Degree Project in Computer Engineering

First cycle, 15 credits

A comparison of Azure's Function-as-a-Service and Infrastructure-as-a-Service solutions

EDVIN ALVAEUS
LUDVIG LINDÉN

Stockholm, Sweden, 2023

A comparison of Azure's Function-as-a-Service and Infrastructure-as-a-Service solutions

EDVIN ALVAEUS

LUDVIG LINDÉN

Degree Programme in Computer Engineering
Date: June 21, 2023

Supervisor: Mira Kajko-Mattsson
Examiner: Håkan Olsson
School of Electrical Engineering and Computer Science
Swedish title: En jämförelse av Azures Function-as-a-Service och Infrastructure-as-a-Service molnlösningar

Abstract

Cloud computing is a growing industry. More and more companies are moving away from on-premise infrastructure. Instead, the choice is often to build their systems based on cloud services. This growth in the industry has brought with it new needs and consequently, new solutions. There have never existed as many different cloud providers and services offered by these providers. One of the newer paradigms in this industry is the serverless approach.

The problem of this thesis is that there is a lack of research into how Azure's serverless Function-as-a-Service offering compare to their more traditional Infrastructure-as-a-Service one. Therefore, the purpose of this work is to compare the two with regards to their performance, cost, and required developer effort. The goal is to provide a comparison that can help software professionals in choosing an appropriate cloud solution for their application. Additionally, it aims to contribute to the increased knowledge of modern serverless solutions while providing a basis for future research.

A qualitative method supported by measurements is used. The two cloud solutions are compared with regards to their performance, cost and developer effort. This is done by implementing and deploying equivalent Representational State Transfer applications with the two Azure offerings. The two implementations are stress-tested to measure their performance, and their incurred costs are compared. Additionally, the effort involved in developing the two solutions is compared by studying the amount of time required to deploy them, and the amount of code needed for them.

The results show that the serverless Function-as-a-Service solution performed worse under the types of high loads used in the study. The incurred costs for the performed tests were higher for the serverless option, while the developer effort involved was lower. Additionally, further testing showed that the performance of the Function-as-a-Service solution was highly dependent on the concept of cold starts.

Keywords

Serverless, Cloud computing, Load testing, Azure, REST

Sammanfattning

Molnbaserade tjänster är en växande industri. Fler och fler företag rör sig bort från att ha sin infrastruktur i egna lokaler. Istället väljer många att bygga sina system med molntjänster. Denna tillväxt inom industrin har fört med sig nya behov och med det nya lösningar. Det har aldrig tidigare existerat lika många molnleverantörer och molntjänster. En ny paradigm inom denna industri är det serverlösa tillvägagångssättet.

Problemet som denna uppsats ämnar att angripa är att det finns en brist på forskning som jämför Azures serverlösa Function-as-a-Service tjänst med deras mer traditionella Infrastructure-as-a-Service tjänst. Syftet med detta arbete är därför att jämföra dessa två med avseende på deras prestanda, kostnad och nödvändig utvecklaransträngning. Målet är att tillhandahålla en jämförelse som kan hjälpa arbetare inom mjukvaruindustrin att välja en passande molnlösning för deras behov. Utöver det ämnar detta arbete att bidra till en ökad kunskap kring moderna serverlösa tjänster, samt att tillhandahålla en bas för framtida forskning.

En kvalitativ metod understödd av mätningar används. De två tjänsterna jämförs med avseende på deras prestanda, kostnad och nödvändig utvecklaranstränging. Detta utförs genom att implementera och driftsätta likvärdiga Representational State Transfer-applikationer med de två tjänsterna från Azure. Implementationerna stresstestas för att mäta deras prestanda, och kostnaden för detta jämförs. Utöver det jämförs den anstränging som krävdes av utvecklarna för att utveckla de två lösningarna, detta genom att studera tiden som behövdes för att driftsätta dem och hur mycket kod som erfordrades.

Resultaten visar att den serverlösa Function-as-a-Service tjänsten presterade sämre för den typ av belastning som användes. Kostnaden för de utförda testerna var högre för den serverlösa tjänsten, medan den nödvändiga utvecklaransträngningen var lägre. Utöver detta visade ytterliggare testning att prestandan för Function-as-a-Service tjänsten till stor del påverkas av kallstarter.

Nyckelord

Serverlös, Datormoln, Belastningstestning, Azure, REST

Acknowledgments

We would like to thank our supervisor, Associate Professor Mira Kajko-Matsson, for continuous feedback and assistance during the thesis work. We would also like to thank our examiner, Håkan Olsson, for showing interest and allowing us to pursue this subject.

Stockholm, June 2023
Edvin Alvaeus Ludvig Lindén

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
FaaS	Function-as-a-service
GCP	Google Cloud Platform
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure-as-a-service
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
PaaS	Platform-as-a-service
REST	Representational State Transfer
SSH	Secure Shell
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine

Chapter 1

Introduction

There is a well-known phenomenon in the tech industry, that of buzzwords. These are words or phrases that become fashionable for a period of time. The term cloud computing can definitely be categorized as such a phrase. Cloud computing is not only a trendy word. Today, it is widely used by companies around the globe for hosting web-based servers, databases and other services (Gartner, n.d.).

A cloud server is like a computer that lives on the internet instead of being physically located on the premises. It's managed by a company that takes care of all the technical details, such as fixing it when it breaks and making sure it's always available. You can use the cloud server to store your files and run programs, and you can access it from anywhere with an internet connection.

1.1 Background

Cloud computing is a broad subject, encompassing many different types of services. Which service to choose for a particular purpose is not always obvious. Many times, similar results can be produced with several of them. The difference between them often lies in how a solution using them is structured, how it can be scaled, how much it costs, and to what extent its infrastructure needs to be managed (IBM, n.d.).

Serverless computing is the newest model in cloud computing, providing a way for developers to create applications without handling the physical or virtual infrastructure of systems. Instead, this is managed by the cloud provider and the developer can focus on the functionality of the program (Chowhan, 2018).

A serverless approach comes with certain aspects that need to be considered. The pricing model is different, employing pay-per-use billing instead of paying for the provisioned infrastructure. The performance guarantees from the providers are not the same as for their other services (Sbarski, Cui, & Nair, 2022). What this means in practice for someone using these services is not always clear, and there is a lack of up-to-date information regarding this.

Function-as-a-service (FaaS) is a type of serverless cloud service, allowing a customer to pay for the execution of functions without managing the underlying infrastructure (Chowhan, 2018). Infrastructure-as-a-service (IaaS) is a more traditional type of cloud service where the customer gets access to the hardware, often using Virtual Machines (VMs) (IBM, n.d.).

1.2 Problem

The problem that this thesis aims to address is the lack of comparative studies regarding FaaS offerings and IaaS ones.

1.3 Purpose

The purpose of this thesis is to compare a FaaS offering with a IaaS one with regards to their performance, cost and developer experience. This comparative study is conducted on Azure's cloud services.

1.4 Goal

The goal of this thesis is to help software professionals in choosing a cloud service for their applications as well as contribute to the increased knowledge of modern FaaS solutions while providing a basis for future research.

1.5 Research Methodology

In order to conduct a comparison between the FaaS and IaaS cloud solutions we needed in-depth knowledge within the cloud computing field. Therefore an extensive literature study was performed in order to gain the required knowledge. With the use of the knowledge gained from the literature study, a comparative study was performed. This comparative study could be categorized as a qualitative study supported by measurements.

1.6 Target Audience

The main target audience for this thesis is anyone looking for current information about modern cloud solutions, both within the industry and academia. In the industry, the thesis can serve as a guidance in choosing an appropriate cloud solution. In the academic world, the thesis can serve an educational purpose both within the subject of cloud solutions, and how they can be systematically evaluated with regards to their performance and pricing.

1.7 Scope and limitations

The thesis aims to investigate the qualities of serverless solutions and compare them to other cloud services. To limit the scope of the investigation, some restrictions have been put in place.

- There are several cloud providers that offer serverless solutions. Here, the choice has been made to limit the investigation to Azure's services.
- To be able to compare the two solutions, an application was developed and deployed using both of them. This application's functionality is not the focus of this investigation and was therefore kept simple.
- Security within cloud computing is a broad subject, and serverless solutions bring many new and unique challenges (Marin, Perino, & Di Pietro, 2022). This thesis does not deal with the security implications of the different solutions.

