run	Not as happy as GCP Cloud Run
Learning curve	Very low	Very high and require fairly good knowledge about Kubernetes	Not as high as AWS EKS
Standalone capability to host applications	Available as standalone, because it does not need any Kubernetes cluster	AWS Fargate, is not available as standalone without the support of ECS/EKS.	Standalone
Integration to the cloud eco-system	Integration can be done only to a handful number of GCP services	Offers deep integration with the AWS eco-system	Well integration with GCP eco-system
Release Year	General availability (GA) was released in November 2019 but the beta was launched in April 2019)	AWS EKS added support for running Pods on AWS Fargate on December 3, 2019.	February 2021

Table 4: Comparison among GCP Cloud Run, GKE AutoPilot, and AWS EKS with Fargate

4 Design and Implementation

This chapter introduces the thesis project and all of its associated technologies. Section 4.1 begins with the project introduction. Section 4.2 and 4.3 discuss project architecture and its deployment to different services, respectively. Section 4.4 presents the synthetic test setup for the performance, and cost analysis, whereas section 4.7 briefly presents other technologies employed in the project.

An application has multiple environments in a real-world scenario, including development, staging, testing, and production. The author had only one production environment in this project.

4.1 Project Introduction

A lightweight single-page web application was developed for this thesis project to assess the performance, cost, and developer experiences. The project plan was to develop a few APIs that sufficiently address thesis research questions. Developing a grandiloquent website is not the goal of this research.

The web application project is an online store where clients can browse from any device selection of laptops from the website's home page. Clients can sort laptops by brand. Additionally, they can view the product's information, which includes the name, price, image, and description. They can check out to purchase a product, but they must first log in to the website to place an order.

At the early stage of the project, the author considered decoupling the frontend using a JavaScript framework. However, the idea was dropped due to the time constraints of the thesis and the added complexity. Decoupling the frontend from the backend could offer many benefits but introduces greater complexity. The frontend implementation was done utilizing the templates served by the Django backend. The main motivation was to keep the application simple so that the thesis's primary pursuit could receive greater focus.

The backend database stores the signed-up users' information. Users can remove or add a product to their cart without signing into the website. They only need to log in to check out and place a purchase. They must provide the delivery address when they proceed with an order. Note that the system does not ask for an address when they first register on the website. Providing an address is only required when placing an order. They do not need to pay to place an order because a payment solution system in our project has yet to be fully implemented. The order record is saved to the database when users' orders are placed. They can find the order details in the order tabs after they log in to the website. The order status remains pending until the website administrator marks the status as delivered or completed. When an order is marked as delivered, the client has received their orders. Furthermore, an admin can administer the website by adding or deleting products. Each product is added to a particular category by the brand name. An admin can update information about a product name and its associated information, for example, product description, image if necessary. Moreover, an admin can also alter order information, for instance. This project also implemented a CRUD REST API. An authorized Django admin user can

manipulate products, orders, and categories of the products by making REST API calls using a client.

4.2 Architecture

The project's architecture is simple, as we were in the early phase. This project was solely created for the thesis work, so no real-world use cases or actual users exist. Some dummy users were created to conduct the performance tests as stated in section 4.4. The architecture, however, can be modernized and upgraded by incorporating new features. Although the web application was intended to deploy on serverless computing, it may still be deployed to bare metal. Apart from FaaS, this web application can be deployed on any platform, including Linux or OSX.

The backend of this application uses the Django framework. Due to the author's prior software development experience, the Django framework and PostgreSQL are chosen. Django is a popular Python programming language-based web framework that helps to rapidly build better, secure, clean, pragmatically designed web applications with less code [56]. The Django framework follows a software-designed pattern called

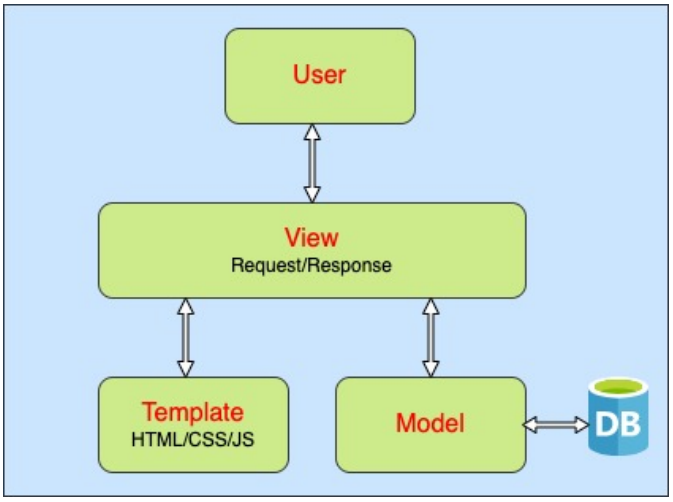

Figure 17: Django MVT pattern

Model-View-Template (MVT). Figure 17 shows that the model acts like an interface of the data, the view is the user interface, and the template contains the static components, for example, HTML output and the special Django template syntax, which explains how dynamic data is incorporated [90].

The backend incorporates a fully managed relational database service storing all stateful data. As this project used two cloud providers, two fully managed SQL services were required to host the database. One vendor-specific SQL service for both providers was not used because it would be difficult to highlight the actual cost and performance variations, consumption rates, and the complexity of setting up, implementing, and deploying the project and service usage.

The vendors patch the minor versions automatically, and conversely, the customers upgrade the database instance size and the major engine version as part of their duty.

When a database cannot handle incoming connection requests because of sudden spikes in traffic for a while, that does not mean that, in such circumstances, cloud providers would automatically adjust the database instance size for customers on the fly to accommodate the connection overhead, as it is not their duty. This means that customers must upgrade to a database instance that is the right size for their use cases. They can always take advantage of enabling the query insights and vendors' recommendations for improving the performance of the database. The database's disk space can expand automatically when data grows over time. A Solid State Disk (SSD) disk was utilized for better performance for the database, which retains data for one day. The cloud vendor oversees and maintains the database's underpinning infrastructure. They must still monitor the instance memory, maximum concurrent connections utilization, and CPU consumption of the database even though the public cloud vendors relieve customers of many administrative tasks off their shoulders. They must configure and tweak the database's parameters to maximize performance efficiency.

Employing a NoSQL serverless document database service, including Google's Firestore or Amazon's Aurora Serverless was opted out as the web application did not have a real-world use case scenario where the user requests spike periodically. A serverless database may be preferable if the website has many visitors. The architecture design could have benefited from incorporating them because they automatically start and stop based on the application's needs on demand. In contrast, customers must continue paying for the computing resource to fully managed services by the minute, even when the database is idle and not processing user queries. In this project, a fully managed cloud database instance was employed. Specifications of the database configuration used in the project are given below:

Features	**Cloud SQL for PostgreSQL**	**Amazon RDS for PostgreSQL**
Database engine version	14.7	14.7
Connection limit	400	849
SSD size	10 GiB	20 GiB
Memory	8 GiB	8 GiB
vCPU	2	2
Network performance (Gbps) up to	2	5

Table 5: Database instance specifications used in GCP and AWS

Provisioning a database instance of the same size and feature offerings is technically impossible. The same feature means the database instance has the same bandwidth, network performance capabilities, and identical vCPU and memory specifications. For instance, AWS offers far more choices for database instance types than IBM and Google Cloud. For this thesis study, the various database instance types and their characteristics—such as maximum network bandwidth, maximum connection limits,

and maximum throughput capabilities—were thoroughly evaluated. The option that best satisfies the requirements of the other Cloud SQL instance type was chosen. The primary factors considered, along with the additional variables specified in Table 5, include the number of vCPUs, memory, and the maximum connection limit. By design, each Cloud Run service is only allowed a maximum of 100 connections to the database instance. The number of database connections grows when the number of instances deployed for each service increases.

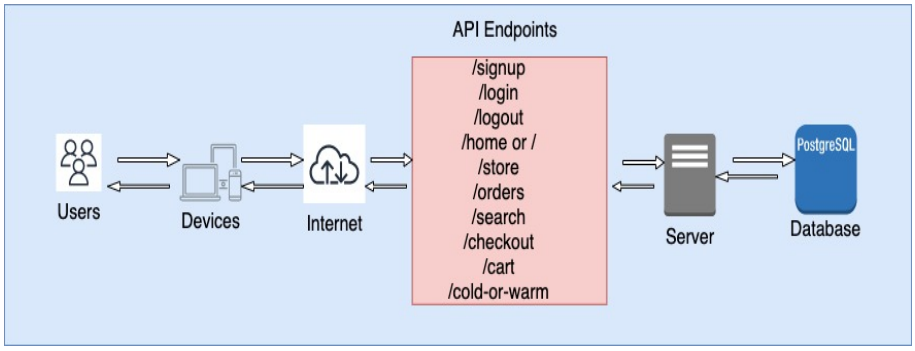

Figure 18: APIs of the project for the non-admin users

Figure 18 shows that users can browse different API endpoints. Besides that, this project implemented a REST API service for products. The REST APIs are the followings:

```
GET /api/v1/products/ - List all products
POST /api/v1/products/ - Create a new product
GET /api/v1/products/<id>/ - Retrieve a specific product
PUT /api/v1/products/<id>/ - Update a specific product
DELETE /api/v1/products/<id>/ - Delete a specific product
```

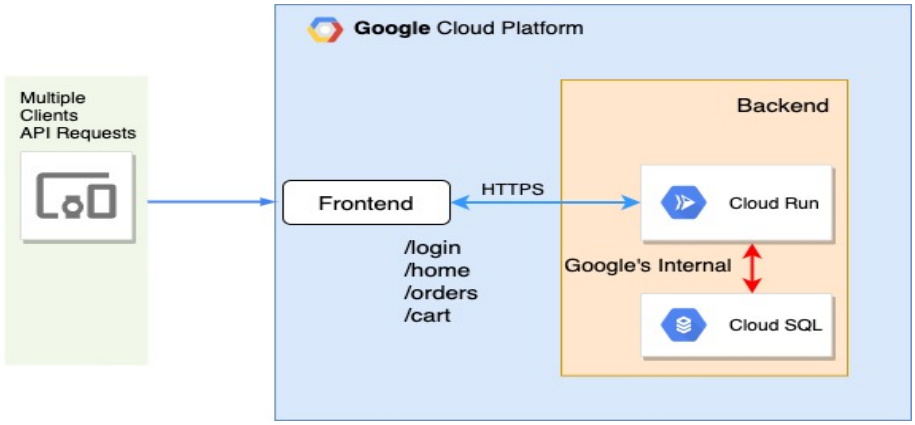

Figure 19: Web application's architecture on GCP Cloud Run

The backend database and serverless Kubernetes run on the same VPC for each architecture presented in figures 20 and 21. Secret manager, backend database in

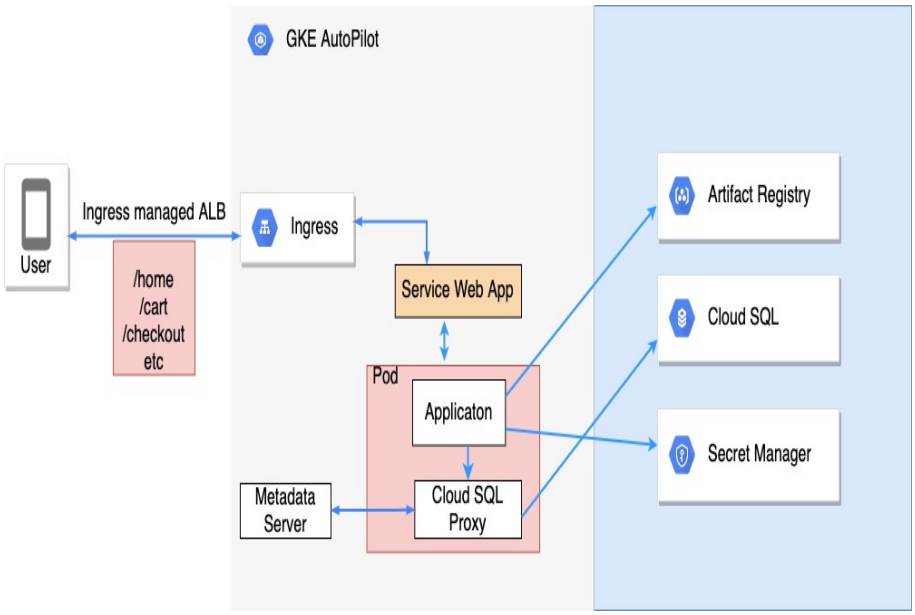

Figure 20: Web application's architecture on GCP GKE AutoPilot

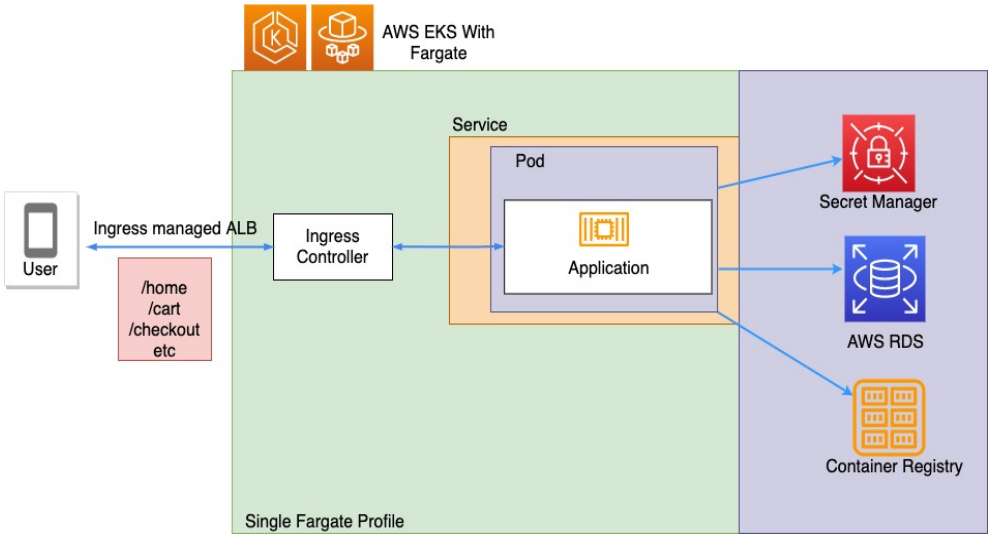

Figure 21: Web application's architecture on AWS EKS With Fargate

figures 20, and 21 are separate services; not part of the Kubernetes cluster. The secret manager shown in Figure 21 in AWS called AWS Secrets Manager. In this thesis, for simplicity, we call it Secret Manager. The ALB, controlled by the Ingress, resides outside the cluster. Fargate has only one profile with two namespaces, *kube-system* and *default*. On the other hand, a single VPC houses Cloud Run and CloudSQL, depicted in figure 19. Figure 20 exhibits that Cloud SQL Proxy ships as a sidecar container within the same Pod where the application container resides, and only authorized network and encrypted communication are allowed to the Cloud SQL. A sidecar container is a secondary container that runs alongside the main application container,

which is also known as the main car, within the same Pod [63]. The securely encrypted and authenticated communication is accomplished by workload identity, which uses a token for each Cloud Proxy's API calls and maps the GCP service account to a Kubernetes service account. GKE AutoPilot enables the Workload Identity by default, but customers must configure the workload identity to use the AutoPilot Pods. The CloudSQL Proxy container cannot listen for any incoming HTTP requests on port 80 of the application container. The application container communicates with Ingress using HTTP port 80. The sidecar and the application container communicate using port 5432.

4.3 Web Application Deployment

When a backend service can connect to the database instance, container registry, secret manager, and other utilized services, deploying the service to GCP Cloud Run, GKE AutoPilot, and AWS EKS with Fargate is straightforward. Providing the necessary permissions to different services is a prerequisite to communicating among them. However, as the single Ingress solution does not work for GKE AutoPilot and AWS EKS, it is essential to fully grasp how Kubernetes Ingress operates to expose the services on the AWS and GCP cloud platforms to the public.

The author leveraged Terraform and Cloud Formation early in the project to orchestrate the service creation and deployment on GCP and AWS due to prior working experience in the industry with those tools. Hashicorp created Terraform, which is a de facto IaC tool in the industry allows customers to provision and manage any cloud resources in human-readable configuration files[16]. On the other hand, AWS Cloud Formation allows customers to provision and manages resources only on AWS [31]. However, in the early stage of the project, AWS EKS cluster creation was automated and version-controlled utilizing Cloud Formation. The version control of the Infrastructure-as-Code (IaC) was not given priority because the project's completion marks the end of the project lifecycle. As a result, the author decided to drop any further platform development throughout the project using those tools. Instead, the author leveraged alternative solutions, such as GCP Cloud Build, Gcloud, and AWS SDK/CLI, to provision and deploy services to GCP Cloud Run, GKE AutoPilot, and AWS EKS with Fargate. Cloud Build is discussed in section 4.7.3. An automated CI pipeline was developed early on since the source code is hosted in *gitlab.com*. A few scheduled pipelines were developed for the thesis project in *gitlab.com* to run the performance tests on AWS and GCP at regular intervals. Section 4.7.2 briefly gave GitLab a little introduction.