1.8 Benefits, Ethics and Sustainability

Cloud solutions bring sustainability questions into focus. Due to cloud operators being able to host multiple smaller virtual servers on a single physical server the operational cost for both the user and the provider is reduced. This can promote a sustainable economic development for the users, as the costs can be more tailored to the user's specific needs.

The costs of using cloud services can also be foreseeable and therefore easier to plan for. The serverless approach can further promote this aspect as the billing is pay-per-use (Tim & Rana, 2022). Cloud services also means that the number of required physical servers is reduced, compared to if all users had their own physical server on premise (Chowhan, 2018). Less usage of physical servers in general leads to less electricity usage and in turn less pollution.

1.9 Thesis Outline

The thesis' chapters are structured as follows:

- *Chapter 2: The cloud and its architecture:* Extended background where relevant concepts are explained.
- *Chapter 3: Research methodology:* Strategy and outline of the method used in the investigation.
- *Chapter 4: Practical steps:* A review of the practical steps taken to complete the study.
- *Chapter 5: Results:* The results of the investigation are presented.
- *Chapter 6: Analysis and discussion:* Analysis and discussion of the results and methods of the thesis.
- *Chapter 7: Conclusions and future work:* Conclusions drawn from the result of the study as well as proposals for future work.

Chapter 2

The cloud and its architecture

This chapter presents the background knowledge needed to follow the rest of the thesis. First is Section 2.1, which gives an overview of the contents of this chapter. The main part of the chapter starts of with Section 2.2, which presents information on different architectures and how they have evolved. Following this is Section 2.3, which presents information on servers. Both the traditional on-premise servers and cloud servers are discussed, and their differences are highlighted. The overview of servers are followed by Section 2.4. It presents an overview of two cloud services, highlighting their differences. The chapter is concluded with a short presentation of related work, found in Section 2.5.

2.1 Overview

The practice of cloud computing has become increasingly popular during the 21st century. Along with the growing popularity, new services and functionality have been added for customers to employ. The most recent service, FaaS, has quickly become a popular alternative to older services.

There are many different companies offering cloud services to customers. These companies are known as cloud providers. The three biggest providers are Amazon Web Services (AWS), Google Cloud Platform (GCP), and Azure (Dahal & Prasai, 2022). They all have their own versions of popular cloud services, such as IaaS and FaaS.

For many years in the tech industry, there has been a push towards decomposing applications into smaller and more manageable parts. Many developers are familiar with the architectural approach of microservices, designed to split the whole system into loosely coupled services. The newest step beyond the microservice architecture is the serverless approach. Calling the serverless approach serverless is in itself a misnomer, since it is dependent on a server to function. While the approach is still using servers to run, the management of these servers are abstracted away from the developer (Chowhan, 2018).

The concept of a server has also evolved throughout the years. Until recently, the term server was clear in its meaning, a physical computer serving content to other computers, either through a local network or the internet. After the rise of cloud computing, the term has become more ambiguous in its meaning. Now, the term could also mean a digital cloud server only accessed through the internet.

A cloud server can be used for many different purposes, all requiring different functionality. Therefore, there exists many available services that make use of cloud servers. The service that most resembles a physical server, IaaS, is commonly considered the first cloud service. It is designed to let developers rent physical servers hosted by cloud providers, and access the rented server through an internet connection. Another available service is the more recent serverless alternative, FaaS, abstracting the server management from the developer.

2.2 Architectural background

This section discusses the architectural background needed to understand the evolution of the serverless approach. It starts with an overview of the traditional monolithic architecture in Section 2.2.1. The following Section, 2.2.2, deals with the microservices architecture, and how it answers some of the issues of the monolith. In the final Section, 2.2.3, the serverless approach is presented.

2.2.1 Monolith

The traditional approach when building a software program is to construct it as one unified unit, a monolith (Newman, 2021). Figure 2.1 shows an example of a monolithic web application. It consists of a client-side browser view, a server and a database. The server handles requests from the client, performs business

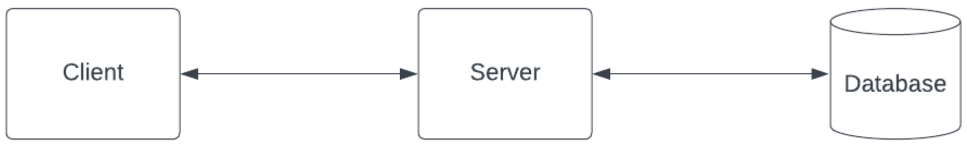


Figure 2.1: A web application with a monolithic architecture. Inspired by Roberts, 2018

logic and communicates with the database. This works well for certain use-cases, such as small programs that are not widely used. The deployment and scaling of such an application is straightforward (Newman, 2021).

When the system grows, either in its scope or usage, some issues can arise. Usually, a server application of a larger monolith contains many parts performing different tasks, henceforth known as services. Examples of these services could be an authentication service, a payment service and so on. The load on these services are often not distributed in a uniform manner, for example the authentication service may experience high traffic and need to be scaled up, while the payment service does not. As the server application consists of a single codebase, scaling a single service involves scaling the entire application, which can be highly inefficient (Kaplunovich, 2019).

Additionally, as new services are added and developers join the project, the complexity of it grows. With this, it can become harder to add new functionality, and to test isolated parts of it. It can also increase build time, negatively affecting the developer experience (Newman, 2021).

2.2.2 Microservices

The microservices architectural pattern is a newer approach to building software that recently has risen in popularity (Kaplunovich, 2019). Its main idea is to develop the application as a group of services that communicate with each other through message passing. The services are to be independently releasable, meaning that they can be changed and deployed without having to deploy any other services. Additionally, the services are modeled around business domains (Newman, 2021).

Figure 2.2 shows an example of a web application built with a microservices approach, consisting of two services. It illustrates another common pattern

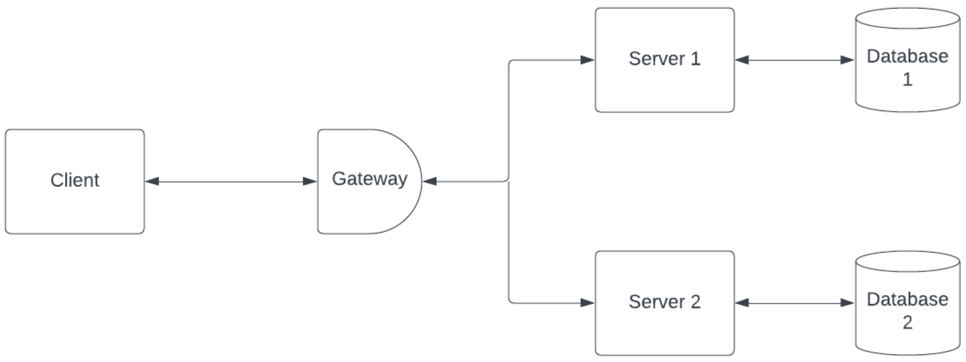


Figure 2.2: A web application with a microservices architecture. Inspired by Newman, 2021

in the microservices approach, that the services own their state in the form of their own database that only the service in question has access to (Newman, 2021). As there exists several possible destinations for an incoming request, a gateway is often used which routes it to the correct service.

This approach answers some of the issues of the monolithic one. Deploying the services separately enables them to be scaled separately. If one service is experiencing an increase in traffic, additional instances of that service can be deployed. The division of systems into smaller parts makes them more manageable. This can enable separate teams to work with one service without affecting others, assuming they are built in a loosely coupled manner. The modularity of the system also makes it easier to add new functionality. As long as it fits within the system's message passing scheme, adding it can often be done without impacting the rest of the system (Newman, 2021).

The microservices approach does bring some new challenges. While the management of the infrastructure, be it on-premise or in the cloud, is straightforward for a monolith, the same can not be said for a microservices application. As each service is deployed separately, it requires its own infrastructure that needs to be configured and maintained (Newman, 2021).

2.2.3 Serverless

The serverless approach to building software is a more recent approach. Views differ on whether it is a distinct architectural pattern or a modern way to implement the microservices pattern. Here, it is treated as its own architecture.

The term is a misnomer, since serverless applications still depend on servers to run, but the handling of them is abstracted away from the developers (Chowhan, 2018).

According to Sbarski, Cui, and Nair, 2022, a serverless offering needs to fulfill two requirements. First, it is to be consumed as a utility service. This means that access to it only happens through a well-defined Application Programming Interface (API), while the underlying infrastructure is hidden. Second, it is to only incur a cost when used, meaning there is no cost for just having it deployed.

A serverless system is often constructed from several different serverless offerings serving various purposes (Roberts, 2018). One example is serverless databases, such as Amazon's DynamoDB (AWS, n.d.). However the most common offerings are the various serverless function offerings, known as FaaS, which are further explained in Section 2.4.2.

Figure 2.3 shows an example of how the previously mentioned web application could be structured if it was built in a serverless manner using serverless functions. The difference from the microservices approach lies in the composition of the services, instead of being deployed as units to their own servers, they are further divided into granular functions and deployed as these units through the provider.

This approach can help solve the issues introduced by the microservices approach. As the cloud infrastructure is handled by the provider, orchestrating it is not a concern of the developer. The advantages that came with the microservices approach regarding modularity and extensibility are still present as the system is divided into manageable parts.

2.2.4 Representational state transfer

Today, Representational State Transfer (REST) is the de facto standard for offering a service on the web. It is a loosely defined architectural style for constructing APIs, and is often used to grant access to a set of resources (Neumann, Laranjeiro, & Bernardino, 2021). In a traditional client-server architecture, it can facilitate the retrieval and manipulation of server resources by a client through requests to said server. Its prevalence in the industry makes it a good basis for building applications for testing purposes.

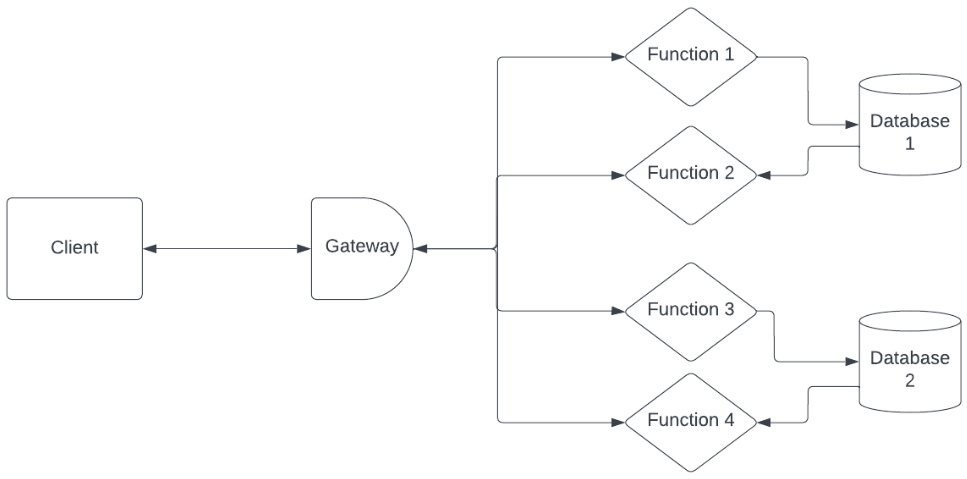


Figure 2.3: A web application with a serverless architecture. Inspired by Roberts, 2018

The standard contains some constraints which are to be followed if the system is to be referred to as RESTful. The first one is that all the resources or data elements are identified by global and unique addresses or identifiers within the system. This is usually achieved through the use of Uniform Resource Identifiers (URIs) (Pautasso, Wilde, & Alarcón, 2014).

Following this, the state of a resource is to be retrievable through a representation of it. In modern REST APIs, these representations are usually in the JavaScript Object Notation (JSON) format (Neumann, Laranjeiro, & Bernardino, 2021). The representations are also used for manipulating the state of a resource, for example by altering it with the intended changes and sending the updated version in a request (Pautasso, Wilde, & Alarcón, 2014).

Additionally, the messages being sent are to be self-descriptive. One common way to achieve this is through the use of the Hypertext Transfer Protocol (HTTP) for communication. The requests use the HTTP methods (GET, PUT, POST, DELETE, among others), where the semantics are clear in terms of their effect on the resource. The responses use HTTP status codes to convey the status of the request (Pautasso, Wilde, & Alarcón, 2014).

The final constraint declares that a RESTful system should use Hypermedia as the engine of application state which means that representations should include references to related resources, enabling decentralized resource discovery. In practice, this constraint is not widely implemented and many systems claim to be RESTful without satisfying it (Pautasso, Wilde, & Alarcón, 2014).

2.3 Servers

This section details the features of a traditional on-premise server in Section 2.3.1. Section 2.3.2 contains information about cloud servers. In Section 2.3.3, the two are contrasted and compared.

2.3.1 On-premise servers

Servers have evolved a great deal since they were first introduced, from the large data centers needed for a single computer to the modern server racks capable of performing a multitude of different tasks at once. More recently, servers have been split into two categories, one of them being on-premise and the other cloud servers. On-premise servers, commonly "on-prem", refers to the private data centers located at companies' own facilities, where they also maintain the data centers themselves (HPE, n.d.).

2.3.2 Cloud servers

Cloud servers is a term most commonly referring to the "public" cloud, where computation resources are rented from cloud providers. The rented resources are also shared by multiple companies or individuals who all use them on a as needed basis (HPE, n.d.). The concept of cloud servers has existed since the mid-1990s, although its popularity has increased greatly during the 2010s (Kranz, Hanelt, & Kolbe, 2016). As a result of this popularity shift, the functionality and usage of servers has changed greatly during the last 10-20 years.

2.3.3 Differences

At its core, a cloud server is infrastructure being rented out by a provider and accessed through the internet (HPE, n.d.). But there still exist some fundamental differences. First, the pricing structure is different. When implementing an on-premise server, one has to pay for the hardware, the maintenance and upkeep of the hardware, licenses and power consumption. Leading to a high initial cost and a moderate recurring cost. For cloud servers, there exist many different payment plans. Some employ a fixed monthly cost corresponding to the scale of cloud infrastructure rented, others incur a cost depending on expended computing resources. Common to them all is a low initial cost and a high recurring cost (Team Cleo, n.d.).

Second, the two approaches differ greatly when it comes to their scaling capabilities. There are two approaches when scaling a system, horizontal and vertical scaling. Horizontal scaling means adding additional machines, whereas vertical scaling means adding more power to the existing ones (Millnert & Eker, 2020). Scaling an on-premise solution with either of the approaches requires new hardware to be bought. This brings with it the risks of over- and under-provisioning of infrastructure, which can be costly.

Cloud solutions however, do not suffer from these issues, as the hardware is owned by the provider. Horizontal scaling of a cloud solution can be achieved by renting more infrastructure, while vertical scaling can be done by renting more powerful infrastructure. Both of these are achieved by expanding the existing cloud service plan, and can be done in a granular way, so as to avoid renting the wrong amount (Fisher, 2018). Nowadays, the cloud providers also offer the ability to grow or shrink the provisioned infrastructure resources dynamically to adapt to workload changes, which is called elasticity (Galante & de Bona, 2012). This can make sure that only the required resources are being paid for at any given time.

Third, the maintenance of the two approaches also differ. Regardless of which approach the customer is using, the configuration of operating systems, communication and security has to be partly handled by the customer. While both approaches include the same elements of maintenance, the difference lies in who the responsibility falls on. Maintenance of the on-premise servers must be handled by the company who owns them. This means that the company has to have someone comfortable and educated in maintenance of servers, or hire a third party to handle it (Team Cleo, n.d.).

The cloud solution shines in comparison. The maintenance of cloud servers is handled by the owner of the underlying physical servers, the cloud provider. The customer renting the cloud server can focus their efforts elsewhere. As the provider tends to own a large amount of servers, they probability that they have encountered any particular problem before is high. Therefore, they are often better equipped to handle any potential issues.

2.4 Cloud services

In this section, different aspects of two cloud services are discussed. Section 2.4.1 and 2.4.2 presents the two different cloud solutions IaaS and FaaS respectively. The section is concluded with a discussion of the cloud solutions' differences in Section 2.4.3.