4.3.1 GCP Cloud Run

The author runs the following command from the author's local computer to create a Cloud Run service named *webapp-service* in the Frankfurt region. Additionally, if a service name is omitted, a name will be suggested with the default value once the command is executed successfully. In this thesis, gen1 means the first generation and gen2 to the second.

```
gcloud alpha run deploy webapp-service-gen1-cold-start \
--region europe-west3 \
--image $CLOUD_RUN_IMAGE \
--add-cloudsql-instances $PSQL_INSTANCE \
--execution-environment gen1 \
--allow-unauthenticated \
--min-instances 0 \
--max-instances 6 \
--cpu 1 \
--memory 512Mi
```

A Docker image is deployed to create the Cloud Run service on a managed platform in the Frankfurt area. The docker image size of this project was 293MB. The Docker image, retrieved from Google's Artifact Registry, contains the application code, packages, and their dependencies for the web application. The default value of the execution environment is managed. This denotes that the service runs in the fully managed version of Cloud Run, choosing a suitable execution environment for customers. Since the Cloud SQL information is embedded, Cloud Run should establish a connection to the designated Cloud SQL instance. Furthermore, the service must utilize the execution environment of gen1, which has a faster cold start capability. Adding the allow-unauthenticated flag means that users do not need to be authenticated while browsing the website. Important to note that the minimum and maximum number of instances by default is zero and three, respectively, regardless of the execution environments. Customers can set the max instance to ten without raising any support tickets to GCP. The minimum instance size of zero means there will not be any instances to keep warm when there are no incoming user requests for a while. Updating an existing Cloud Run service is made straightforward.

```
gcloud run services update webapp-service \
--region europe-west3 \
--image europe-west3-docker.pkg.dev/ \
$PROJECT_ID/$REPO/$IMAGE_NAME
```

The above command updates an existing Cloud Run service by leveraging the most recent docker image from the GCP Artifact Registry. Figure 22 portrays the service deployment process to Cloud Run. The deployment is automated via a Cloud Build-specific script that builds, pushes, makes migrations (generates the SQL command if changes are made to the database model), and migrates (executes those SQL commands to the database) changes into the database. Furthermore, the script creates (if the service does not exist) or updates the existing Cloud Run service and routes 100% traffic to the newly created application revision.

The author deployed two GCP Cloud Run services of the web application for each execution environment, each connected to the same backend database. Each execution environment (gen1 and gen2) had two services deployed. One for the cold start, and the

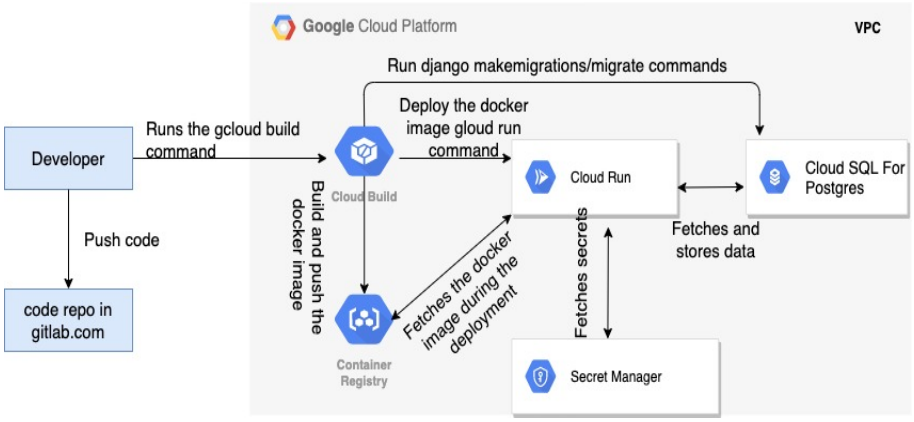

Figure 22: Deployment architecture for the Google Cloud Run

other for the warm start, which means four GCP Cloud Run services were employed for the web application.

4.3.2 AWS EKS with Fargate and GKE AutoPilot

The deployment of updating a new version of the application of the service to GKE AutoPilot and AWS EKS was not automated. The author runs the kubectl command to create/update application running in the Kubernetes cluster. The cluster creation in AWS EKS with AWS Fargate is only automated. Since the scope is expansive compared to time limitations; therefore, the author discarded automating the service deployment via a pipeline in GKE AutoPilot and AWS EKS. The following command deploys and updates the application in the Kubernetes cluster.

```
kubectl create -f deployment.yaml
kubectl apply -f deployment.yaml
```

The *kubectl create* command creates a resource in the Kubernetes cluster from the deployment manifest file. In contrast, the *kubectl apply* command applies a configuration to the resource that was created by the *kubectl create* command when any changes are made to *deployment.yaml* file.

Kubernetes, by default, uses the rolling update strategy to ensure zero application downtime when deploying a new revision to the running Kubernetes cluster. The rolling update strategy enforces that resource Pods are gradually replaced by rolling updates to achieve zero application downtime throughout the update process. As a result, the application is always usable, even when a new application reversion is being deployed. A rolling update is viable when an application cannot afford downtime for seconds to minutes.

GKE AutoPilot and AWS EKS with AWS Fargate had their own GCP CloudSQL and AWS RDS for PostgreSQL database for the web application.

4.4 Test Setup

This section briefly introduces two different performance test types performed on AWS and GCP. The successive chapter presents the performance test results.

The performance test's objective was to assess the web application performance and how well the system—the platform—performs under various circumstances when the load fluctuates over average traffic patterns. The platform's performance assessment is measured based on the container/instance start times in AWS and GCP. On the other hand, the web application's performance is measured against each successful HTTP request response time. All tests were conducted using K6, which is a JavaScript-based load-testing framework. K6 is discussed in section 4.7.1 succinctly. Two different types of load testing were planned and executed to evaluate the serverless platforms' performance and the web application backend created for this thesis. Both load testing mentioned below was run against GCP Cloud Run, GCP GKE AutoPilot, and AWS EKS with AWS Fargate services used in this thesis. The synthetic load tests were conducted in this project with the simulated virtual users (VUs). A simulated virtual user mimics the behaviour of an actual user. Since running the performance load tests are time-consuming and expensive with numerous VUs and for a longer duration [62]. Therefore, the author decided to keep the maximum duration of incremental load tests at ten and seven minutes for the spike load tests to minimize the total project costs. The load testing results are analyzed in the following chapter.

- **Incremental Load Testing** assesses the platform's performance under traffic surges that are more intense than in a typical load test. In proportion to the increase in load, the ramp-up stage lasts longer. Examples of such events are paydays and end-of-work weeks; the website anticipates frequent higher traffic than the average. These tests were carried out to ensure the platform's availability, stability, reliability, and scalability in heavy use. Furthermore, these tests recognize any response time degradation or platform issues when the load exceeds the typical usage.

  The following table best explains the incremental load tests that were run against different HTTP endpoints on GCP Cloud Run.

  Table 6 shows that the total duration for the incremental load tests was executed for ten minutes with 500 VUs, including 30 seconds of the ramp-down period to scale down to zero VUs. The loads were generated gradually to put the platforms under stress at various time intervals with different VU counts. These tests were performed only against GCP Cloud Run.

- **Spike Testing** evaluates how rapidly resources are created to manage abrupt increases and decreases in incoming traffic load for a brief period without inducing higher latency. Furthermore, these tests were conducted to determine if the system can function normally and survive when subjected to sudden, heavy traffic inflows. For instance, an excellent example of such events is Black Friday, ticket sales opening for concerts, and new product launches. Another scenario is when a website expects a spike in traffic after a marketing campaign that lasts

GCP Cloud Run incremental load test case scenario	
Duration in minutes	**Number of VUs**
1	50
2	100
2	300
1	500
1,5	350
1	450
1	100
0,5	0

Table 6: Test case scenario for the incremental load tests

only a short while. In both scenarios, traffic surges dramatically with high loads in a very short or nonexistent ramp-up phase. Real users in such situations often browse a certain page and then leave; they do not hang around doing additional actions [62]. Spike load tests were performed against GCP Cloud Run, AWS GKE AutoPilot, and AWS EKS with Fargate.

GCP Cloud Run test case scenario	
Duration in minutes	**Number of VUs**
1	50
5	500
1	0
AWS EKS/Fargate and GKE AutoPilot test case scenario	
1	10
5	60
1	0

Table 7: Test case scenarios for the spike load tests

Table 7 shows that the total test duration for the spike load testing was seven minutes, including one minute of a graceful ramp-down period. The GCP Cloud Run spike load tests started with fifty VUs to generate one minute load. The following five minutes test runs with 500 VUs while the last one minute is set to ramp down to zero VUs. The spike tests continued running until the allotted test duration times out. On the other hand, the spike load test duration remains identical for the GKE AutoPilot and AWS EKS with Fargate, except for the number of VUs. The spike test started with ten VUs for the first minute and bumped into sixty VUs for the following five minutes, while the last minute was set for ramping down to zero VUs.

Table 7 demonstrates that the spike load tests ran for two different scenarios over

three stages, while Table 6 ran tests over eight stages. A *graceful-stop* was specified to allow K6 to complete ongoing iterations. Technically, if the graceful-stop is not specified when the test contained a total test duration or ramp-down period, K6 might interrupt ongoing iterations. These interruptions could result in skewed metrics and unexpected test results [106]. The author examined the test data after all test runs were finished to determine if any iterations were interrupted during the test run. The author noticed two or three occurrences where a few iterations were interrupted, which is negligible.

Due to the vCPU quota limit—set by default by the vendors at eight vCPU for the Frankfurt region of AWS and GKE—a similar volume of incremental and spike load tests could not be performed in GKE AutoPilot. One fundamental discrepancy between the two is that in GKE AutoPilot, only three Pods can be created with one vCPU for each container, whereas six in AWS Fargate with one vCPU for each container. For instance, GKE AutoPilot runs two containers in one Pod, as demonstrated in figure 20. The reason they differ is due to the way they are designed to work. Regardless, AWS does not depend on sidecar containers to communicate to the backend database, which is an advantage. Thus, AWS allows customers to leverage more Pods while staying under the quota limit compared to GKE AutoPilot. To prevent surpassing the maximum allotted vCPU in both load test scenarios, as shown in Table 7, the number of VUs was modified by setting the maximum permissible replication to two. More VUs result in increased loads on the web application; therefore, more replicas must be created to handle the incoming requests. Suppose more than three pods are created in GKE AutoPilot during the load tests. In that case, the application Pods are left pending since there is not a vCPU available to schedule them to the worker nodes, which directly affects the request response time. This is why the number of VUs had to be reduced to sixty for spike load tests in GKE AutoPilot to stay under the quota usage. This led to the abandonment of the incremental load testing against the AWS EKS with AWS Fargate and GCP GKE AutoPilot.

The internet connection utilized to run the performance tests from the author's workstation had Mbps download and upload speeds of 250 and 50, respectively. If the default MTU size is larger (max 9000 bits), each packet can transmit more data over the network. The Maximum Transmission Unit (MTU) size, which is 1500 bits by default, is not altered. All performance tests were executed from Helsinki, Finland, and the author's workstation. Figure 23 shows the workstation's specifications.

k6 is heavily multi-threaded and uses much memory to generate large loads. Suppose extensive load tests hit the maximum resource utilization for CPU and memory. In that instance, the tests might encounter certain throttling restrictions, resulting in a higher response time for the result metrics. However, we did not have any extensive performance test scenarios in our case, as we had only 500 VUs. For instance, a load test with 1000 VUs might use one to five GB of memory [147]. The author did not notice any hardware resource constraints during the performance tests as the resource utilization stayed below 60%, which means that resources were never a bottleneck to generate loads throughout the performance tests from the author's workstation.

Figure 23: Workstation's specificatin

4.5 Platform Configuration and Performance

This section highlights the platform's configuration for the performance tests in AWS and GCP. The following chapter presents and analyses the test results.

Container startup times measure the performance of the platform. The initial hypothesis is that GCP Cloud Run's container instance startup time would be much lower than its counterparts GKE AutoPilot and AWS EKS with Fargate [81]. The first time when Kubernetes pulls an image, it can take several minutes to start a container when the image size exceeds a few hundred MBs to GBs over time. Furthermore, vendors acknowledged this an issue still [82]. The main contributory factor to the slowness is that the container runtime must fetch the container image from a container registry before a container can start. Although Kubernetes uses image caching on the underlying nodes to solve this issue, this reduces the container startup time for the subsequent containers but remains a challenge for the first container image pulls [82].

GCP Cloud Run creates a new container instance depending on the number of concurrent incoming HTTP requests. As discussed in chapter 3 concurrency limit for incoming requests may be impacted if the instance CPU utilization is 60%. But users can only configure the concurrency but not the CPU utilization, as this is invisible to them. On the other hand, in serverless Kubernetes, figure 24 demonstrates that HPA was configured based on the CPU utilization metrics. The target CPU utilization metrics were set to 50, 60, and 70 during the performance tests, and the data is shown in the following chapter. The minimum and maximum instances were assigned to one and six for the application container for GCP Cloud Run, whereas GKE AutoPilot and AWS EKS had two max instances. This means that one instance must always be warm to handle the influx of requests over time, and a maximum of two and six containers can be launched to handle the incoming requests for GKE AutoPilot, AWS EKS with Fargate, and Cloud Run, respectively.

The load tests were carried out in GCP Cloud Run with a minimum instance size of zero and one; the results are shown in the successive chapter. This project utilizes the stable version of the GKE cluster, which does not support the alpha APIs. Regardless,

the GKE cluster does not support *HPAScaleToZero* feature, which is enabled only in the alpha cluster. This is why setting the minimum instance to zero was not possible based on the metric utilization. However, under any circumstances, more than six containers would not be created, which is a hard limit in Kubernetes. Conversely, this rule does not apply to GCP Cloud Run while performing load tests. Typically, load tests exceed the max instance size boundaries depending on the sustained load patterns [118].

Figure 24: A HPA manifest file used in GKE AutoPilot and AWS EKS

Instance Specification	AWS EKS/Fargate	GCP Cloud Run	GKE AutoPilot
vCPU limit	1	1	1
Memory limit	512 Mi	512 Mi	512 Mi

Table 8: Container instance specifications used in different services

All services presented in Table 8 reside in the Frankfurt region, as mentioned earlier in chapter 3. Furthermore, it displays the container instance specifications utilized in the thesis. The primary motivation for using identical and smaller instance sizes is to minimize the performance and cost gap. However, if the services are located

in different parts of the world, network latency may significantly impact how quickly a web application responds to HTTP requests. This is why they are geographically co-located so that compared results in the following chapter can be more accurate. The same container configuration is used for all spike and incremental load testing.

4.6 Web Application Performance

The section introduces the HTTP endpoints of the web application selected to perform the performance tests against the deployed services in GCP and AWS to measure the HTTP requests response time. The web application incorporated a PostgreSQL backend database presented in Table 5. In addition, Table 8 presents the application container specifications.

The performance tests were divided into two groups. These groups are categorized based on the database queries. The first group of tests (let us call it *group1*) involves the endpoints that query the database to fetch the product's information to the visitors. These tests only make the HTTP GET requests, implying no data is manipulated in the backend database. The second group of tests (let us call it *group2*) was run against the endpoints, which never involved any database calls or queries; for example, browsing the login and signup pages. In these tests, simulated users do not log in or sign up to the pages. They only visit the pages. The following table shows the HTTP endpoints selected for the performance tests of the web application created for this thesis.

group1	**group2**
/home /store	/login /signup
Make calls to the backend database of the web application to fetch products information	No calls to the backend database

Table 9: Spike and Incremental load testing endpoints

Performance testing of the rest of the HTTP endpoints in our web application for the research study was not viable due to time constraints and project costs. However, to achieve the research goal, load testing all of the HTTP endpoints is optional because the objective can be achieved by testing a few endpoints.