2.4.1 Infrastructure-as-a-Service

IaaS is often considered the original cloud service. It is a way for customers to rent infrastructure through the cloud. Customers get to provision and configure the infrastructure in a manner similar to that of an on-premise server. The difference is that the cloud provider manages, hosts and maintains the servers. The IaaS user only accesses the infrastructure through an internet connection (IBM, n.d.). Users can often choose between using an unshared physical server or a VM hosted on shared hardware.

2.4.2 Function-as-a-Service

FaaS is the newest of the cloud services and is already widely available through many providers. FaaS is an event driven solution that allows customers to run code without having to provision and manage a server. FaaS being event driven means that your code, rather than running all the time, only runs as a response to an event configurable by the user. Consequently, your code only runs when it is needed. As a customer, you only pay for the time your functions run (Chowhan, 2018).

The scaling of your application is also handled by the provider, if you need to execute your function many times in parallel, the provisioner scales up the infrastructure in order to perform just that. The containers that are created to run your code is ephemeral, meaning that they are automatically destroyed as soon as the function is finished executing.

Deploying your code is also different, since there is no server that you need to update with your new code. Instead, the provider handles the code you upload and makes sure that the newest code is used in following event triggers.

An important concept regarding FaaS solutions is that of cold starts. The ephemeral nature of the containers that run the code can lead to not enough

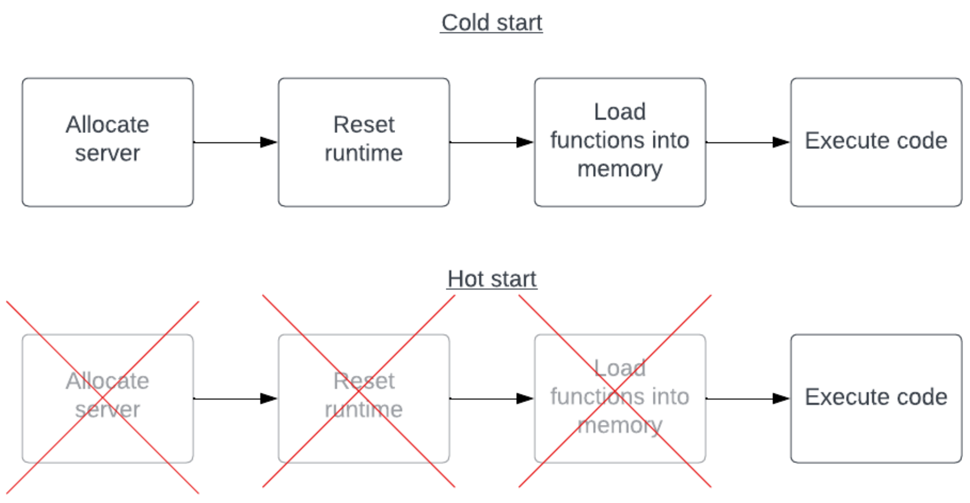


Figure 2.4: Illustration of the concept of cold starts. Inspired by Colby Tresness, 2018

active containers existing when the code is to be executed. Initializing these containers adds latency to the execution time of your functions (Chowhan, 2018). Figure 2.4 illustrates this concept, showing the steps that need to be completed before the code can run during a cold start.

2.4.3 Differences

While both IaaS and FaaS are cloud solutions, they differ in many ways. One big difference lies in the amount and type of code that is required from the developer. Using IaaS, the code required is the same as if you deployed the program on your own server. This differs greatly from FaaS, in which the code required is limited to the specific functions of the program. Using FaaS means that many parts of coding needed for a full architecture can be ignored, such as organizational and architectural code.

Another difference lies in the scaling of the services. While both approaches can be used as a microservice application, the FaaS solution requires no different actions from the developer. Creating a microservice application with IaaS requires separate provisioning and scaling of infrastructure for the separate services. As the provisioning is abstracted away from the customer with FaaS, this can instead be handled by the cloud provider.

There is also a difference in the billing of the two approaches. IaaS has a pricing model based on usage and capacity, often meaning a monthly or hourly cost for provisioning the VMs or servers. In contrast, the FaaS solution follows a pay-per-use model. This means that the customer pays for the number of function invocations and execution time (Azure, n.d.-c; Google, n.d.-b).

2.5 Related work

Cloud computing is a popular subject and there exists multiple previous works that this thesis builds upon. This section presents those works and explains how they influenced our thesis.

Jääskeläinen, 2019 measures the performance and scalability of three Azure services. The first one is a traditional IaaS deployment service. The second one is a Platform-as-a-service (PaaS) service. Here, the application is deployed to an environment managed by Azure, letting them handle the scaling and upkeep of the infrastructure. The third one is the retired Azure Service Fabric Mesh service, where containerized applications were deployed in a serverless manner, not based on FaaS.

Malla and Christensen, 2019 performs a comparison between FaaS and IaaS with the basis of High Performance Computing (HPC). This comparison is performed on GCP's versions of the cloud solutions, Cloud Functions and Compute Engine. The comparison measures performance and cost for the two approaches with a sample HPC workload.

Villamizar et al., 2016 performs a case study where different deployment methods available through AWS are compared with regards to their infrastructure cost. It compares a monolith deployed through an IaaS solution, a microservices application deployed through an IaaS solution, and a microservices application deployed using FaaS. Although the study includes a performance comparison, it is not the focus of it. Instead, the performance measurements are used to get an even basis for the cost comparison.

From these previous research papers, some inspiration was taken for the work in this thesis. The way the measurements were done in Jääskeläinen, 2019, by deploying REST APIs to each service and running several different load tests, inspired the layout of the performance analysis done in this thesis. The usage of both FaaS and IaaS for the comparison in Malla and Christensen, 2019 is what inspired us to compare those two approaches.

Chapter 3

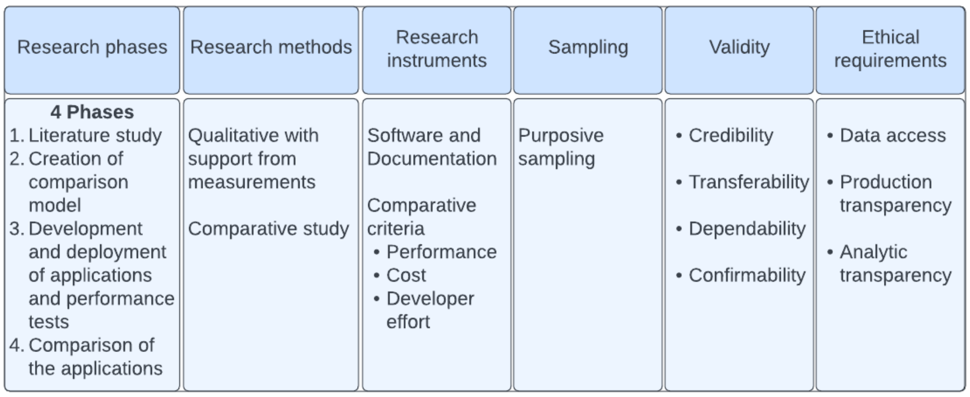
Research Methodology

This chapter presents the research methodology used for this thesis. Section 3.1 contains an overview of the strategy that was adopted. Section 3.2 introduces the phases of the conducted research. In Section 3.3, the method used for the thesis work is explained. Section 3.4 presents the different instruments used during our research. The sampling method of the research is explained in Section 3.5. In Section 3.6, the validity threats of the research are presented. Section 3.7 contains a discussion of the ethical requirements for this thesis. The chapter is concluded with Section 3.8 introducing the problems and experiences encountered during the thesis work.

3.1 Research Strategy

The research area that this thesis addresses is both vast and quickly evolving. Due to the fast changes in the area, many papers regarding the subject become less relevant just a few years after being published. To provide a research result that could be as long lasting as possible, while still carrying out the research within the given time frame a good research method had to be chosen.

The chosen research method is shown in Figure 3.1. Our strategy consisted of five different components: (1) research phases, (2) research methods, (3) research instruments, (4) sampling, (5) validity threats, and (6) ethical requirements. These components are all discussed further in other sections of this chapter.