4.7 Adapted Relevant Services

The services and technology employed to implement the project are described in this section. There are two sets of services: (i) cloud-specific services specialized to clouds and (ii) non-cloud-specific services. Cloud-specific services refer to AWS and GCP services. AWS and GCP-exclusive services, on the other hand, refer to non-cloud-specific services. K6 and GitLab fall into the non-cloud-specific services, while the rest in this section are cloud-specific services. The cloud-specific services

from AWS and GCP may overlap because similar services serve identical purposes to the end users but with different price rates, capabilities, and SLAs.

4.7.1 K6

K6 is a free, open-source, developer-centric, extendable, high-performant modern load testing tool written in Golang and JavaScript that makes performance testing easy and productive for developers and engineering teams. Grafana Lab created K6 in 2017 [144]. It has a built-in CLI, which is backed with developer-friendly APIs. It helps in testing chaos and reliability, load testing, performance, and synthetic system monitoring, as well as detecting performance regressions and faults early in the SDLC. K6 offers a GUI version in their managed cloud with different pricing models. In the K6 cloud, users can only run up to 50 cloud tests for free [143]. As of today, users can write test cases only in JavaScript. This tool integrates well with standard CI tools and can output test results to various backends and formats, including Kafka, DataDog, NewRelic, JSON, and more. This tool's architecture has some trade-offs as it does not run natively on a modern web browser. Furthermore, it does not run on Node.js [145].

4.7.2 GitLab

This thesis used GitLab to version control source code. GitLab is a collaborative Software development platform for DevSecOps projects and includes an intuitive web interface on top of Git. Git is a distributed and popular version control system [13]. Gitlab offers three different types of subscriptions to its users: (i) free, (ii) premium, and (iii) ultimate [141]. We chose the free tier for this thesis because there are no additional costs. GitLab free tier offers shared runners to run pipeline jobs on their managed infrastructures. However, the free tier only allows us to run 400 CI minutes monthly [141, 142]. This is why we set up a local workstation to function as a private runner so that all the workloads could be executed. The pipeline is triggered automatically when a new commit is pushed into a Merge Request (MR) and main branches. Furthermore, when pipeline jobs pass successfully, a new Docker image of the Django application is published to the remote container registry in AWS and GCP. The main is the default branch for our project in GitLab.

4.7.3 GCP Cloud Build

GCP Cloud build is a managed service that executes builds on the GCP's compute capacity powered by the GCP Compute Engine service. Customers are billed per build minute. The price of each build minute is decided by the size of the machine type utilized for the build, such as vCPU, RAM, and so on. While writing this thesis, only the first generation of GA service offerings for the Cloud Build was available, whereas the second generation was in Preview. Second-generation offer better connection service offerings between the third-party source code providers and Cloud Build [148].

4.7.4 AWS/GCP Secret Manager

The secret manager is a centralized, secure, highly available storage system for storing and managing secrets, including API keys, passwords, other confidential data. The secret manager works as a single source of truth to manage, access, and audit secrets across GCP and AWS Cloud [150]. Secret data is stored based on the user-selected regions, and cloud providers handle the data replication automatically. Importantly, secret data is immutable, and the only way to update it is to generate a new secret value, which is also version controlled. Confidential data can be purged depending on the configuration. Confidential data is encrypted in transit and rest with TLS and REST with AES-256-bit encryption keys by default [150]. The IAM roles allow users to grant the least privileges and access rights to confidential data. However, an audit log is created for each request made to the secrets. Secrets are rotated by configuring a rotation policy; in contrast, the expiration policy governs the secrets' life cycle [150, 151].

4.7.5 AWS/GCP PostgreSQL

Each cloud vendor names the RDS for PostgreSQL differently. For example, in AWS, they call it AWS RDS for PostgreSQL; however, GCP quotes it as GCP CloudSQL for PostgreSQL. Both are fully managed relational database service that helps create, manage, maintain, and administer PostgreSQL databases on their cloud platform. Each vendor hosts, organizes, controls, and administers the underlying VM for each instance. Each VM runs the database applications and service agents for logging and monitoring. Customers can configure and manage database instances, including the database instance's size, zones, redundancy, data backup frequency, instance updates time window to deploy security or minor patches, replication configuration, user and database administration, and much more. The database is stored on a persistent disk, a scalable and durable network storage device connecting to the VM. The pay-as-you-go pricing model is determined by the user-defined configuration, which includes storage, CPU, memory, the number of IPs assigned to the instance, and the quantity of traffic leaving the database instance [65]. Their functionality is similar to that provided by a locally hosted PostgreSQL instance. With a few exceptions, the instance deprives the customer of superuser privileges in general. Only three RDS engine types with various instance size capacities are now available on GCP, compared to seven engine types on AWS. One of the key differences is that GCP's current offerings start with disk space as low as 10 GB, whereas AWS's minimum disk space requirement is 20 GB [65, 111, 112].

4.7.6 AWS/GCP Container Registry

A container registry is a secure, fast, scalable, and highly available serverless managed service that supports private and publicly accessible repositories with fine-grained access control. GCP Container Registry was used in the early stages of the project implementation to store and manage Docker images for GCP Cloud Run and GKE AutoPilot. Since GCP Container Registry is deprecated in favor of the GCP Artifact

Registry, the images were migrated to Artifact Registry. According to GCP, Artifact Registry is the next generation of container registries on the GCP platform. AWS Container Registry and GCP Artifact Registry offer vulnerability assessments for known vulnerabilities and exposures after images are published to the registry. One of the most prominent distinctions is that GCP provides the capability to deploy an image to GCP GKE, Cloud Run, and Compute Engine in just a matter of a single click from the service itself, in contrast to AWS Container Registry [152, 156].

5 Evaluation

This chapter provides an insightful analysis of the web application and the platform's performance, evaluates the cost variations, and compares the developer experiences. Section 5.1.1 demonstrate the platform performance, whereas section 5.1.4 assesses the web application's performance. Section 5.2 calculates and examines the cost differences. Finally, section 5.3 evaluates the developer experiences, which include the application development and deployment experiences.

The author wants to clarify that this study was not conducted to determine which serverless CaaS and PaaS services deliver the best performance and cheapest-to-run. This study's sole intent was to explore how these services' performances and costs differ to decide which platform to use if the performance and the costs become the key factors in choosing a platform.

5.1 Performance Analysis

A web application's QoS is typically measured based on response time, throughput, and service availability [62]. The synthetic tests were conducted through load testing to assess the platform's performance, which evaluates how well the platform and the web application support the anticipated workload by running scripts that generate load to simulate the customer behavior at various load levels. The study bases its assessment of the web application's QoS based on the availability and the response time to requests. The platform performance is measured based on container startup latency. This chapter utilizes observational and metric data to identify the performance bottleneck and explains why performance may have declined.

Two different serverless CaaS and one PaaS service were put under performance tests from AWS and GCP: GKE AutoPilot, AWS EKS with Fargate, and GCP Cloud Run. The former two CaaS services operate pretty differently from the latter one. The GCP Cloud Run represents the PaaS service model, while the remainder is CaaS. This section compares and contrasts the performance of the chosen said services. Note that all load tests were run sporadically at different times during May and June 2023.

The spike and incremental load test results presented in this chapter are randomly chosen from many test runs. The intention was to avoid picking up the best results from the test results. For example, spike/incremental load tests were performed ten times for one or more HTTP endpoints of the web application, of which one or two test results were portrayed in the subsequent tables and graphs. The primary reason is that results varied for the same HTTP endpoint from one test run to another. Technically, the results could have been more consistent, as observed. In theory, the author anticipated that the results would be closed in most cases, but that was not in practice.

5.1.1 GCP Cloud Run Platform Performance

This section introduces and evaluates the platform performances of GCP Cloud Run used in this study. The monitoring logs and observations are the basis for the startup container latency statistics shown for all figures in this chapter. These numbers showed how long a cold start container took to handle the initial requests in AWS and GCP.

Figure 25 shows the container startup latency differences between the first and second generation execution environments in GCP Cloud Run while performing the performance tests. Cloud Run fires up new instances on demand based on the needs; their startup time has direct impact in the request latency and throughput of the service. The startup latency is noticeable in web applications where a sporadically large traffic volume is anticipated. Since the incoming request must wait for a new computing capacity to be available before the piles of requests can be started to process.

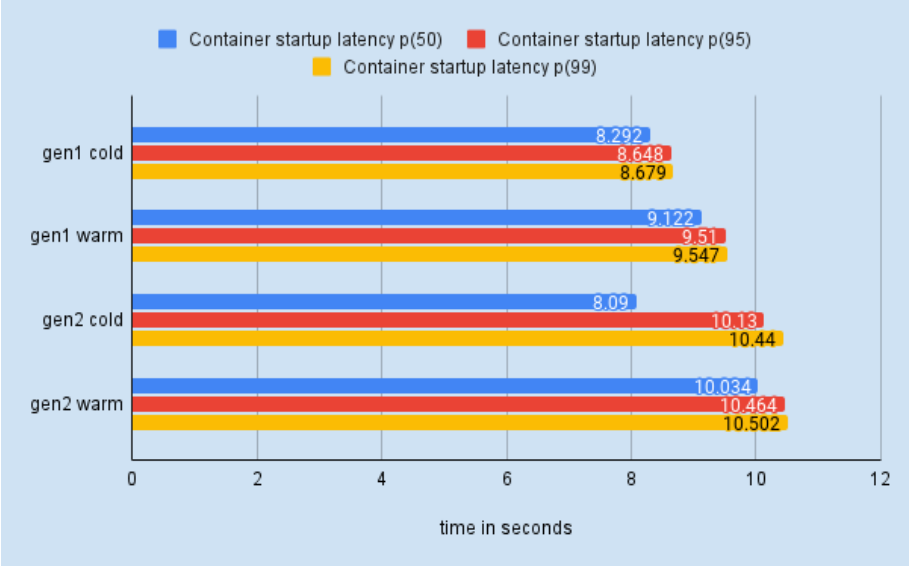

Figure 25: Container startup latency in GCP Cloud Run

Figure 25 demonstrates that the cold start time for the second-generation execution environment was slightly longer than for the first-generation. In the above figure, gen1 and gen2 warm means the web application was deployed on those keeping one instance warm. However, the data presented for the container instances were created on-demand to highlight how fast they started a container from the ground up.

Since Cloud Run only holds onto incoming user requests for a maximum of ten seconds, as a consequence, requests are dropped if new instances are not launched within that time frame. This means that the higher the container startup latency, the fewer requests a service can process at any given time. The above figure shows that the overall instance startup time for the first-generation execution environment stays below ten seconds, which means that the probability of getting requests dropped in 99% of cases is none, particularly for gen1 cold than gen1 warm. Although gen1 warm remain under ten seconds, there is still a small probability of getting some requests dropped from the queue. Additionally, the second-generation cold start instance's

container startup latency was below nine seconds in the 50 percent percentile. This entails that none of the requests will be aborted since they will be processed before the requests queue times out. However, the container starting latency in the 95 and 99 percent percentiles for gen2 cold and warm were more than the ten-second request queue duration, resulting in many incoming requests perhaps being discarded. In contrast, there is a substantial likelihood that a few requests will be dropped even at the 50 percent percentile when using gen2 warm. The per-second request throughput is directly impacted by the container starting latency and is also correlated with it. The proof of the correlation is noticed in the following section by examining the total HTTP requests metrics.

Based on the GCP Cloud Run metrics data, the author observed that in 99% cases, each container had around 85 concurrent requests served while running load tests, although the max concurrent request was at 80. The CPU utilization was at 92% during the peak hour of the load tests.

One possible performance improvement could be utilizing the *startup CPU boost* feature, which helps minimize the startup latency by temporarily allocating a bit more CPU to a container instance during startup. Whatsoever, the author did not utilize the CPU boost feature during the instance startup in this project.

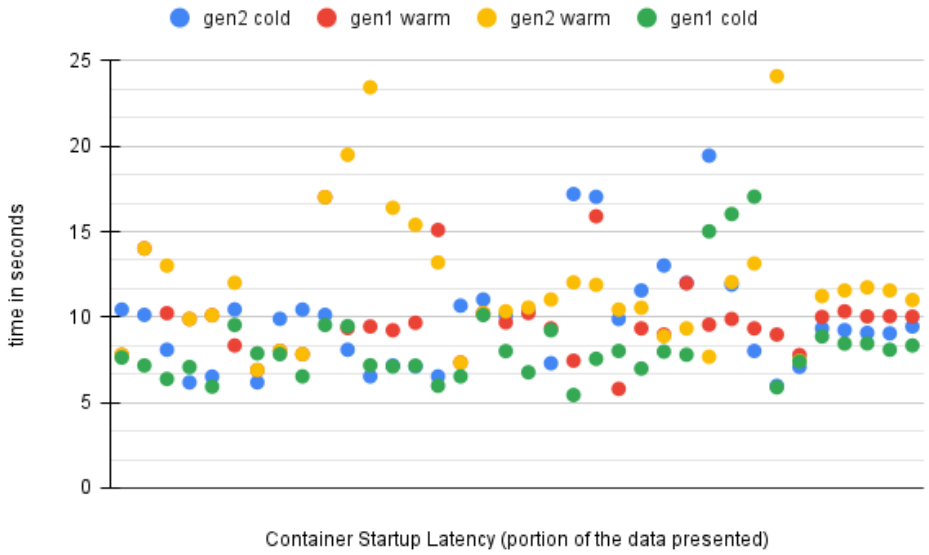

Figure 26: GCP Cloud Run container startup latency

GCP Cloud Run does not provide a readiness probe as of now. Technically, instantly the requests are sent to the newly spawned container instances. A readiness probe periodically checks if the container is ready to serve traffic when adding a new container. Instead, GCP Cloud Run offers a startup probe that users can configure, acting like a readiness probe. The author did not configure any HTTP startup probes for this project mainly for two factors. First, adding a health check would have increased the cold start time by a few seconds, which might have directly impacted the request

throughput. Second, to eliminate the high latency that a small percentage of visitors experience because of the cold start.

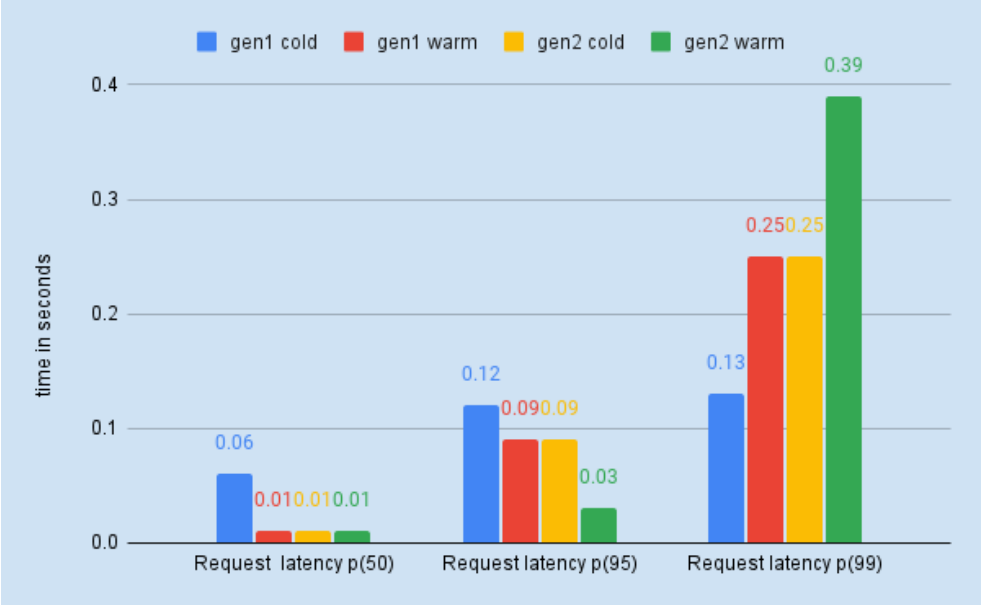

Figure 27: GCP Cloud Run request latency

Figure 27 illustrates the request latency reaching the Cloud Run services in seconds. Latency is calculated from when a request arrives at a running container until it exits. Nevertheless, container startup delay is not included. The figure displays that in 99% cases, gen2 warm had a more significant latency than the rest. In addition, gen1 cold suffered the least, except in the 50% and 95% percentiles, where it experienced the most latency than the rest. Noticeably gen1 warm and gen2 cold had similar request latency in 95% and 99% cases.

5.1.2 GCP GKE AutoPilot and AWS EKS with Fargate Platform Performance

This section analyses the platform performance differences between the GKE AutoPilot and AWS EKS with AWS Fargate.

Figures 28, 29 show the time to cold start a container during the load tests in AWS and GCP. The most interesting data is noticeable in figure 28 for GKE AutoPilot container startup latency. The main reason behind this is the sidecar container for the database proxy. Based on the metric data and observation, the application container (main car) was created in under 18 seconds but, on average, 14 seconds in GKE AutoPilot. However, the application container needs to connect to the database. Therefore, as long as the sidecar container cannot communicate with the database instance and application container, the application container can not function, indicating the request can not be severed.

The health check on AWS was set to the default value of fifteen seconds. In GCP, however, it was set to five seconds. The author decided to minimize GCP's health check interval from fifteen to five seconds because it typically takes a while compared to AWS to get a container running before it can receive user requests. This allows it to accept user requests at least ten seconds earlier than its counterpart in AWS. Once a new application container is created, the ALB performs periodic checks on the health check endpoint. The ALB classifies the container as healthy and begins delivering the incoming user requests to the following application container if the health check endpoints return an HTTP success status code in the range of 200-299 for AWS and 200-399 for GCP. The test scenarios were presented in section 4.4.

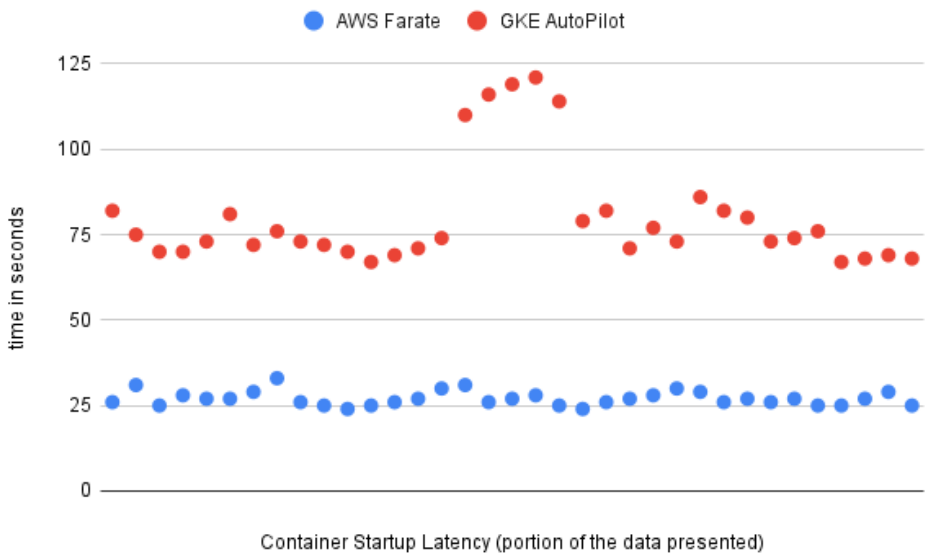

Figure 28: GCP GKE AutoPilot and AWS Fargate container startup latency.

The drift in container startup latency between them is due to the sidecar container, contributing to the high cold start latency. The author examined the API call logs and found that the GKE AutoPilot occasionally took over two minutes before serving the

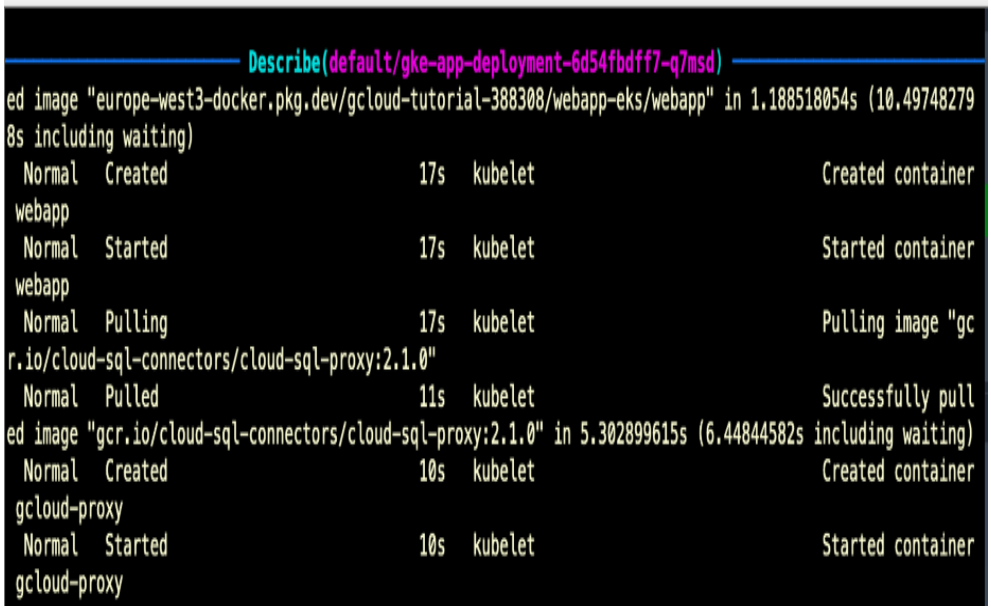

Figure 29: A screenshot is added from the k9s CLI tool for GKE AutoPilot. K9s is an open-source project that lets users interact with the Kubernetes cluster.

first request. One of the possible reasons behind this is that GKE AutoPilot needed to create new worker nodes to host the newly created application Pods, which contributed directly to a higher cold start. Furthermore, the author examined the first request rarely served in under 70 seconds. The negative impact of this was also observed in the first warm container (which was running always) as it had over 93% CPU load for over three minutes when the load started growing gradually. Due to this behavior, the GKE AutoPilot processed comparatively fewer requests than its counterparts and experienced relatively a higher latency. These results are also visible in the application performance tests comparison sections.

Figure 29 demonstrates and breaks down the time a GKE AutoPilot cold start container consumes before a container can be associated with the ALB. The health check time is excluded from the screenshot.

Figure 30 demonstrates that the GKE AutoPilot has a more significant container startup latency than the AWS Fargate. Additionally, GKE AutoPilot's distributions are skewed towards higher values due to some positive skewness and higher variability. AWS Fargate, on the other hand, exhibits a more condensed dispersion and is essentially nearly constant. The startup container latency data presented in Figure 27 and 28 is based on the monitoring logs and observations. These figures demonstrated how long a cold start container took to process the first requests.

5.1.3 Database Performance

This section explains the database overhead encountered during the load tests executed on AWS and GCP services. Although the author did not intend to present the database platform's performance, it is good to mention that the database became a bottleneck,

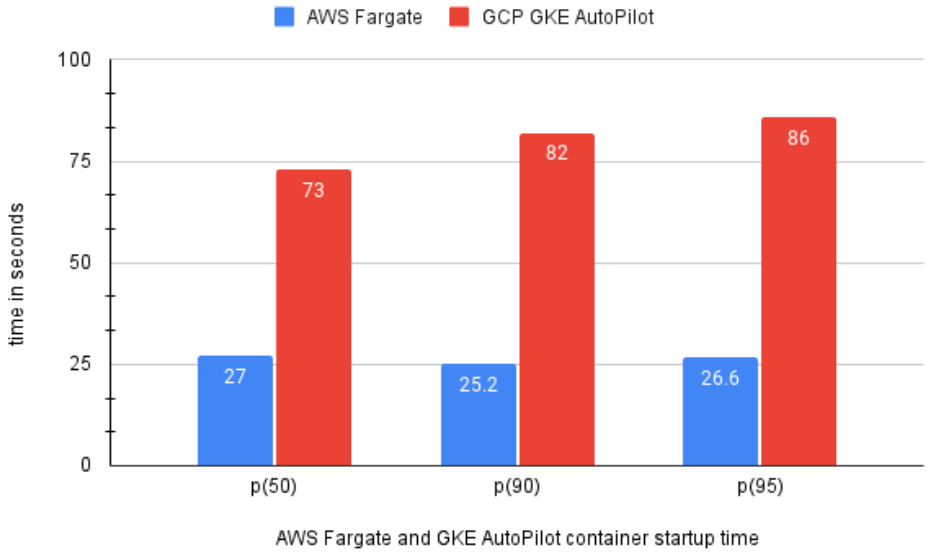

Figure 30: AWS EKS with AWS Fargate and GKE AutoPilot container startup time at different percentiles.

as this can impact the overall request-response time during the load tests. Therefore, this section briefly discusses the overhead of database performance.

Figure 31 shows that the author sporadically performed only one test run to just one instance of Cloud Run to avoid overheads to the database. The database CPU utilization remained below 4% when no load tests were conducted.

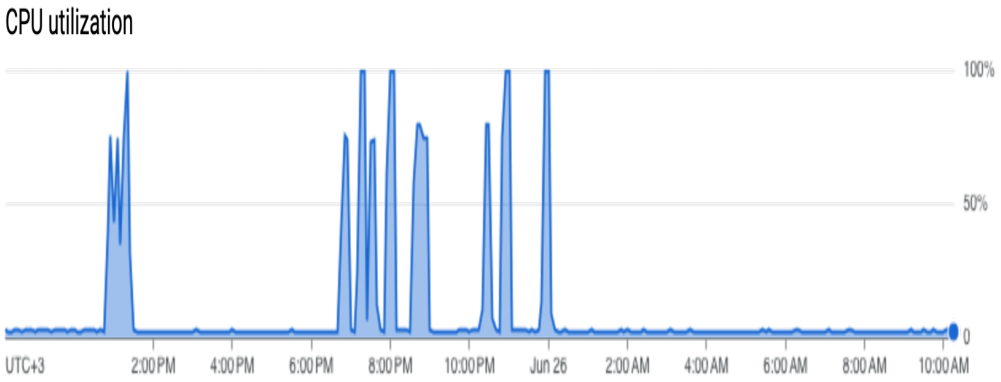

Figure 31: Database CPU utilization while running load tests on GCP Cloud Run at various time intervals

Figure 31 demonstrates that the backend database hit the maximum CPU utilization limit for several minutes while performing the incremental and spike load tests on GCP Cloud Run at various time intervals. During the spike and incremental load tests, 500 VUs made approximately 300 calls per second to the application. The CPU resource utilization remained over 99% for nearly four minutes during the spike tests, negatively impacting the overall request-response time. The incremental load tests generate loads

with 500 VUs only for a single minute, whereas the spike tests maintain an equal number of VUs for five minutes. Due to this reasons, the CPU overhead is much more prominent in spike tests. The author observed the similar behavior while running the spike tests in AWS EKS with AWS Fargate with 500 VUs. The CPU utilization was over 99% in AWS RDS for the PostgreSQL instance. To conclude, the databases experienced a similar overhead in GCP Cloud SQL and AWS RDS while performing the load tests, which contributed to the requests' latency. The database metrics were examined for AWS and GCP, but no connection errors were found. Furthermore, test results did not report any failed or timed-out HTTP requests. On the other hand, the backend database for the application hosted in GKE AutoPilot and AWS EKS with AWS Fargate did not encounter a significant resource overhead when tested against them with sixty VUs.

5.1.4 Web Application Performance

This section evaluates the web application's performance on GCP Cloud Run, GKE AutoPilot, and AWS EKS with AWS Fargate. First, the GCP Cloud Run's performance results are compared between gen1 and gen2 execution environments. Second, the load test results of the GKE AutoPiot and AWS EKS with Fargate's are analyzed and contrasted. Finally, the performance of the GCP Cloud Run is compared to that of the AWS EKS with Fargate.

The average request response time presented in this chapter is the sum of the HTTP request sending, waiting, and receiving times. Table 10 shows the request response time for the spike tests performed against the GCP Cloud Run gen1 and gen2 execution environments. The test results on the left-hand side of each column for all tables in this section were from the group1 HTTP endpoints. Table 10 demonstrates no considerable performance difference between group1 and *group2* HTTP endpoints regarding minimum HTTP request-response times. There is a discernible performance difference between gen1 and gen2 for the maximum request response time. According to GCP [134], one of the primary reasons is that gen2 performs better under sustained loads and offers quicker CPU and network performance than gen1.

Since *group2* HTTP endpoints did not interact with the backend database, their throughput per-second was comparatively more observable. However, gen1 had a slightly faster average response time for group1 HTTP endpoints than gen2. Gen1 warm had the best results in this category. Gen2 (HTTP endpoints for group1) generally had some latency. However, there were no significant latency variations in the average response time for *group2* HTTP endpoints for gen1 and gen2.

The response time for the 90th and 95th percentiles for gen1, including cold and warm, performed better than its counterpart for gen2. On the contrary, *group2* HTTP endpoints had a slightly higher performance than gen2. There is a considerable performance difference between the first and second generations, as seen by the 95th percentile (2600 ms in gen1 warm) being lower than the 95th percentile (5320 ms in gen2 warm). This indicates that the gen1 warm instance's response time distribution is skewed toward faster response times, while the gen2 warm instance suffered longer. Moreover, the remaining 5% of the gen2 warm response time is higher than that figure.

The test results contradicted the author's initial prediction that the warm instances in generations first and second would perform better than the cold start instances.

	gen1 cold		**gen1 warm**		**gen2 cold**		**gen2 warm**	
Total HTTP requests	108888	319750	126792	328982	83510	339646	84950	359686
	109310	331420	123918	343340	81144	354304	86060	330878
	177946	**527562**	**178627**	**492098**	**130652**	**434524**	**126502**	**481764**
Minimum response time (ms)	43.64	46.61	42.94	45.92	42.64	45.23	42.87	45.22
	42.87	46.26	43.12	46.34	43.44	46.19	42.89	45.84
	43.24	**45.42**	**130.03**	**46.53**	**43.06**	**45.90**	**43.43**	**46.12**
Maximum response time (ms)	26400	25600	25700	60000	15170	13230	14380	25910
	25740	25550	25470	25500	13250	24700	51430	25620
	25690	**51150**	**60000**	**25530**	**13270**	**25700**	**25720**	**13040**
Average response time (ms)	911.67	379.49	782.85	391.05	1190	395.85	1170	375.94
	908.13	375.86	801.51	382.85	1220	375.18	1150	385.03
	764.66	**350.90**	**760.76**	**362.10**	**1040**	**377.45**	**1070**	**372.66**
Median response time (ms)	251.44	60.99	229.87	100.14	661.82	141.63	594.65	118.83
	239.89	68.68	227.37	119.43	847.82	94.96	411.35	90.87
	229.62	**119.86**	**216.05**	**99.86**	**288.76**	**55.02**	**651.56**	**121.90**
Req. per second	259	761	301	783	198	808	202	856
	260	789	294	817	193	843	204	787
	296	**879**	**289**	**820**	**217**	**723**	**210**	**802**
90th percentile (time in ms)	2600	1570	2170	1410	3020	1110	3600	1110
	2660	1470	2340	1220	2960	1320	4050	1350
	2270	**1140**	**2260**	**1300**	**4020**	**1590**	**2910**	**1400**
95th percentile (time in ms)	2930	2230	2610	1700	3620	1350	3990	1250
	2950	1970	2600	1500	3320	1530	5320	1710
	2570	**1300**	**2560**	**1600**	**4710**	**2670**	**3380**	**1590**

Table 10: Web application spike and incremental load test results. The data presented in bold represents the incremental load test results, while the rest from the spike test results performed on GCP Cloud Run's first and second-generation environments

The data presented on the right and left-hand side of the columns *gen1* and *gen2* is from *group1* and *group2* test results, respectively.

The incremental load test results for the GCP Cloud Run are shown in bold in Table 10. Ten minutes were allotted for the tests. The average latency was nearly the same, except for the gen1 warm, where the minimum response time for *group1* HTTP endpoint calls was three times greater. On the other hand, gen1's maximum latency was significantly greater than gen2's. In addition, the latency for gen1 warm was one minute, followed by the gen1 cold, which had almost the same latency as gen1 warm. This indicates that gen1 warm had difficulty serving user requests with minimal latency. Gen2 performed significantly better than Gen1 in this area as they had comparatively faster response time. The typical latency for HTTP endpoints in *group2* was nearly the same. Alternatively, the average latency for *group2* HTTP endpoints, gen2, had shorter latency than gen1.

The significant performance difference was noticeable in the 90th and 95th percentiles latency for gen1 and gen2. Gen2 cold had nearly twice the latency for both percentiles than gen1 cold. However, gen1 warm surpassed gen2 warm in terms of performance. Gen2 cold suffered from performance bottlenecks as the response time was notably impacted.

In conclusion, the author inspected that the response times of generations first and second varied at different test runs. The maximum response time can occasionally vary from one test result to another, which is not uncommon. The main takeaway is that while the maximum response time may occasionally differ, the average response time remains nearly the same.

The response time for the *group1* HTTP endpoints of the web application may suffer a bit because the database instance reached the maximum CPU utilization during the test runs for several minutes, as described in section 5.1.3. This would degrade the request processing, thus eventually contributing to a slow response time.