Research phases	Research methods	Research instruments	Sampling	Validity	Ethical requirements
4 Phases 1. Literature study 2. Creation of comparison model 3. Development and deployment of applications and performance tests 4. Comparison of the applications	Qualitative with support from measurements Comparative study	Software and Documentation Comparative criteria • Performance • Cost • Developer effort	Purposive sampling	• Credibility • Transferability • Dependability • Confirmability	• Data access • Production transparency • Analytic transparency

Figure 3.1: Overview of the research strategy used for this thesis

3.2 Research Phases

This section presents the phases of the work of this thesis. Figure 3.2 shows how the thesis work was divided into four phases. Section 3.2.1 outlines the literature study phase. Section 3.2.2 deals with the creation of the model used in our comparison. In Section 3.2.3, the phase where the applications were developed and deployed is presented. Finally, in Section 3.2.4, the phase where the applications were compared is presented.

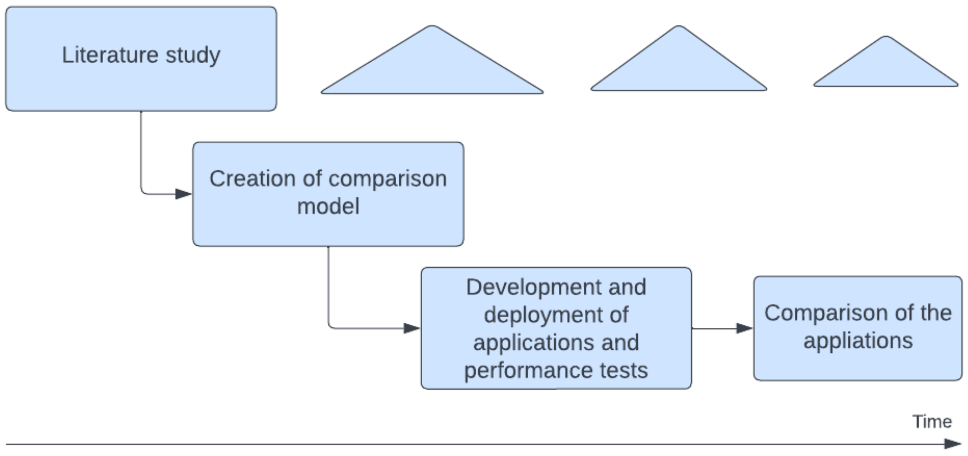

Literature study
Creation of comparison model
Development and deployment of applications and performance tests
Comparison of the applications
Time

Figure 3.2: Overview of the research phases of this thesis

3.2.1 Literature study

The literature study phase was conducted in conjunction with the formulation of this thesis' problem. Entering this phase, a basic idea of what was to be investigated existed, but it was not complete. There were two main goals of this phase. The first one was to find and examine the relevant research that had previously been done within the field of cloud computing in general, and serverless computing in particular. The second one was to get a better understanding of the relevant technical aspects of the field. Both goals were targeted simultaneously, but they are presented as distinct parts for clarity.

To achieve the first goal, established research databases were consulted. These were mainly IEEE Xplore and Google Scholar. The databases were queried with relevant keywords including, but not limited to, the following: *serverless*, *serverless computing*, *serverless comparison*, *serverless load test*, *serverless cost*, *Function-as-a-Service*, *FaaS*, *FaaS comparison*, *FaaS load test*, *FaaS cost*, *Azure Functions*, *Azure Functions comparison*, *Azure Functions load test*, *cloud computing*, *cloud comparison*, *developer experience*, *developer experience cloud*, *developer experience serverless*. A large amount of previous research was found, some of it close to the idea of this thesis' research question (Jääskeläinen, 2019; Malla & Christensen, 2019; Villamizar et al., 2016). From this, the problem of the thesis could be formulated.

During the search for related research, some sources with a more general content were also identified. Three of these were referenced frequently in existing research, so the choice was made to focus on these to achieve the second goal of this phase (Chowhan, 2018; Roberts, 2018; Sbarski, Cui, & Nair, 2022). By reading the relevant parts of these, a good understanding of the technical aspects underlying this thesis was gained.

This phase did not end at this stage. As can be seen in 3.2, it continued with a decreasing intensity throughout the whole work of the thesis. Some research was consulted frequently during the creation of the comparison model (Villamizar et al., 2016), and some during the design and deployment of the applications that were to be compared (Chowhan, 2018).

3.2.2 Creation of comparison model

This phase consisted of the creation of the model that was to be used to perform the comparison of this thesis. During the previous literature study phase, the problem that served as the basis of the thesis had been formulated. The goal of this phase was to construct a comparison model that could be used to tackle this problem. It had been decided that three main aspects of the cloud solutions were to be included in the comparison. These were performance, cost and developer effort. What follows is an outline of how these three aspects were handled while creating the model. A more detailed description of exactly what criteria were included for each aspect in the model can be found in Section 3.4.

The inclusion of the performance aspect was integral to the structure of both the comparison model and the applications being compared. It is what motivated the use of a load testing tool during the execution of the comparison. As there existed previous research that measured the performance of cloud solutions, inspiration could be taken from these sources when creating our model (Jääskeläinen, 2019; Villamizar et al., 2016). Although the concept of performance is vast, the literature study indicated that most of the previous research focused on the same established criteria. Therefore, this thesis also incorporates these criteria in its comparison model.

As the concept of cost is broad, a decision had to be made regarding what it entailed in the context of this thesis. The choice was made to focus on the monetary cost of the cloud solutions. As the pricing model of the two solutions differed greatly, an effort was made to create criteria that took this into account. Existing research was consulted to get an idea of how this could be done, and inspiration was drawn especially from Villamizar et al., 2016.

The subject of developer experience can be hard to define, as it includes several different broad concepts. Fagerholm and Münch, 2012 characterizes it as consisting of three aspects: the experience regarding the development infrastructure, the experienced feelings regarding one's work, and the experiences around the value of one's own contributions. Here, the focus was on a small subset of the first aspect, henceforth called developer effort. The goal was to create measurable criteria that could represent this aspect, without becoming too subjective. During the literature study, previous research measuring this could not be found. Therefore, the criteria are based on some of the aspects that cloud providers claim are advantages of serverless solutions.

3.2.3 Development and deployment of applications and performance tests

This phase of the thesis work involved creating the two applications needed to perform the comparison as well as deploying them to the cloud services. The first part of the phase consisted of formulating the requirements and designing the applications. The applications needed to provide us with enough data during testing to perform our comparison. The performance tests were to subject the applications to varying amounts of traffic, and provide the data needed for the comparative criteria.

The second part consisted of implementing the designed applications and performance tests. Using the design, our previous programming knowledge, and relevant documentation, we could create both the programs and the performance tests for our purpose.

The next part of this phase was to deploy the two applications to their respective cloud services. As mentioned in earlier chapters, we needed one application for each of the two cloud services. The applications were to include the same functionality. As the applications needed to be deployed through IaaS and FaaS respectively, they needed to include parts of code specific to those services. Because of this, the programs could not be identical in their entirety, but instead only their base functionality was to be identical.

A more in depth discussion of the practical steps taken to develop the two applications, as well as the performance tests, can be found in Chapter 4.

3.2.4 Comparison of applications

The goal of this phase was to apply the created comparison model to the applications created in the previous phase. This model could then be used to gather data about the applications and allow conclusions to be drawn. As mentioned in Section 3.2.2 the comparison consisted of three major aspects, performance, cost and developer effort. The main part of this stage was therefore the execution of performance tests. When the tests had been executed, the results relevant to the comparison model had to be produced. This part included cost calculations and analysis, as well as evaluating the time spent and amount of code written. This part of the phase is explained further in Section 4.4.

After the comparison was performed, the gathered data on performance, cost and developer effort was used to draw conclusions and to analyze similarities and differences between the two solutions. The conclusions and discussions about gathered data can be found in Chapter 6 and 7.

3.3 Research methods

This section presents the research methods chosen for this thesis. In Section 3.3.1 the type of research is discussed and motivated. And in Section 3.3.2 the type of the study is introduced.

3.3.1 Qualitative or quantitative research

In order to accurately answer this thesis' research question, the research type had to be chosen. For this thesis, the type chosen was a qualitative study with support from measurements. Choosing the right approach was decided by analyzing the nature of our thesis' problem and the main characteristics of the qualitative and quantitative approaches.