Target percentage of CPU utilization			
	50 percent	**60 percent**	**70 percent**
Average response time (ms)	116.36 127.00 **243.63 206.07**	204.41 186.55 **242.70 228.06**	134.45 137.17 **292.55 315.54**
Minimum response time (ms)	26.39 26.73 **39.29 41.17**	25.39 25.47 **39.27 41.43**	26.87 26.65 **39.53 42.09**
Median response time (ms)	81.30 91.92 **112.37 156.85**	80.41 84.61 **110.35 185.45**	91.10 90.35 **138.75 266.62**
Maximum response time (ms)	1140 1250 **2310 1570**	2300 1990 **2150 1590**	1130 1140 **2100 2250**
Total HTTP requests	160422 148782 **51264 93548**	61084 66892 **51460 86330**	136500 135688 **42676 64996**
Request per second	381 354 **122 222**	145 159 **122 205**	324 323 **101 154**
90th percentile (ms)	260.79 262.26 **706.89 430.45**	593.43 520.42 **687.91 461.24**	279.26 289.87 **758.99 616.11**
95th percentile (ms)	310.69 308.76 **915.76 527.82**	780.62 678.60 **894.74 556.70**	319.80 335.65 **925.48 744.05**

Table 11: Spike test results based on CPU metrics utilization in AWS EKS with Fargate and GCP AutoPilot. The bold ones represent the test results from GKE AutoPilot, and the normal ones are from the AWS EKS with AWS Fargate.

Table 11 demonstrates the test results based on the CPU metrics utilization at various thresholds. The 90th and 95th percentiles show that the latency difference between the two is quite significant. In AWS, it was evident that a relatively small portion of requests experienced higher latency, as indicated by the higher 95th percentile value. The fact that the 90th percentile is lower than the 95th percentile implies that most requests had faster response times, while a smaller segment of requests experienced longer response times. Based on the analysis, the AWS EKS with AWS Fargate performed well for most requests compared to GKE AutoPilot, with a large portion of the requests experiencing response times below or equal to 310.69 ms for *group1* and the *group2* 308 ms. However, a subset of requests experienced longer response times, as indicated by the 95th percentile value of 925.48 ms for *group1* and 744.05 ms for *group2* in GKE AutoPilot. Furthermore, it can be seen that the typical latency for GKE AutoPilot was two folds slower than AWS's. The per-second throughput was substantially higher in AWS than in GCP. Table 11 shows that the overall performance of the AWS EKS with AWS Fargate was significantly better in nearly every category

presented in the table.

Table 11 displays the CPU utilization at various thresholds of fifty, sixty, and seventy percent, indicating that when the Pod workload exceeds the threshold limits during the test runs, a new Pod must be deployed to accommodate the incoming traffic.

The load test results illustrated in the 95th percentile show that at 50% CPU utilization, the latency did not differ much. On the other hand, the 60% CPU utilization had relatively worse performance compared to 50 and 70 in AWS. GKE AutoPilot had nearly the same latency at 50 and 60. Ideally, this is due to the small set of virtual users utilized to run the tests. At least, that was the author's prediction. Therefore, in AWS, the author performed another set of tests to validate the prediction with 500 VUs, as we conducted for GCP Cloud Run. The following table shows the test results run against AWS EKS with AWS Fargate.

Target percentage of CPU utilization in AWS EKS with AWS Fargate			
	50 percent	**60 percent**	**70 percent**
Average response time (ms)	**1080 433.79** **1120 427.32**	**1270 482.01** **1100 413.17**	**2120 526.38** **1890 549.33**
Minimum response time (ms)	24.58 0.069 24.62 26.31	25.17 26.42 25.13 26.30	0.08 26.25 25.03 26.81
Median response time (ms)	91.62 82.76 96.03 80.3	78.84 82.97 104.05 85.79	83.41 82.20 90.36 83.56
Maximum response time (ms)	**15200 4480** **12320 5820**	**19830 5620** **15850 5220**	**22380 4800** **18260 5280**
Total HTTP requests	91454 290234 88350 286262	78354 262994 89954 303710	47264 233756 53004 222486
Request per second	217 691 210 692	186 626 214 723	112 556 125 529
90th percentile (ms)	**2970 1550** **3010 1510**	**5730 1830** **2940 1590**	**8270 1830** **7410 1860**
95th percentile (ms)	**4540 2410** **4880 1920**	**9080 2220** **4320 1930**	**10620 2180** **9570 2370**

Table 12: Spike test results based on CPU metrics utilization in AWS EKS with AWS Fargate. The highlighted results show that the lower the target for CPU utilization is set the lower the latency and higher throughput.

Table 12 shows the data from two spike test runs performed against the AWS EKS with Fargate. The data demonstrated that the author's prediction was entirely correct. The average latency was significantly higher for the 70th percent of CPU utilization

than it was for the 50th percent and 60th percent. Adding together the test scores reveals that the 50th percentile performed better across the board. If the latency for the 90th and 95th percentiles from the two runs are added, it is evident that the 50th had a shorter response time than the rest. The minimum response time of *0.069ms* and *0.08ms* appears to be an anomaly.

	AWS EKS with Fargate	**GCP Cloud Run**
Average response time (ms)	1080 433.79	911.67 379.49
Maximum response time (ms)	15200 4480	26400 25600
Total HTTP requests	91454 290234	108888 319750
Request per second	217 691	259 761
90th percentile (ms)	2970 1550	2600 1570
95th percentile (ms)	4540 2410	2930 2230

Table 13: Spike test results between AWS EKS with Fargate and GCP Cloud Run. The results are shown for AWS when the target CPU utilization was 50 percent. GCP Cloud Run represents the data of gen1 cold.

Table 13 shows that the spike tests were performed with 500 virtual users and were allotted seven minutes. The data shows that the average latency was slightly less in GCP Cloud Run than AWS EKS, while AWS performed better in the category of maximum response time. However, the most noticeable performance difference can be seen in the 90th and 95th percentile latency. Overall, GCP Cloud Run performed better as the 95th percentile was 2.6 and 1.5 seconds for *group1* and *group2*, respectively.

Even though the maximum instance size was set to six for AWS and GCP, Cloud Run created more instances to handle the requests during the intense traffic surge during the spike load tests. The number of instances created in GCP during this specific test run, for which data is presented in the table, in GCP Cloud Run was not examined by the author. Cloud Run's monitoring metrics showed that it usually created more than nine instances but never exceeded twelve. The author would not conclude that GCP Cloud Run topped the performance of AWS EKS with Fargate. If an equal number of instances are not considered a comparison parameter, the GCP Cloud Run PaaS platform for the web application performed relatively better than AWS EKS with the Fargate CaaS platform.

Based on the performance test results, both platforms performed (GCP Cloud Run and AWS EKS with AWS Fargate) quite well, while the GKE AutoPilot was slightly behind. The only drawback in GCP Cloud Run is that users may encounter occasional latency of more than eight seconds for the first user requests. This behavior might be noticeable only for the cold start instance if the web application has few frequent visitors.

5.1.5 Summary

This section spotlights the main findings of the performance analysis and presents our reasonable opinions on the possible bottleneck that was observed in the previous sections.

Based on the performance analysis and findings, it was identified that GCP Cloud outclassed the serverless Kubernetes services utilized in this study. The *group1* HTTP endpoints could perform better if a more powerful database instance were used. At times, particularly during the spike load tests, the database appears to have overhead, as the requests are needed to fetch the data from the backend database. On the other hand, *group2* HTTP endpoints performed better in terms of throughput and request latency due to not having any database calls. Since the data query requires CPU computing power, perhaps, having a database instance of 4 vCPU would have been better.

The author used the imagePullPolicy always flag in the Kubernetes deployment manifest file intentionally so that none of those services can take advantage of image caching on the node. This concept does not apply to the Pod that is already running on a node, for instance, in our case, the warm replica, which always ran. Since each application Pod was deployed to a new AWS Fargate node, AWS could never cache the images on the node. When the Pod died, the node died as well. While running the performance tests, the author manually examined this using the k9s tool. The same behavior was observed for the GKE AutoPilot. The author was well aware of this behavior. Therefore, tests were run with precaution over two months after many hours or days of intervals from one run to another.

One of the main reasons the GKE AutoPilot had poor performance due to consuming much time in getting the application Pod ready. The existing container, which already serves piles of requests, experiences CPU overhead. It processed many of the user requests, eventually leading to poor performance. For example, this poor performance might not have been noticed if more containers were simultaneously during the tests run and the tests had a longer duration. For example, AWS Fargate creates the node faster. While in GKE AutoPilot, it was comparatively slow to create a node to host the application Pod. Furthermore, in figure 30, we noticed that in 95% of cases, the first request was served in 86 seconds, while AWS Fargate did that in under 27 seconds. The time gap of 59 seconds between them makes AWS a winner, which allows it to process more requests, directly contributing to reducing the latency and taking on more requests.

The network latency may play a role to some extent as the tests were performed at different times of the day. Studies showed that cloud vendors may choose CPU resources of the underlying server from a different hardware generation to maximize their resources' use and predictability [64]. This may directly impact the request response time, and latency may vary if the cold start instances serve the request.

Since the GCP Cloud Run had the advantage of firing up more instances, allowing Cloud Run to process more requests while lessening latency. Nevertheless, the cold start instance was significantly faster to sever the initial requests than GKE AutoPilot and AWS Fargate. In contrast, in serverless Kubernetes, it is a hard limit, meaning that

under any circumstances, the maximum number of replicas must not exceed.

The next section evaluate the costs for this thesis's PaaS and CaaS services.

5.2 Cost Analysis

This chapter analyses the cost differences for AWS EKS with Fargate, GCP GKE AutoPilot, and GCP Cloud Run based on resource utilization throughout the performance tests. The cost differences between GCP and AWS are not computed for the other related services used in this research because they are not in the scope of this study. This is why they were excluded from the cost analysis. For example, this study does not assess the cost of network egress bandwidth, database instance cost, and other relevant services utilized in this project. Presenting the actual incurred costs for vCPU and memory runtime for each round of spike and incremental load tests is very cumbersome. The author monitored the first round of incurred costs and multiplied by ten to calculate the ten rounds of total incurred costs for the spike load tests. Suppose ten rounds of costs were monitored for the vCPU and memory runtime for the performance. The prices may fluctuate by a small margin of a few seconds to minutes due to the on-demand scaling behavior. Section 5.2.1 calculates costs for the CaaS services, whereas section 5.2.2 evaluates the PaaS. Furthermore, section 5.2.3 contrast and highlights the cost difference among the services used in this study. The prices are shown in the thesis in USD for the Frankfurt region (*eu-central-1* for AWS, *europe-west3* for GCP).

GCP region *europe-west3* falls into the *Tier 2* pricing model. All regions in *Tier 2* have the same pricing strategy for GCP Cloud Run. *Tier 2* assures substantially higher reliability, is less prone to malfunctions, and has higher service availability than *Tier 1*. This is why the price in the *Tier 2* region is higher than in the *Tier 1* region. AWS EKS with AWS Fargate price varies by region. However, unlike GCP, AWS does not have the concept of a *Tier 1* and *Tier 2* pricing model.

In this thesis, one GB equals 1024^3 bytes, but one gibibytes (GiB) equals 1000^3 bytes. A GiB-second, for instance, is equivalent to running an instance of one GiB for one second or an instance of 512 mebibytes (MiB) for two seconds and 256 MiB for four seconds. The vCPU-second unit operates on the same principles. This project utilized one vCPU and 512 MiB of memory for each instance.

5.2.1 AWS EKS With AWS Fargate and GKE AutoPilot

The cost results presented in this section are only for the spike tests executed on AWS EKS with AWS Fargate and GCP GKE AutoPilot. There are two different types of resources grouped in the calculation: one is the Kubernetes cluster's per-hour flat fees which apply even though the cluster is idle, meaning that the cluster serves no user requests. The second is the runtime resource consumption, for instance, the vCPU, memory, and ephemeral storage. For the entire project life cycle, which lasted for almost two months, the ephemeral storage bill was only $0.3 from GKE AutoPilot. By default, when a Pod is created in GKE AutoPilot, a non-persistant storage is created along with it. The size of the ephemeral storage is one GB containing no data

(*emptyDir*). The ephemeral storage does not apply to AWS Fargate because all Pod runs in AWS Fargate is given 20 GB container storage free of charge. Therefore, the ephemeral storage bill calculation is excluded as this is negligible since the bill is too small. However, the cost calculation focuses on the billable time for vCPU and memory for the performance tests, including the Kubernetes cluster's per-hour flat fee. The cluster's cost calculation is presented based on the billable runtime of the performance tests. For instance, one round of performance tests based on the various vCPU metrics (50, 60, and 70 percent) for the *group1* and *group2* HTTP endpoints took about 139,63 minutes, rounding it to 140 minutes. The results are presented for the ten spike tests performed during the performance tests.

Service Name	Number of Cluster	Per Hour Price	Region (Frankfurt)	Total cost incurred for performance tests for 20 hours	Estimated monthly cost for 720 hours
AWS EKS	1	$0.10	*eu-central-1*	$2	$72
GKE AutoPilot	1	$0.10	*europe-west3*	$2	$72

Table 14: Hourly flat fee of AWS EKS and GKE AutoPilot Kubernetes cluster

Table 14 shows the total Kubernetes cluster's per-hour flat fee incurred for the spike load tests in AWS and GCP. Table 14 demonstrates that regardless of cluster size, a flat fee of $0.10 per hour applies for each cluster in addition to the resources (CPU, memory, and ephemeral storage) application workload consumes during the runtime. If a cluster is created until decommissioned, the per-hour flat fees must be paid even if there is no inbound traffic. The estimated monthly cost is projected based on the per-hour bill for the cluster.

The incurred cost presented in Table 14 for the Kubernetes clusters is shown only for the ten rounds of spike load tests. In each round, two spike load tests were executed, for example, one for the *group1* and the other for the *group2* HTTP endpoints, as stated in Table 9. The tests were conducted at various target CPU percentage metrics utilization, for instance, 50, 60, and 70. For example, when the target CPU utilization was 50 percent, the spike tests were executed against two groups of HTTP endpoints. This means that six tests were performed for two groups of HTTP endpoints in each round due to the target CPU utilization set at various metrics. This translates into sixty spike tests carried out on a single serverless Kubernetes cluster in AWS and GCP. The spike tests were completed for each round (six load tests) under two hours time window.

Since the data set for the costs are relatively small in figure 32, it is hard to understand how much it would cost in AWS and GCP for the runtime if we do not estimate with a higher billable runtime hour. We provide an estimated cost for the

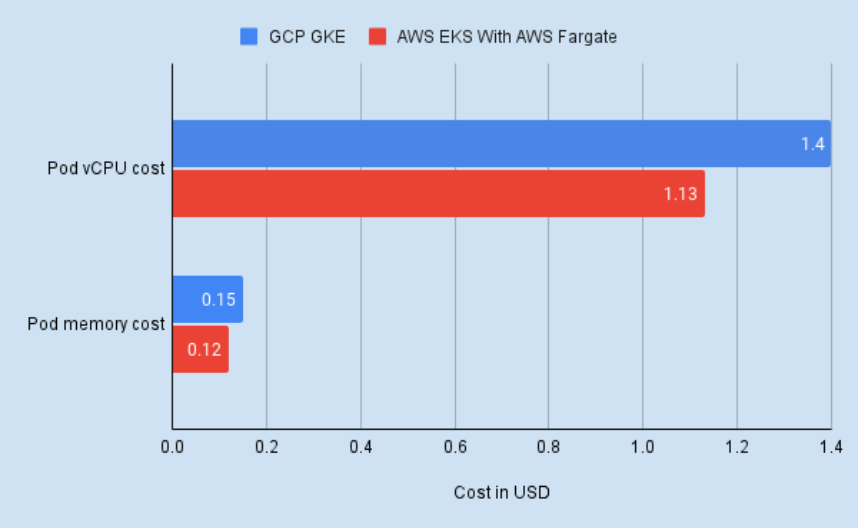

Figure 32: The incurred cost in AWS and GCP for the spike load tests

billable runtime hours of 1080 for the CaaS services in Table 15. The billable hour of 1080 for forecasting is chosen arbitrarily.

Table 15 displays the general-purpose Pods (also known as default Pods) price. The per-second billing price for the Frankfurt region for AWS and GCP is presented. The fees for AWS Fargate are calculated based on the Linux/X86 architecture, as it was utilized in the project. Furthermore, it shows that GCP's per-second vCPU and memory cost is more expensive than the AWS Fargate.

Service Name	vCPU and memory	Kubernetes cluster	ephemeral storage (100 GB)	Total Cost
AWS EKS with AWS Fargate	$55.80	$108	$14.25	$178.05
GKE AutoPilot	$68.73	$108	$7.62	$184.35

Table 15: Estimated cost for AWS and GCP for the billable runtime. The cost is 1080 hours of billable time for cluster, vCPU, memory, and ephemeral storage.

Based on the calculation in figure 15 for the vCPU and memory runtime bill for GCP is 23.13% more expensive than the AWS, whereas the ephemeral storage of GCP is 87.04% cheaper than the AWS. Overall, the total forecasted cost of GCP is 3.42% more costly than AWS. The overall cost for GCP would be much higher if a project does not have much utilization of ephemeral storage. Customers can save up to 23.13% only in vCPU and memory resource consumption. CPU and memory cost factors do not alone decide if the GCP is more expensive in general because there are other factors, including database instance costs, network bandwidth, ALB, and additional costs associated with the relevant services used in the project.

5.2.2 GCP Cloud Run

This section breaks down the incurred costs for the GCP Cloud Run during the performance tests.

Figure 33 demonstrates the total costs incurred for the incremental and spike load tests. Gen1 and Gen2 warm instances are slightly less expensive overall. The main reason for this is that during the spike and incremental load testing, six and nine instances, one of which was a warm instance, were respectively created. Typically, nine instances were concurrently produced for the incremental load tests briefly. Note that the warm instance has a substantially lower per-second cost. As a result, compared to cold start instances, the overall pricing cost variance is slightly lower for gen1 and gen2 warm instances. The data presented in this figure is the collective costs of ten spike and incremental load tests. The figure shows that the total runtime computing costs of CPU and memory were 5.64% more expensive for the cold start instances than the warm instances, while the total expenses were 2.25% more expensive. Suppose the computing consumption is substantively more significant; the price discrepancies between them would be much more noticeable.

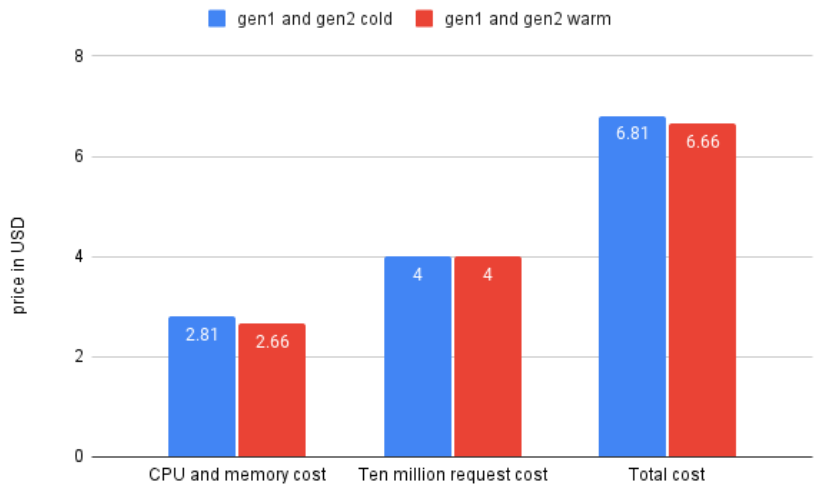

Figure 33: Estimated incurred cost calculation for GCP Cloud Run for generation first and the second.

GCP Cloud Run has distinct pricing policies for cold and warm start instances. The cost of a warm start instance per second is less than that of a cold start. GCP Cloud Run also charges $0.4 for a million requests. The data presented for gen1 and gen2 for the HTTP request is based on the rough calculation from a few test results. The actual number is close to 3.8 million. However, for the convenience of the calculation, it was rounded up to 4 million. There are no fees for requests served by the warm start instance as they do not tally up to the per million request fees. Furthermore, the warm start instance has an advantage over the cold start instance because it costs less per second to allocate vCPU and memory during active usage. Suppose a warm container instance runs for five days and is idle for two days before the service is terminated.

According to the warm instance billing policies, the inactive instance bill is applied for two days, and the remaining three days are charged to the usual per-second invoice.

The idle period cost per second for the warm start instance is significantly less than the active period. The billable CPU and memory are charged simultaneously per second while the warm start instance is inactive. In contrast, a distinct CPU and RAM price rate applies when the instance is active.

There are no per-second price variations between the first and second-generation execution environments. For instance, gen1 and gen2 cold start instances have the same per-second bill. On the other hand, gen1 and gen2 warm have the same per-second billing policies. Customers are not charged for idle container instances for the cold start instance.

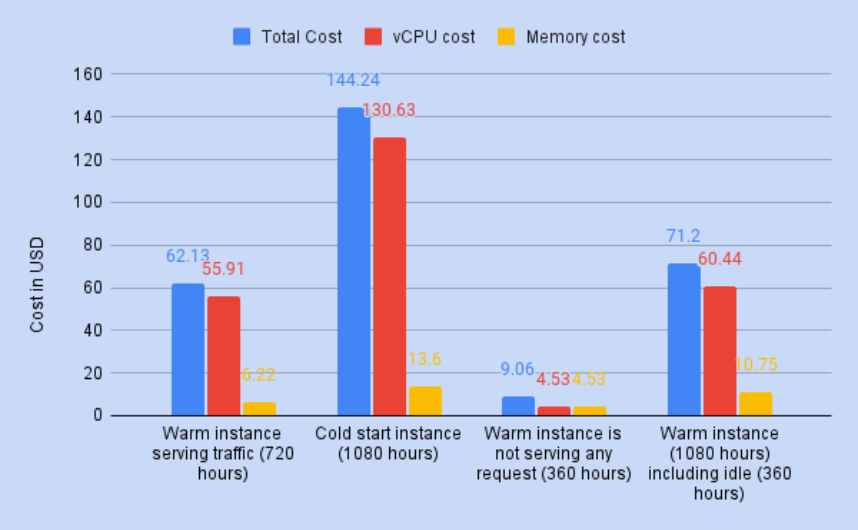

Figure 34: Estimated cost calculation for GCP Cloud Run Cold vs Warm instance

Figure 34 demonstrates an estimated bill of 1080 hours (equivalent to 45 days) for the warm and cold start instances. The cost is calculated based on the per-second billable vCPU and memory utilization. The cold start instance means no instance is kept warm, and the CPU is allocated only during the request processing. On the other hand, the warm instance's CPU is always allocated as the instance runs always. When the warm start instance does not process any request, it remains idle. One of the significant cost differences between the cold start and warm instance is that the warm instance is 55.56% and 45.83% cheaper compared to the cold start instance when the CPU is not idle for vCPU and memory, respectively. One interesting fact is that the cold start instance does not charge customers when the CPU is idle, meaning no request is processed. Conversely, this is quite the opposite of warm instances, as the idle fees are applied when vCPU and memory are idle. For instance, as the above figure shows, a minimal per-second per-GiB bill is charged. In such circumstances, a per second $0.00000350 is charged for the vCPU and memory. The request per million fees is not included in this figure. This is because per million request fees are not applied for the warm instance but for the cold start instances. Suppose the cold

start instance processed twenty million user requests; therefore extra $8 bill would be added as an additional cost on top of billable vCPU and memory costs.

The estimated price for the warm instance will be different if more instances are created, which is the norm when sudden traffic is considerable. Suppose three instances are running while processing the user requests, of which one is the warm instance. In such cases, the price calculation for the cold start instance applies the regular per second bill, and the warm instance would get two different price values: one for the idle and the other when the request is processed. The following figure explains this better.

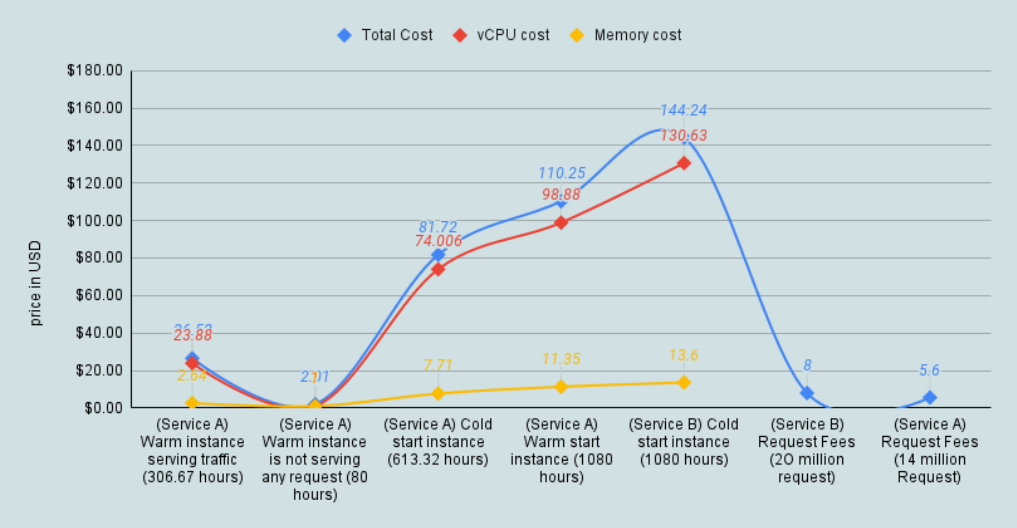

Figure 35: Estimated cost calculation for GCP Cloud Run cold start and warm instance.

Figure 35 illustrates that service A utilizes warm instances. When the traffic surges abruptly, it creates cold-start instances to handle incoming requests. Cold start instances incur two-thirds of the total billable hours, whereas warm instances incur one-third of billing hours. Inversely, service B only utilized the cold start instances to serve the incoming traffic, incurring the most cost for the project, as shown in the figure.

Finally, running a warm instance can be much cheaper than a cold start instance, depending on the use cases. For example, running a service with three instances, including one warm instance, could be relatively less expensive than running a cold start instance.

5.2.3 Summary

The section spotlights the cost variations that are examined in the prior sections. The summary is presented to emphasize the primary outcome of the cost analysis.

In our project, we had one replica running at all times when performing the load tests in AWS EKS with AWS Fargate and GCP GKE AutoPilot. Let us call this replica a warm instance replica, which means it runs always even though no traffic is prevalent

on the website. AWS and GCP do not offer a pricing policy like GCP Cloud Run for the warm instance. The warm and cold start replicas incur the same per-second bill. Due to this pricing distinction, GCP Cloud Run is relatively cheaper than those.

The following table demonstrates the cost differences between the three services of AWS and GCP. The costs are estimated for the 1080 hours of active CPU and memory, excluding the ephemeral storage bill, because we did not use the ephemeral storage in our project. The cold start instance per-second bill is shown in the table for the GCP Cloud Run. The estimated request for the 1080 billable hours is 20 million.

Service Name	Total Cost
GKE AutoPilot	$176,73
AWS EKS with AWS Fargate	$163.80
GCP Cloud Run	$152,24

Table 16: Estimated cost for the billable runtime of GKE AutoPilot, AWS EKS with AWS Fargate, and GCP Cloud Run

The AWS EKS with AWS Fargate and GKE Autopilot would add more bills if the ephemeral storages were used in the project. In the case of GCP Cloud Run, if we had calculated the estimated cost by the warm instances inactive and active billable hours, the price would have dropped, as we noticed in figure 35. The more warm instances are used, the lower the price in GCP Cloud Run. The forecasted per million request bill may lower the cost if the request counts are lower than anticipated for the calculation. Based on the data presented in Table 16, we can conclude that GCP Cloud Run is the cheapest among them, while AWS stood second and the GKE AutoPilot the third. GKE AutoPilot is 7.89%, 16,08% more expensive than the AWS EKS with AWS Fargate and GCP Cloud Run, respectively. On the other hand, GCP Cloud Run is 7.59% less costly than the AWS EKS with AWS Fargate.

In conclusion, predicting the monthly costs of using the GCP Cloud Run service is challenging because GCP Cloud Run creates more instances briefly to handle the traffic than the max configured allowed container instances at any given time. Based on the metrics data and observations, we found that the spike tests never produced more than six instances. In contrast, the incremental load tests were different; sometimes, they started nine, ten, and even twelve max instances simultaneously briefly, which could add up more to the bills. This is why the exact cost forecasting often might need to be corrected due to the inconsistency in scaling behavior. In contrast, serverless Kubernetes always adhere to the number of replicas stated in the HPA. As a result, the serverless Kubernetes cost forecast may be more accurate than the GCP Cloud Run. Nevertheless, the actual cost estimation is nearly impossible in serverless computing as the costs depend on the on-demand resource consumption [64].

5.3 Developer Experience

The developer experiences presented in this section are based on the author's experiences while using the services in the project. The developer experiences are divided into two sections based on the application development and deployment experiences on AWS, and GCP.

5.3.1 Application Development Experience

This section highlights the application development experiences for this thesis. After developing the web application, the entire Django project's only change is required in the *settings.py* file. In Django, a *settings.py* file contains information about the database configuration, API keys, the static files configuration, tightening the security-related configuration, and all other configuration information needed to run the web application. The deployment, particularly to Google Cloud Run, GKE AutoPilot, and AWS Kubernetes with Fargate, would not function with a uniform *settings.py* file for all services without modification. The application was set up in *settings.py* so that during the application runtime, the backend retrieves environment variables, secrets, and database connection secrets from the AWS and GCP Secret Manager to establish the database connection for the application to operate. This is why, during the project implementation phase, the *settings.py* file on the project level was tweaked so that application can be deployed following the best practices of AWS EKS, GKE AutoPilot, and GCP Cloud Run.

5.3.2 Application Deployment Experience

This section highlights the key takeaways from the service deployment experiences from AWS and GCP. Service providers' official documentation is the best source of truth to learn about the services. The most crucial thing for a developer to keep in mind when using Kubernetes services in the public cloud is that just because a feature is available in Kubernetes does not necessarily mean that it is enabled in serverless or servefull Kubernetes public cloud service offerings, for example, in AWS EKS and GKE AutoPilot. This is because these services are built on top of Kubernetes, with additional features and benefits distinct to each cloud platform. For instance, the metric server is baked with the Kubernetes cluster in GKE AutoPilot by default. So users do not need to install it after the cluster creation, unlike the AWS EKS cluster. Suppose the metric server is not installed in the AWS EKS cluster. In that case, HPA can not communicate to the resource metric or the custom APIs; as a result, HPA will not create replicas. However, for HPA to work, it needs to be made aware of the application Pod's resource utilization. A developer who does not have prior experience working with the public cloud's Kubernetes might get confused to understand why HPA works in GCP GKE AutoPilot but not in AWS EKS. This is why a developer must read the vendor's service-specific documentation in the first place to understand the service offerings and their limitations.

An ingress controller must be installed in the AWS EKS cluster after creating a cluster so that the service can be accessible to the public. Otherwise, the application

can not be made public, whereas the GCP does not require this step. The application is exposed to the public via an ALB when an Ingress service is created. For example, changing the *serviceType: LoadBalancer* in AWS in *service.yaml* manifest file does not create an ALB but makes the classic load balancer. The primary distinction between the two is that a Classic Load Balancer routes TCP (or Layer 4) traffic, while an ALB transports HTTP/HTTPS traffic (or Layer 7).

One interesting finding in AWS EKS Fargate was that every application Pod was deployed to its own Fargate node. The node was terminated when a Pod's life cycle ended. This behavior differs from GKE AutoPilot; many application Pods were scheduled to the same node as inspected, running the performance tests. The author observed that after creating the GKE AutoPilot and AWS EKS clusters, in each cluster, two nodes were created. In the GCP cluster, one node runs only system-related Pods, not allocating application Pods from the application deployment. On the other hand, the other node runs the application Pods in addition to the system-related Pods. When the HPA creates more replicas during the traffic surge, application Pods are hosted on newly created nodes.

GCP Cloud Run is among the simplest to deploy a service, compared to GKE AutoPilot and AWS EKS. GCP Cloud Run only needs a minimal configuration from a developer; it takes care of the rest. Developers can choose which GCP CloudSQL database instance an application should connect to and which secrets should be accessed from the GCP Secret Manager. In addition, they can configure or update environment variables, health check and change the HTTP request timeout. However, the default value of HTTP request timeout is five minutes. If necessary, they can specify the session affinity to serve the same visitors with request responses within the lifetime of the same container instances. They can configure a new revision to serve traffic to a specific percentage of visitors gradually rather than to all visitors instantly when it is launched. They can roll back an application revision to a previously deployed version at any time.

The Kubernetes manifest files were almost identical except for the *deployment.yaml*, and *ingress.yaml* files in AWS EKS with AWS Fargate and GCP GKE AutoPilot.