A qualitative approach entails gaining understanding of reasons and motivations for questions. Additionally, the samples used in qualitative research are often small and a representation of specific cases (Bhandari, 2020b). These two characteristics described our research well, since we worked with and compared two specific situations. We were also trying to understand the reasons behind our results, to determine why one cloud solution is better than another.

The quantitative approach also has characteristics that could be found in our research, mainly working with numerical data and measurements. However, our research could not be classified as quantitative research since one of the main characteristics of the quantitative approach is a large sample size of data, which is also often randomly sampled (Bhandari, 2020a). These characteristics did not apply to our research, which used only two specific samples for gathering data.

3.3.2 Comparative study

Performing a comparative study involves contrasting two or more phenomena of interest in order to identify their similarities and differences. The goal of a comparative study is to gain a better understanding of the phenomena under investigation, and to help make informed judgements about their advantages and disadvantages (Miri & Dehdashti Shahrokh, 2019). The purpose of this thesis was to compare two cloud offerings from Azure, focusing on their price, performance, and developer effort. One of the goals was to help software professionals make an informed decision about these cloud solutions.

A comparative study has some inherent advantages, one of them being the ability to gain understanding of a phenomenon by juxtaposing it with a more familiar one (Miri & Dehdashti Shahrokh, 2019). The phenomena that were compared in this thesis was a well understood cloud solution, IaaS, and a less familiar one, FaaS. The contrasting of these two provided a context for understanding the FaaS solution, as it could be studied in relation to the IaaS one.

The choice of method brought with it some possible disadvantages as well. First, a comparative study can be vulnerable to bias if the differences between the two phenomena are not controlled for. Additionally, the transferability of a comparative study can be low as the things being compared are often specific in their nature (Miri & Dehdashti Shahrokh, 2019). These threats and how they were met are discussed further in Section 3.6 and 6.4.

3.4 Research instruments

This section discusses instruments used for the performed research. Section 3.4.1 briefly presents the documentation and software instruments used. And Section 3.4.2 introduces the comparative criteria used for our comparison.

3.4.1 Software instruments and documentation

In order to conduct this research, some software instruments had to be employed. The main one was the software used for the performance tests, k6. It is a load testing tool aimed at developers wanting to test their applications under different loads. k6 allows the developer to write test scripts in JavaScript. The tests scripts can be configured to produce different loads and output reports. The tests can then be executed using k6 command-line interface. k6 allows the tests to be run both locally and through their own cloud configuration. In this study, the tests were executed locally from one computer. By varying the number of virtual users in the test scripts, loads that mimicked traffic of real users could be produced locally.

Additionally, the developed and deployed applications can also be considered software instruments. Executing the k6 tests on them produced usage data and statistics through the cloud provider, Azure. These metrics were used for the comparison.

The research also included using documentation regarding the two cloud services. This was both general documentation to aid us in developing our applications, as well as pricing documentation to enable us to calculate the monetary cost of the cloud solutions. All documentation for the cloud services could be found on their respective websites.

3.4.2 Comparative criteria

As mentioned in Section 3.2.2 our comparative criteria were based on three major aspects. These were, performance, cost and developer effort. The aspects were further divided into comparative criteria to easily measure the desired aspects, shown in Figure 3.3. Section 3.4.2.1, 3.4.2.2 and 3.4.2.3 discusses these aspects further.

3.4.2.1 Performance

The aspect performance was divided into two smaller criteria, average response time and error percentage. The average response time criteria measured the response time of our applications using the load testing software mentioned earlier. This was done because a program subjected to many requests may experience high response times, this time can be a good measurement of its performance. The expected outcome of this criteria was that the IaaS solution would show higher response times when subjected to high loads, whereas the FaaS solution would show a more even distribution with small deviations because of cold starts.

The other criteria for performance, error percentage, measured how many of the requests sent to our programs resulted in error responses. This data was also gathered using the load testing tool and documented with its statistics.

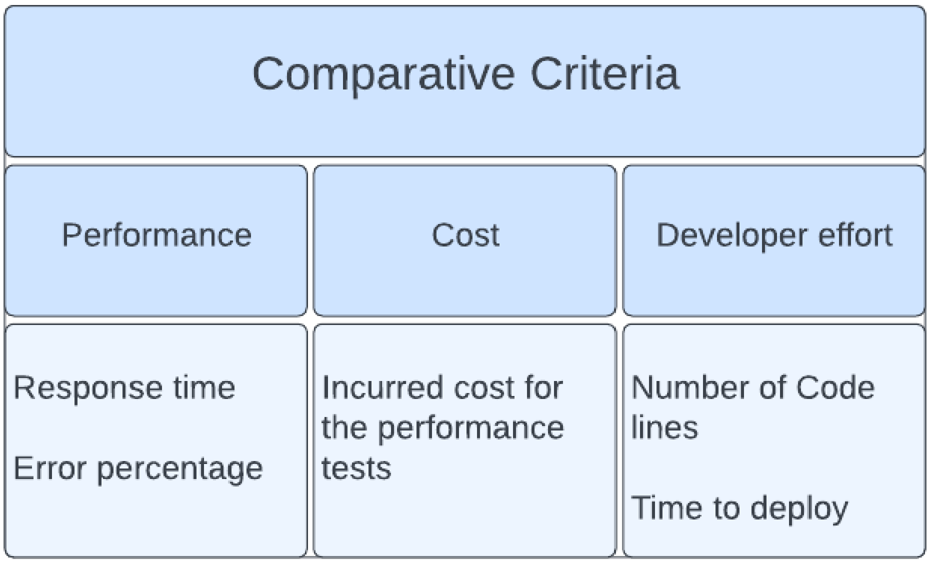


Figure 3.3: Overview of the comparative criteria

This criteria was also a good way to measure performance since, just like the response time, a program under high load might encounter errors when pushed beyond its limits. Here we also expected that IaaS would exhibit a higher error percentage when pushed to its limit while FaaS would barely result in any errors.

3.4.2.2 Cost

The next main aspect to measure was cost, which in our case was defined as the cost for performing our load tests. This criteria measured the total cost for our tests, performed on our deployed cloud applications. This measurement only showed the financial cost of each of the cloud solutions in our specific situation. Our experiment resulted in the applications only being deployed on the cloud services for a short period of time, while being subjected to a high load during the performed tests. Therefore, the expected measurements of these solutions were that the IaaS application would result in lower costs than the FaaS application. This was expected since the IaaS application would only result in costs during the short duration of being deployed. In contrast, the FaaS solution would be billed for our usage of the function, which with the high load would result in a higher cost.

3.4.2.3 Developer effort

Our last main aspect was developer effort. This consisted of measuring the number of code lines and the time to deploy. These criteria were both some way to measure how easy the cloud solution was to work with. The first criterion for developer effort was the number of code lines needed for each application. This criterion was highly dependent on the context of this study, both regarding the author's previous experience and the developed applications' structure. This criterion was a good measurement of developer effort since an application needing more code lines could lead to ineffective use of the developer's time. Since the IaaS solution would have to include boilerplate code for the architecture of the server, the expected measurements of this criterion was that the IaaS would have a considerably higher number of code lines.

The second criterion for developer effort was time to deploy. Here, we measured the time it took for us to fully deploy our two applications to the cloud services. This measurement included the steps configuration, provisioning and deployment. Our expected outcome of this criterion was that the application for FaaS had a faster deployment time since the configuration and provisioning steps with FaaS are basically nonexistent.

3.5 Sampling

The sampling method of a study describes the process that lead to a certain sample of a larger population being included in it. In the case of a qualitative study, these often consist of human respondents (Taherdoost, 2016). This study did not include any such respondents. Here, the sampling method was concerned with the choice of the phenomena included in the comparative study, that is the cloud services.

First, the sampling of this study was non-probabilistic, as the cloud services were not chosen in a random manner from the entire population of available services. Instead, there existed a motivation for the inclusion of them. This included the choice of provider, and in turn services offered through this provider. Therefore we chose to use purposive sampling as our sampling method.

Purposive sampling is a method where certain elements of a population are chosen to be included for a reason. The researcher uses their own judgement when deciding on what to include in the sample. The basis is often the belief that important information can only be obtained from these choices (Taherdoost, 2016).

For this study, the first sampling choice that had to be made was which cloud provider to use. Azure was chosen, and the reason for this was the problem that had been identified, that there was a lack of comparative studies on Azure's serverless offerings.

The second sampling choice was concerned with what type of services to compare. As the focus of the study was to be serverless computing, a decision had to be made regarding which serverless offering to use. FaaS was chosen as it was the most widely used serverless option. One advantage of a comparative study is the ability to elucidate a less familiar concept by juxtaposing it with a more well known one (Miri & Dehdashti Shahrokh, 2019). As the decision had been made to perform a comparative study, a more familiar second member was to be chosen to compare with FaaS. IaaS was chosen as it was the most conventional alternative. It closely resembles the usage of an on-premise server, while also being the option furthest removed from the concept of serverless computing.

3.6 Validity

The validity of qualitative research can be assessed by focusing on four criteria. These are: *credibility*, *transferability*, *dependability* and *confirmability* (Guba, 1981). Here we present the potential threats to our thesis work, and in Section 6.4 how we combated the threats is discussed.

- *Credibility*: The concept of *credibility* corresponds to the concept of internal validity in quantitative research. It deals with the consistency of the findings with reality, that is how well the findings coincide with and describe what is actually the case (Shenton, 2004). The low complexity of the applications in this study made this a possible threat to the validity, as an application in a real-life scenario is often more complex.

- *Transferability*: The *transferability* of qualitative research findings refers to its ability to be applied generally, that is if the results can be transferred from its own context to a different one (Guba, 1981). This thesis' comparative study was specific in its nature as it dealt with two distinct cloud solutions from one provider. Therefore, a threat to the study's transferability had to be considered.

- *Dependability*: For qualitative research, the *dependability* corresponds to the concept of reliability in quantitative work, which is the degree to which it could be repeated while achieving the same results. Due to the nature of qualitative research, this is often hard to attain, and the focus is often instead on just being able to repeat it (Guba, 1981).

- *Confirmability*: The concept of *confirmability* in qualitative research corresponds to the objectivity of quantitative research (Shenton, 2004). It is concerned with the researcher's inherent bias not affecting the findings and the conclusions drawn from them. The subjective nature of some of the criteria of the comparison model led to this being a possible threat to the validity of this study.

3.7 Ethical requirements

The ethical requirements of a qualitative study often concern the handling of respondents by focusing on principles such as informed consent, confidentiality, and beneficence (Kang & Hwang, 2021). As the study of this thesis did not involve any respondents, these principles were not applicable here. Instead, the ethical requirements of this study were concentrated on the way the data was gathered, utilized, and presented.

The first principle that was adhered to was that of data access. This principle states that the data used for making claims should be accessible to the reader (Lupia & Elman, 2014). In this thesis, this was achieved by including the data gathered during the comparison in an appendix.

The second principle followed is called production transparency. To adhere to it, researchers should include a full account of how data was gathered (Lupia & Elman, 2014). This was done by including a full account of steps taken to produce the data in Chapter 4. Additionally, the source code of applications was included in Appendix A.2 to help enable the reproduction of them.

Finally, the principle of analytic transparency was followed. It demands that it should be clearly explained how the claims being made are linked to the underlying data (Lupia & Elman, 2014). This was done by clearly referencing the data while discussing the results in Chapter 6.

Together, these principles ensured the integrity of the study and its data. By including the data, how it was gathered, and what it was used for, the risk of it being manipulated or misrepresented was lowered.

3.8 Experience gained

During the conduction of the thesis work, a problem surfaced which prevented us from continuing with the research as planned. At the start of the research, the plan was to use AWS as the cloud provider. All preparations had been made and applications had been successfully deployed to the cloud. But during the testing of our applications, it was discovered that there was a hidden limit regarding the scaling of the FaaS application.

The AWS Lambda deployment that we had started for our FaaS application included a limit on how many functions could be executed concurrently. While we could send more requests than the limit allowed, the Lambda app responded with errors to these requests instead of waiting to handle them. The structure of our tests did not handle this problem well and this resulted in the data gathered being a false representation of the cloud solutions' performance.

To combat the limits on the FaaS application with Amazon as provider, it was decided to instead conduct our research on a different provider, Azure. This was a planned measure in case something prevented us from working with AWS. All designs and requirements were kept the same, the only change was to deploy our applications to Azure's cloud services instead of Amazon's.

Chapter 4

Practical steps

In this chapter the steps conducted while constructing the applications and tests are described. It also details how the tests were run on the applications. In Section 4.1, an overview of the practical steps is given. Section 4.2 describes the process of developing the applications. Following this, Section 4.3 explains how the performance tests were constructed. Finally, Section 4.4 details the process of gathering data from the applications and tests.

4.1 Overview

The research had several practical steps that had to be completed. There were three main stages of this, which covered the practical work needed for the third and fourth phases described in Section 3.2. The first one consisted of creating the applications that were to be tested. The second part was to construct the performance tests that were to test the applications. The final part comprised executing these tests, and collecting the data needed for the comparative criteria of the study. Figure 4.1 gives an overview of these stages and their constituent parts. The two earlier stages had to be completed before the final one could be initiated.

The development of the applications and the performance tests were similar in their process. Both began with a requirement specification, followed by a design phase. With the design done, the implementation could be completed. As the applications were to be cloud applications, a final deployment phase was needed for that stage.

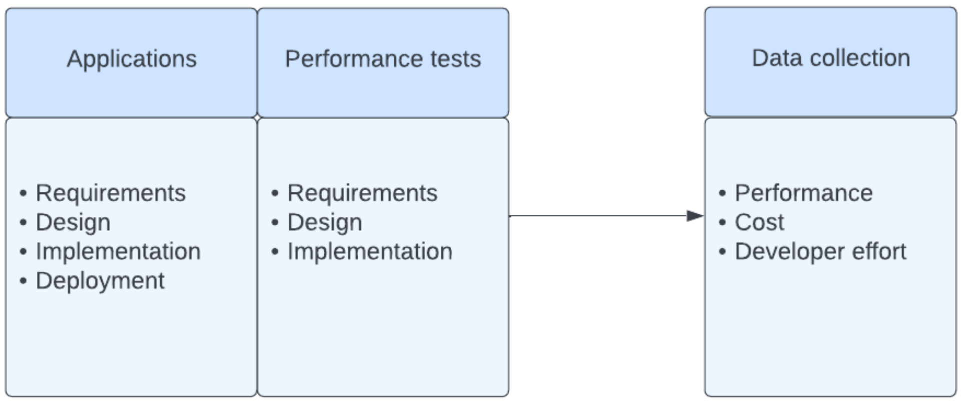


Figure 4.1: Overview of the practical steps

The data collection stage could be started when the applications and the performance tests were finished. By running the implemented performance tests on the deployed applications, the data needed for the results of this study could be collected.

4.2 Applications

This section presents the development of the applications used in this study. In Section 4.2.1, the requirements on the applications are described. Section 4.2.2 outlines the design of the applications and Section 4.2.3 explains the specific implementation of the application code. The section is concluded with Section 4.2.4 detailing the deployment process for the applications.

4.2.1 Requirement specification

This section presents the requirement specification that was written before the applications were built. Section 4.2.1.1 presents the purpose of the applications. In Section 4.2.1.2, the dependencies of the applications are explained. Section 4.2.1.3 details the functional requirements of the applications.

4.2.1.1 Purpose

This is a requirement specification for the two applications developed for this study. They are to be simple REST applications. The first one is to be developed and deployed through a IaaS solution, the second one through a FaaS one. The applications have one specific purpose, to serve as the basis for the tests performed in this study. The applications will be exposed to high loads of traffic, and the resulting data will serve as the basis for the performance and cost criteria of the comparison model. The development and implementation of the applications themselves will provide data for the developer effort criteria.

4.2.1.2 Dependencies

The applications are dependent on software during their development and execution. Most of the dependencies are common for the two applications, these are the following: *Node.js 18, Visual Studio Code, ESLint, Prettier*

Both applications use Node.js as their JavaScript runtime. Visual Studio Code is used as an Integrated Development Environment (IDE). ESLint and Prettier are used to enforce a common coding style for the two applications. Additionally, there exists some individual dependencies. For the IaaS application, this is: *Express 4.18.2*

Express is used as a framework for routing and binding HTTP methods in the IaaS application. For the FaaS application, the individual dependency is: *Azure Functions*

The Azure Functions extension for Visual Studio Code is used during the development of the FaaS application.