6 Conclusions and Future Work

This thesis assessed the viability of CaaS and PaaS services for serverless cloud computing in AWS and GCP for a web application based on performance, cost, and developer experience. The performance tests were carried out by running synthetic tests on CaaS and PaaS services to compare and contrast the performance variations. This study utilized GCP Cloud Run as a PaaS service; on the other hand, GCP GKE AutoPilot and AWS EKS with AWS Fargate as a CaaS service. The cost was analyzed based on the resource runtime consumption for the performance tests. The PaaS and CaaS services costs were contrasted. Finally, the developer's experience using the CaaS and PaaS services in developing and deploying a web application to each of those services was assessed.

6.1 Conclusions

The answers are based on the research work's inspections, performance test results, cost analysis, and findings.

RQ1 *Is a serverless CaaS and PaaS solution on the public cloud platforms for a containerized web application a viable approach regarding cost and performance?*

The performance of PaaS-based solutions between the first and the second generations was comparable. The first-generation execution environment performed, on average, far better than the second if we base our conclusions on the request latency. However, the first generation's warm instance had a lower request response time than the first generation's cold start instance overall. Furthermore, the second-generation execution environment outclassed the first regarding the higher request latencies, even though the second generation occasionally had higher latency. The first-generation cold start instance had the lowest latency for the container startup. Even though the high request response time can occur in the second generation of the execution environment occasionally, this could lead to a poor end users experience for the customer-facing applications.

The performance test results revealed that GCP Cloud Run had a lower latency and higher throughput. On average, GCP Cloud Run provided better performance in terms of request response time. However, certain users may experience more significant latency due to the occasional cold start, which is relatively low in AWS EKS with AWS Fargate. On the contrary, the spike test findings demonstrate that AWS EKS had lower latency and higher throughput per second. The GKE AutoPilot occasionally encounters higher latency, which directly contributes to unfavorable user experiences for the customer-facing web application. The results of the performance load testing indicate that GCP Cloud Run has lower latency and higher request throughput than CaaS-based solutions, which translates into lower costs and satisfied clients. In addition, the findings of the cost analysis demonstrated that GCP Cloud Run is more cost-efficient than CaaS-based solutions.

One of the PaaS platform's standout performance characteristics was its ability to scale quickly during an unexpectedly large spike in traffic without degrading request throughput and average latency. Inversely, the CaaS platforms can scale comparatively

slowly to accommodate the creasing load, directly contributing to a temporary increase in response time. The cost calculations in previous chapter showed that PaaS is considerably more affordable than the CaaS platforms for web a application with occasional significant traffic volumes. The PaaS-based solution would better suit large monolith web application projects.

In conclusion, PaaS-based solutions might be an option for monolithic web applications if the sporadic high latency never becomes a concern. However, the microservice-based web application is recommended to use the CaaS-based solutions as multiple services for a single web application can be run simultaneously, unlike PaaS. The smaller the images the faster to pull; therefore a container can initialise the application quickly and can serve the user requests much rapidly in CaaS. The multi-container features of GCP Cloud Run are in Preview a the time of writing this thesis. Once the necessary capabilities are added, GCP Cloud Run could be a great candidate to host a web application based on the microservice architecture. Nevertheless, the PaaS offerings from different public cloud vendors must be carefully considered. They may impact the application scaling and performance behavior while raising the costs.

RQ2 *How does developing and hosting a web application on a serverless CaaS and PaaS solution influence the development experience?*

With PaaS, a developer can deploy a web application in a matter of minutes while requiring little to no knowledge of the underlying technologies. The CaaS platforms, on the other hand, demand extensive knowledge of Kubernetes in order to employ a web application because the services are run on Kubernetes. Moreover, a developer's excellent expertise in Kubernetes is required to optimize resource usage, tighten security, improve application performance, and save costs. Additionally, third-party monitoring tools, such as DataDog and Grafana, must be integrated into CaaS for detailed monitoring of the web application's performance and API calls. Between the CaaS and PaaS, the least workload and knowledge are needed in PaaS to deploy a web application, which means that time to market is relatively faster compared to traditional architectures. This frees up a developer's time to build the business logic rather than tinkering how the pertinent services would interact if used in CaaS-based solutions. In summary, PaaS would be an economically viable choice compared to CaaS for an extensive monolith web application where significant traffic is anticipated, provided the occasional high request response time is not a concern.

6.2 Future Work

Due to time constraints, it was impossible to do a thorough analysis to improve the study or explore comparable services from other major public cloud vendors, such as Microsoft Azure, IBM Cloud, and Alibaba Cloud. Based on the findings of our study, additional work can be conducted to supplement this research, which has limitations. The author proposes the ensuing work that could be enhanced.

- This thesis research covered only two major public cloud vendors for serverless services. Additionally, the study could be expanded to use additional similar

service offerings from big players, including Microsoft Azure, IBM, and Alibaba Cloud.

- This project utilizes a fully managed database service from AWS and GCP. A database connection pooling solution could be enhanced for a fully managed database to improve database performance by temporarily storing the most frequently used queries in memory. Database connection pooling is a technique for keeping database connections open so other incoming connections can utilize the idle connections instead of creating a new connection for each call made to the backend from the frontend. Database connection pooling can be used in serverless Kubernetes to implement the per-container database connection limitations. For instance, a single production and test environment container instance may be limited to thirty connections simultaneously to the database. In contrast, a single development environment container instance may make up to fifty simultaneous connections. If not, the database connection limit will be reached when numerous connections are opened, eventually leading to additional CPU overhead. However, future studies may consider using a serverless database service to examine the price, performance, and developer experiences.

- A Content Delivery Network (CDN) service might be implemented to serve static content files, including images, CSS, JS, and even the most frequently used database queries. An API Gateway, which functions as a front door to access the backend services and assists developers in building, publishing, and securing APIs at any scale, would be an excellent addition to the architecture. The project implementation utilizes an ALB for serverless Kubernetes, which incurs a per-hour ALB fee; switching to an Nginx ingress controller would reduce cost if the Nginx runs as a Pod in the Kubernetes cluster.

- The web application's functionality can be expanded by including new features. The front and back ends might be separated using a well-known JavaScript-based framework, including Next.js or React.js. A payment system could be integrated with a third party, for example, PayPal. The API calls could be authenticated using a short-lived Token.

- The web application lacked real-world use cases or actual users, as synthetic tests were conducted to simulate the behavior of real users. Future enhancements might examine analyzing a real-world use case for cost and performance evaluation.

References

[1] Castro, P., Ishakian, V., Muthusamy, V., & Slominski, A. (2019). *The rise of serverless computing*. Communications of the ACM, 62(12), 44-54.

[2] Lee, H., Satyam, K., & Fox, G. (2018). *Evaluation of production serverless computing environments*. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD) (pp. 442-450). IEEE.

[3] Moreau, D., Wiebels, K., & Boettiger, C. (2023). *Containers for computational reproducibility*. Nature Reviews Methods Primers, 3(1), 50.

[4] Paakkunainen, O. (2019). *Serverless Computing and FaaS Platform As a Web Application*. Master's thesis, Aalto University, School of Science, Espoo, Finland.

[5] Hussein, M. K., Mousa, M. H., & Alqarni, M. A. (2019). *A placement architecture for a container as a service (CaaS) in a cloud environment*. Journal of Cloud Computing, 8(1), 1-15.

[6] Oludele, A., Ogu, E. C., Shade, K., & Chinecherem, U. (2014). *On the evolution of virtualization and cloud computing: A review*. Journal of Computer Sciences and Applications, 2(3), 40-43.

[7] Miell, I., & Sayers, A. (2019). *Docker in practice*. Simon and Schuster.

[8] Boonstra, L. (2021). *Definitive guide to conversational AI with dialogflow and Google cloud*. Berkeley: Apress.

[9] Ahmed Shaikh, K., Agaskar, S. S. (2022). *Introduction to Microservices and AKS*. Azure Kubernetes Services with Microservices: Understanding Its Patterns and Architecture, 1-24.

[10] Poulton, N. (2018). *Docker Deep Dive: Zero to Docker in a single book*. Nigel Poulton.

[11] McKendrick, R., & Gallagher, S. (2018). *Mastering Docker: Unlock new opportunities using Docker's most advanced features*. Packt Publishing Ltd.

[12] Burns, B., Beda, J., & Hightower, K. (2019). *Kubernetes: up and running: dive into the future of infrastructure*. O'Reilly Media.

[13] Chacon, S., and B. Straub. (2014) *Pro Git*. 2nd ed., Apress, p13-17

[14] Sayfan, G. (2017). *Mastering Kubernetes*. Packt Publishing Ltd.

[15] Lakshmanan, V. (2022). *Data Science on the Google Cloud Platform*. O'Reilly Media, Inc.

[16] Brikman, Y. (2019). *Terraform: Up & Running: Writing Infrastructure as Code*. 3rd ed., O'Reilly Media.

[17] Idrissi, A., & Abourezq, M. (2014). *SKYLINE IN CLOUD COMPUTING*. Journal of Theoretical & Applied Information Technology, 60(3).

[18] Abourezq, M., & Idrissi, A. (2014). *Introduction of an outranking method in the Cloud computing research and Selection System based on the Skyline*. In 2014 IEEE Eighth International Conference on Research Challenges in Information Science (RCIS) (pp. 1-12). IEEE.

[19] Saboor, A., Hassan, M. F., Akbar, R., Shah, S. N. M., Hassan, F., Magsi, S. A., & Siddiqui, M. A. (2022). *Containerized microservices orchestration and provisioning in cloud computing: A conceptual framework and future perspectives*. Applied Sciences, 12(12), 5793.

[20] Eder, M. (2016). *Hypervisor-vs. container-based virtualization*. Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), 1.

[21] Zhu, H., & Bayley, I. (2018, March). *If docker is the answer, what is the question?*. In 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE) (pp. 152-163). IEEE.

[22] Desai, A., Oza, R., Sharma, P., & Patel, B. (2013). *Hypervisor: A survey on concepts and taxonomy*. International Journal of Innovative Technology and Exploring Engineering, 2(3), 222-225.

[23] Truyen, E., Lagaisse, B., Joosen, W., Hoebreckx, A., & Dycker, C. D. (2020). *Flexible Migration in Blue-Green Deployments within a Fixed Cost*. In Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds (pp. 13-18).

[24] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C. C., Khandelwal, A., Pu, Q. & Patterson, D. A. (2019). *Cloud programming simplified: A berkeley view on serverless computing*. University of California at Berkeley, Electrical Engineering and Computer Sciences. arXiv preprint arXiv:1902.03383.

[25] Shah, J., & Dubaria, D. (2019). *Building modern clouds: using Docker, Kubernetes & Google cloud platform*. In 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC) (pp. 0184-0189). IEEE.

[26] Miller, S., Siems, T., & Debroy, V. (2021, October). *Kubernetes for Cloud Container Orchestration Versus Containers as a Service (CaaS): Practical Insights*. In 2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) (pp. 407-408). IEEE.

[27] Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N. J., Popa, R. A. & Patterson, D. A. (2021). *What serverless computing is and should become: The next phase of cloud computing*. Communications of the ACM, 64(5), 76-84.

[28] Davydov, D. S., Riabovol, D. A., Kramarenko, A. O., & Kvitka, A. V. (2020). *The role of cloud technologies in the digital economy*. Business inform, (8), 171-177

[29] Grzesik, P., Augustyn, D. R., Wyciślik, Ł., & Mrozek, D. (2022). *Serverless computing in omics data analysis and integration*. Briefings in Bioinformatics, 23(1), bbab349.

[30] Bentaleb, O., Belloum, A. S., Sebaa, A., & El-Maouhab, A. (2022). *Containerization technologies: Taxonomies, applications and challenges*. The Journal of Supercomputing, 78(1), 1144-1181.

[31] Dalbhanjan, P. (2015). *Overview of deployment options on aws*. Amazon Whitepapers.

[32] Pérez, A., Moltó, G., Caballer, M., & Calatrava, A. (2018). *Serverless computing for container-based architectures*. Future Generation Computer Systems, 83, 50-59.

[33] Nastic, S., Rausch, T., Scekic, O., Dustdar, S., Gusev, M., Koteska, B., ... & Prodan, R. (2017). *A serverless real-time data analytics platform for edge computing*. IEEE Internet Computing, 21(4), 64-71.

[34] Glikson, A., Nastic, S., & Dustdar, S. (2017). *Deviceless edge computing: extending serverless computing to the edge of the network*. In Proceedings of the 10th ACM International Systems and Storage Conference (pp. 1-1).

[35] Boniface, M., Nasser, B., Papay, J., Phillips, S. C., Servin, A., Yang, X. & Kyriazis, D. (2010). *Platform-as-a-service architecture for real-time quality of service management in clouds*. In 2010 Fifth International Conference on Internet and Web Applications and Services (pp. 155-160). IEEE.

[36] Serrano, N., Gallardo, G., & Hernantes, J. (2015).*Infrastructure as a service and cloud technologies*. IEEE Software, 32(2), 30-36.

[37] Eismann, S., Scheuner, J., Van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N. & Iosup, A. (2020). *A review of serverless use cases and their characteristics*. arXiv preprint arXiv:2008.11110.

[38] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V. & Suter, P. (2017). *Serverless computing: Current trends and open problems*. In Research advances in cloud computing (pp. 1-20). Springer, Singapore.

[39] Newman, S. (2019). *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019, p15-25.

[40] Neuman, S. (2015). *Building Microservices: Designing fine-grained systems*. Oreilly & Associates Inc.

[41] Richards, M. (2015). *Software architecture patterns (Vol. 4, p. 1005)*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Incorporated.

[42] Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", p4-10.

[43] Bui, T. (2015). *Analysis of docker security*. arXiv preprint arXiv:1501.02967.

[44] Mahmoudi, N. (2022). *Performance Modelling and Optimization of Serverless Computing Platforms*.

[45] Kalske, M., Mäkitalo, N., & Mikkonen, T. (2017). *Challenges when moving from monolith to microservice architecture*. In International Conference on Web Engineering (pp. 32-47). Springer, Cham.

[46] Eismann, S., Scheuner, J., Van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N. & Iosup, A. (2020). *A review of serverless use cases and their characteristics*. arXiv preprint arXiv:2008.11110.

[47] Khan, N., Ahmad, N., Herawan, T., & Inayat, Z. (2012). *Cloud Computing: Locally Sub-Clouds instead of Globally One Cloud*. International Journal of Cloud Applications and Computing (IJCAC), 2(3), 68-85.

[48] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). *Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade*. Queue, 14(1), 70-93.

[49] Wang, X., Shen, Q., Luo, W., & Wu, P. (2020). *RSDS: Getting System Call Whitelist for Container Through Dynamic and Static Analysis*. In 2020 IEEE 13th International Conference on Cloud Computing (CLOUD) (pp. 600-608). IEEE.

[50] Wong, A. Y., Chekole, E. G., Ochoa, M., & Zhou, J. (2023). On the Security of Containers: Threat Modeling, Attack Analysis, and Mitigation Strategies. Computers & Security, 128, 103140.

[51] Young, E. G., Zhu, P., Caraza-Harter, T., Arpaci-Dusseau, A. C., & Arpaci-Dusseau, R. H. (2019). *The True Cost of Containing: A gVisor Case Study*. In 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19).

[52] Venema, Wietse (2020). *Building Serverless Applications With Google Cloud Run*. 1st ed., O'Reilly Media, Inc., Chapter 1.

[53] Abourezq, M., & Idrissi, A. (2016). *Database-as-a-service for big data: An overview*. International Journal of Advanced Computer Science and Applications, 7(1).

[54] Kavis, M. J. (2014). *Architecting the cloud: design decisions for cloud computing service models (SaaS, PaaS, and IaaS)*. John Wiley & Sons.

[55] Al Shehri, W. (2013). *Cloud database as a service*. International Journal of Database Management Systems, 5(2), 1.

[56] Plekhanova, Julia (2009). *Evaluating web development frameworks: Django, Ruby on Rails and CakePHP*. Institute for Business and Information Technology 20 (2009)

[57] Van Eyk, E., Iosup, A., Seif, S., & Thömmes, M. (2017). *The SPEC cloud group's research vision on FaaS and serverless architectures*. In Proceedings of the 2nd International Workshop on Serverless Computing (pp. 1-4).

[58] Li, Z., Kihl, M., Lu, Q., & Andersson, J. A. (2017). *Performance overhead comparison between hypervisor and container based virtualization*. In 2017 IEEE 31st International Conference on advanced information networking and applications (AINA) (pp. 955-962). IEEE.

[59] Zahariev, A. (2009). *Google app engine*. Helsinki University of Technology, 1-5.

[60] Shahin, M., Babar, M. A., & Zhu, L. (2017). *Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices*. IEEE Access, 5, 3909-3943.

[61] Ding, R. (2020). *A Cloud-Based DevOps Toolchain for Efficient Software Development*. Master's thesis, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, The Netherlands.

[62] Menascé, Daniel A (2002). *Load testing of web sites*. IEEE internet computing 6.4: 70-74.

[63] Dias, W. K. A. N., & Siriwardena, P. (2020). *Microservices security in action*. Simon and Schuster.

[64] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C. C., Khandelwal, A., Pu, Q., ... & Patterson, D. A. (2019). *Cloud programming simplified: A berkeley view on serverless computing*. arXiv preprint arXiv:1902.03383.

[65] Ciaburro, Giuseppe, V. Kishore Ayyadevara, and Alexis Perrier (2018). *Hands-on machine learning on google cloud platform: Implementing smart and efficient analytics using cloud ml engine*. Packt Publishing Ltd, pp 59-64.

[66] Mell P., & Grance T. (2011). *The NIST Definition of Cloud Computing*.

[67] Serverless in the Enterprise, (2021). *Building the Next Generation of Efficient, Flexible, Cost-Effective Cloud Native Applications*. Available: https://www.ibm.com/downloads/cas/ZJLWQOAQ. Accessed 18 Jan 2023.

[68] Ma, Sha and Ben Linders (2021). *Github's journey from monolith to microservices*. Available: https://www.infoq.com/articles/github-monolith-microservices/. Accessed 26 Jan 2023.

[69] Reinhold, Emily (2016). *Lessons learned on uber's journey into microservices*. Available: https://qconnewyork.com/ny2016/ny2016/presentation/project-darwin-uber-jourbey-microservices.html. Accessed 20 Jan 2023.

[70] Sharwood, Simon (2021). *AWS US East Region Wobbles for Eight Hours: ec2 instances were impaired, redshift hurt, and some of you may still struggle to access your data*, the register. Available: https://www.theregister.com/2021/09/28/aws_east_brownout/. Accessed 21 Feb 2023.

[71] Azure Status History (2021). *Summary of Azure service interruption in the year 2021*. Available: https://status.azure.com/en-us/status/history/. Accessed 02 Feb 2023.

[72] Incident Affecting Google Cloud Infrastructure Components (2021).*Google cloud service status board*. Available: https://status.cloud.google.com/incidents/8DhiwfKvD987f5tJrj1G. Accessed 20 Jan 2023.

[73] Butler, Brandon (2012). *Amazon EBS Failure Brings down Reddit, Imgur, Others*. Available: https://www.networkworld.com/article/2160902/amazon-ebs-failure-brings-down-reddit--imgur--others.html. Accessed 20 Jan 2023.

[74] Clark, Jack (2013). *Amazon's Weekend Cloud Outage Highlights EBS Problems*. The red-headed stepchild of Bezos & Co's cloud just can't keep up. Available: https://www.theregister.com/2013/08/26/amazon_ebs_cloud_problems/. Accessed 02 Mar 2023.

[75] Samsung Database Migration Case Study (2020). *Samsung migrates 1.1 billion users across three continents from oracle to amazon aurora with aws database migration service*. Available: https://aws.amazon.com/solutions/case-studies/samsung-migrates-off-oracle-to-amazon-aurora/. Accessed 26 Jan 2023.

[76] Bradstock, D (2021). *Introducing GKE Autopilot: A Revolution in Managed Kubernetes*. Available: https://cloud.google.com/blog/products/containers-kubernetes/introducing-gke-autopilot. Accessed 25 Feb 2023.

[77] Costello, K., & Rimol, M. (2020). *Gartner Forecasts Worldwide Public Cloud End-User Spending to Grow 23% in 2021*. Available: https://www.gartner.com/en/newsroom/press-releases/2021-04-21-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-23-percent-in-2021. Accessed 20 Feb 2023.

[78] Sverdlik, Yevgeniy (2015). *Report: Aliyun Suffers 12-Hour Data Center Outage in Hong Kong*. Alibaba's cloud services arm offers conflicting explanations for prolonged downtime. Available: https://www.datacenterknowledge.com/archives/2015/06/25/report-aliyun-suffers-12-hour-data-center-outage-in-hong-kong. Accessed 02 Feb 2023.

[79] CNCF Serverless Working Group (2018). *CNCF WG-Serverless Whitepaper*. Available: https://github.com/cncf/wg-serverless/raw/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf Accessed 23 Jan 2023.

[80] Koomey J., Taylor J., (2017). *Zombie And Comatose Servers Redux*. Available: https://blog.anthesisgroup.com/zombie-servers-redux. Accessed 27 Feb 2023.

[81] Hayes, Andrew (2019). *GCP Cloud Run VS Cloud Function Cold Starts*. Available: https://adhayes88.medium.com/gcp-cloud-run-vs-cloud-function-cold-starts-cde538f7c675. Accessed 20 Feb 2023.

[82] Shanmugam, E., N. Kumar Bathula, and R. Alvarez-Parmar. *Start Pods Faster by Prefetching Images | Containers*. Available: https://aws.amazon.com/blogs/containers/start-pods-faster-by-prefetching-images/. Accessed 05 May 2023.

[83] Linux Kernel - CVE Details: *the ultimate security vulnerability datasource*. Available: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33. Accessed 01 Feb 2023.

[84] Namespaces(7) - Linux Manual Page. *Namespaces - overview of linux namespaces*. Available: https://www.man7.org/linux/man-pages/man7/namespaces.7.html. Accessed 07 Jan 2023.

[85] Microservice Architecture Pattern. *Microservice Architecture*. Available: https://microservices.io/patterns/microservices.html. Accessed 10 Feb 2023.

[86] Cloud Run release notes, 2023. Available: https://cloud.google.com/run/docs/release-notes. Accessed 04 Jun 2023.

[87] Monitor a Microservices App in AKS - Azure Architecture Center. *Distributed tracing*. Available: https://docs.microsoft.com/en-us/azure/architecture/microservices/logging-monitoring. Accessed 17 Jan 2023.

[88] Monolithic applications. *Monolith Apps in Azure*. Available: https://docs.microsoft.com/en-us/dotnet/architecture/containerized-lifecycle/design-develop-containerized-apps/monolithic-applications. Accessed 15 Feb 2023.

[89] Run the Docker Daemon As a Non-Root User. *Rootless Mode, prerequisite and limitations*. Available: https://docs.docker.com/engine/security/rootless/. Accessed 20 Feb 2023.

[90] Django Templates. Available: https://docs.djangoproject.com/en/4.2/topics/templates/. Accessed 12 Jun 2023.

[91] Alpine Official Docker Image, Docker Hub (2023). Available: https://hub.docker.com/_/alpine. Accessed 11 Jun 2023.

[92] Introduction to SaaS. Available: https://www.oracle.com/applications/what-is-saas/. Accessed 20 Feb 2023.

[93] Monolithic Architecture Pattern. Available: https://www.microservices.io/patterns/monolithic.html. Accessed 20 Mar 2023.

[94] Deconstructing the Monolith (Shopify Unite Track 2019) - YouTube. Available: https://www.youtube.com/watch?v=ISYKx8sa53g. Accessed 21 Feb 2023.

[95] How to Break a Monolith Application into Microservices With Amazon. *Break a Monolith Application into Microservices*. Available: https://aws.amazon.com/getting-started/hands-on/break-monolith-app-microservices-ecs-docker-ec2/. Accessed 21 Feb 2023.

[96] Implementing Microservices on AWS. *Microservices on AWS*. Available: https://d1.awsstatic.com/whitepapers/microservices-on-aws.pdf. Accessed 21 Mar 2023.

[97] Shared Responsibility Model - Amazon Web Services (AWS). *aws cloud security*. Available: https://aws.amazon.com/compliance/shared-responsibility-model/. Accessed 25 Mar 2023.

[98] Summary of the Amazon EC2, Amazon EBS, and Amazon RDS Service Event in the EU West Region (2021). Available: https://aws.amazon.com/message/2329B7/. Accessed 02 Feb 2023.

[99] Introduction to Database as a Service(DBaaS). Available: https://www.ibm.com/topics/dbaas. Accessed Jul 25 2023.

[100] Notable Cloud Outages of 2020. Available: https://totaluptime.com/notable-cloud-outages-of-2020/. Accessed 02 May 2023.

[101] *Docker Overview*. Available: https://docs.docker.com/get-started/overview/. Accessed 03 Jun 2023.

[102] *Overview of Docker Compose*. Available: https://docs.docker.com/compose/. Accessed 04 Jun 2023.

[103] Concepts of Kubernetes, *Kubernetes Documentation*. Available: https://kubernetes.io/docs/concepts/. Accessed 10 May 2023.

[104] Kubernetes Cluster Components. Available: https://Kubernetes.io/docs/concepts/overview/components/. Accessed 10 Feb 2023.

[105] Spot Instances - Amazon Elastic Compute Cloud. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html. Accessed 12 Feb 2023.

[106] Load testing scenarios. Available: https://k6.io/docs/using-k6/scenarios/. Accessed 27 Jun 2023.

[107] Containers As a Service (CaaS). Available: https://www.ibm.com/services/cloud/containers-as-a-service. Accessed 12 Feb 2023.

[108] Cloud Run General development tips. Available: https://cloud.google.com/run/docs/tips/general#run_tips_global_scope-python. Accessed 04 Jun 2023.

[109] Cloud SQL Service Level Agreement (SLA). Available: https://cloud.google.com/sql/sla. Accessed 12 Mar 2023.

[110] Amazon RDS Service Level Agreement. Available: https://aws.amazon.com/rds/sla/. Accessed 12 Mar 2023.

[111] Amazon Relational Database Service (Amazon RDS). Available: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html. Accessed 15 Jul 2023.

[112] Cloud SQL for MySQL, PostgreSQL, and SQL Server. Available: https://cloud.google.com/sql. Accessed 15 Jul 2023.

[113] Cloud Functions, Google Cloud. Available: https://cloud.google.com/functions. Accessed 12 Mar 2023.

[114] Google Cloud Run Concurrency, Google Cloud. Available: https://cloud.google.com/run/docs/about-concurrency. Accessed 16 Feb 2023.

[115] Introduction to GVisor. Available: https://gvisor.dev/docs/. Accessed 16 Feb 2023.

[116] Container Runtime Contract, Google Cloud. Available: https://cloud.google.com/run/docs/reference/container-contract. Accessed 16 Mar 2023.

[117] Linux/Amd64 - *gVisor syscalls table reference*. Available: https://gvisor.dev/docs/user_guide/compatibility/linux/amd64/. Accessed 12 Apr 2023.

[118] About Container Instance Autoscaling, Google Cloud Run. Available: https://cloud.google.com/run/docs/about-instance-autoscaling. Accessed 23 Jun 2023.

[119] General Development Tips, Google Cloud. Available: https://cloud.google.com/run/docs/tips/general. Accessed 20 Feb 2023.

[120] Google Cloud Run Pricing. Available: https://cloud.google.com/run/pricing. Accessed 21 Feb 2023.

[121] Gioia, Stefano (2018). *Have you ever considered ci/cd as a service?*. Available: https://blogs.cisco.com/cloud/have-you-ever-considered-ci-cd-as-a-service. Accessed 22 Feb 2023.

[122] Amazon EKS Service Level Agreement. Available: https://aws.amazon.com/eks/sla/. Accessed 10 Jul 2023

[123] Introduction to Cloud Computing. Available: https://www.ibm.com/topics/cloud-computing. Accessed 23 Jul 2023.

[124] Topics Understanding DevOps: CI/CD. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd. Accessed 22 Feb 2023.

[125] Azure DevOps, Microsoft Azure. Available: https://www.docs.microsoft.com/en-us/devops/deliver/what-is-continuous-delivery. Accessed 22 Feb 2023.

[126] Serverless Computing - IBM Cloud Education. Available: https://www.ibm.com/cloud/learn/serverless. Accessed 23 Feb 2023.

[127] Why Use Serverless Computing? Pros and Cons of Serverless. Available: https://cloudflare.com/learning/serverless/why-use-serverless/. Accessed 28 Jan 2023.

[128] CI/CD: The What, Why, and How. *Building automated workflows for faster releases*. Available: https://resources.github.com/ci-cd/. Accessed 06 Mar 2023.

[129] Introduction to Amazon EKS - Amazon EKS. Available: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html. Accessed 9 Jan 2023.

[130] Pod Execution Role - Amazon EKS. Available: https://docs.aws.amazon.com/eks/latest/userguide/pod-execution-role.html. Accessed 09 Jan 2023.

[131] Setting Request Timeout, Google Cloud Run. Available: https://cloud.google.com/run/docs/configuring/request-timeout. Accessed 19 Jan 2023.

[132] Cloud Reference Architectures and Diagrams, Cloud Architecture. Available: https://cloud.google.com/architecture/disaster-recovery. Accessed 17 Jan 2023.

[133] Stream container logs to CloudWatch in Amazon EKS. https://repost.aws/knowledge-center/cloudwatch-stream-container-logs-eks. Accessed 20 Jul 2023.

[134] About Execution Environments, Google Cloud Run. Available: https://cloud.google.com/run/docs/about-execution-environments. Accessed 26 Jan 2023.

[135] Selecting an execution environment, Goolge Cloud Run. Available: https://cloud.google.com/run/docs/configuring/execution-environments. Accessed 22 Jul 2023.

[136] Introduction to GCP Cloud Run. Available: https://cloud.google.com/run. Accessed 10 Jul 2023.

[137] Serverless Compute Engine – AWS Fargate FAQs – Amazon Web. Available: https://aws.amazon.com/fargate/faqs/. Accessed 22 Jan 2023.

[138] Infrastructure as a Service, IBM Cloud. Available: https://www.ibm.com/cloud/learn/iaas. Accessed 7 Feb 2023.

[139] Platform as a Service, Microsoft Azure. Available: https://azure.microsoft.com/en-us/overview/what-is-paas/. Accessed 20 Jan 2023.

[140] App Service Pricing, Microsoft Azure. Available: https://azure.microsoft.com/en-in/pricing/details/app-service/linux/. Accessed 07 Jun 2023.

[141] Sijbrandij, S. *Upcoming Changes to CI/CD Minutes for Free Tier Users on GitLab.Com*. Available: https://about.gitlab.com/blog/2020/09/01/ci-minutes-update-free-users/. Accessed 25 Apr 2023.

[142] *GitLab Pricing Model*. Available: https://about.gitlab.com/pricing/. Accessed 25 Apr 2023.

[143] *Plan & Pricing for K6 Cloud*. Available: https://k6.io/pricing/. Accessed 25 Apr 2023.

[144] *Grafana/K6: A Modern Load Testing Tool, Using Go*. Available: https://github.com/grafana/k6. Accessed 25 Apr 2023.

[145] *Introduction to K6*. Available: https://k6.io/docs/. Accessed 25 Apr 2023.

[146] *Pricing, Google Kubernetes Engine (GKE)*. Available: https://cloud.google.com/kubernetes-engine/pricing. Accessed 15 Jun 2023.

[147] Running Large Tests. Available: https://k6.io/docs/testing-guides/running-large-tests/. Accessed 10 Jul 2023.

[148] Cloud Build Serverless CI/CD Platform. Available: https://cloud.google.com/build. Accessed 15 Jul 2023.

[149] Comparison of Hypervisor Type 1 and Type 2. Available: https://www.ubackup.com/enterprise-backup/type-1-hypervisor-vs-type-2.amp.html. Accessed: 01 Jul 2023.

[150] Secret Manager Documentation. Available: https://cloud.google.com/secret-manager/docs. Accessed 15 Jul 2023.

[151] Cloud Password Management - AWS Secrets Manager - AWS. Available: https://aws.amazon.com/secrets-manager/. Accessed 15 Jul 2023.

[152] Amazon ECR Features. Available: https://aws.amazon.com/ecr/features/. Accessed 15 Jul 2023.

[153] Jess, B. (2006). *Amazon EC2 Beta*. Available: https://aws.amazon.com/blogs/aws/amazon_ec2_beta/. Accessed: 25 Jul 2023.

[154] Paul, M. (2008). *Introducing Google App Engine + our new blog*. Available: https://googleappengine.blogspot.com/2008/04/introducing-google-app-engine-our-new.html. Accessed 25 Jul 2023.

[155] GCP App Engine. Available: https://cloud.google.com/appengine/. Accessed: 25 Jul 2023.

[156] GCP Artifact Registry. Available: https://cloud.google.com/artifact-registry. Accessed 15 Jul 2023.

[157] AWS Lambda - Run code without thinking about servers or clusters. Available: https://aws.amazon.com/lambda/. Accessed: 21 Jul 2023.