4.2.1.3 Functional requirements

The functional requirements are identical for both of the applications. Figure 4.2 shows a sequence diagram of the desired functionality. The client in the figure could be any HTTP client, such as a browser. Full requirement specification can be found in Appendix A.1. The application should:

- listen for HTTP requests.
- invoke a function, `getUsers()`, when a HTTP GET request to a certain endpoint is received.
- produce a response that includes a JSON array of user objects.

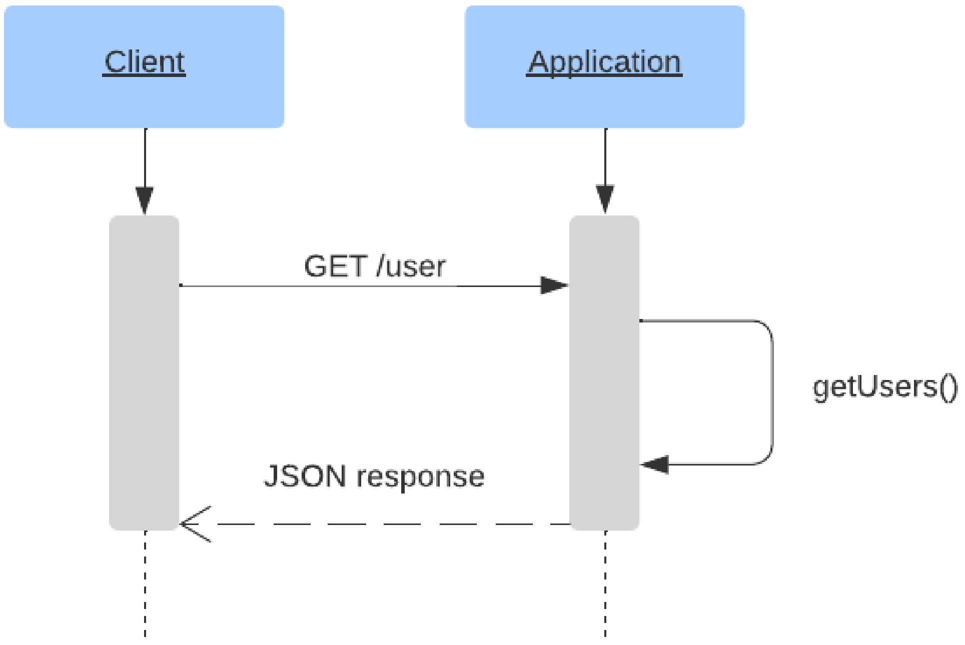


Figure 4.2: Sequence diagram illustrating the functionality of the applications

4.2.2 Design

The user objects were constructed with the goal of mimicking how a user resource could look in a production environment. It contained five fields, *id, firstName, lastName, age* and *role*. The array consisted of twenty of these objects. These were generated with an online tool, found at https://generatedata.com/. The content of this array was not significant to this study, but the size of it could influence the results.

A REST application is often used to give access to a certain resource. In a real-world setting, this resource is usually contained in a database (Neumann, Laranjeiro, & Bernardino, 2021). The choice was made to not include a database in the applications of this study. The main reason was that the goal of the study was to compare the deployment methods in their most basic form. Including a database would introduce the risk of the performance measurements being obfuscated by the time needed to communicate with the database. Instead, the resource, the array of user objects, was initialized in each function invocation and returned as such.

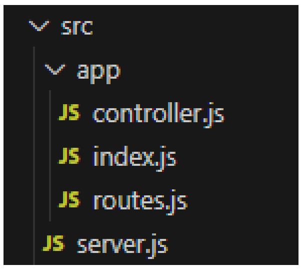


Figure 4.3: The file structure of the IaaS application

4.2.3 Implementation

This section presents the steps taken during the deployment of the two applications. Section 4.2.3.1 outlines the process of implementing the IaaS application. In Section 4.2.3.2, the implementation of the FaaS application is explained.

4.2.3.1 Infrastructure-as-a-Service

The application that was deployed to the IaaS service followed a monolithic structure. This meant that the entire program was kept in one codebase, and deployed as a single unit. As was explained in Section 3.4, the chosen runtime environment was Node. It is possible to create a RESTful API using only the built-in functionality of Node, but often a framework is used. Express is the most commonly used framework for building REST APIs in Node (Mozilla, 2023), and was therefore chosen for this application as well.

Express is an unopinionated framework regarding the structure of projects (Mozilla, 2023). As the number of lines of code was a criterion of the comparison model, the chosen structure mattered for the results of the study. Therefore, inspiration was taken from the examples provided by Express (Express, n.d.). Figure 4.3 shows the file structure of the application.

When creating a REST application with Express, certain parts need to be included. First, the application has to be configured and started so that it can accept incoming HTTP requests. Here, this was done in the `server.js` and `app/index.js` files. Second, the functions that are to be invoked when requests arrive have to be declared. The function of this application was declared in the `app/controller.js` file.

Finally, the functions need to be explicitly bound to a certain endpoint and HTTP method, done in the `app/routes.js` file. The code for the entire program is included in Appendix A.2.1.

4.2.3.2 Function-as-a-Service

The application that was deployed to FaaS was not created as a contained unit to be run on its own. Instead, the structure was highly dependent on the chosen cloud provider, Azure. The entire code base comprised an Azure Function App, which allows a user to group Azure Functions for easier management.

The process of creating the application closely resembled the process outlined in Azure's official documentation (Azure, 2023a). As the IaaS application had already been created before this, the approach taken was to use parts of it when constructing the FaaS one. Using Visual Studio Code and the accompanying Azure Functions extension, a template Azure Function App project was created.

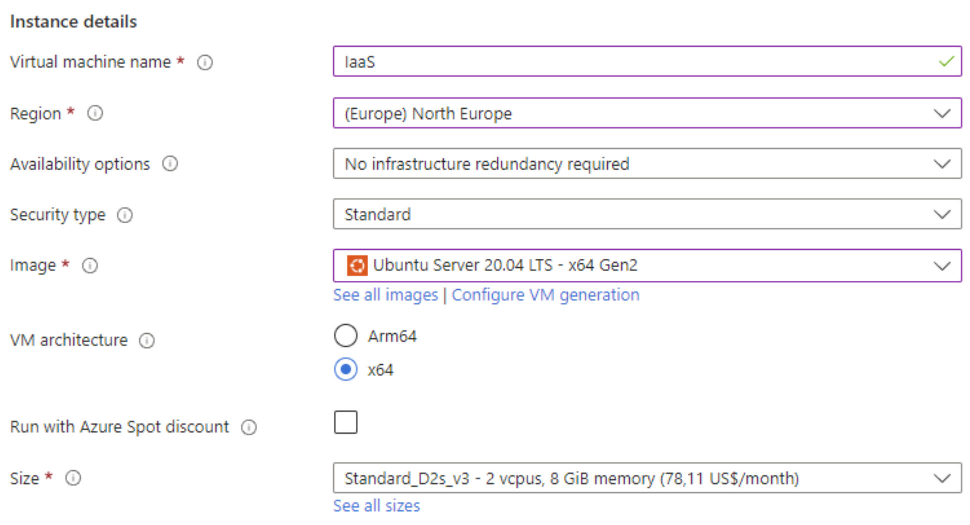
Using the *Azure Functions* extension, a new function was created. When doing this, two files were generated. The first one, `function.json`, contained the configuration for the specific function. It was altered to mimic the IaaS functionality. This included the trigger type, a HTTP request in this case, and the HTTP method that was to invoke the function. It was also possible to configure the authentication needed to invoke the function, in this case it was left open to anyone. The second file, `index.js`, contained the actual function. This code was similar to the controller function of the IaaS application.

4.2.4 Deployment

This section presents the procedure for deployment of the two created applications. Section 4.2.4.1 discusses the deployment of the IaaS application, and Section 4.2.4.2 the FaaS application.

4.2.4.1 Infrastructure-as-a-Service

The first stage of deploying the IaaS application was to provision the VM from the provider. This was done through Azure's portal page for their cloud



Instance details

Virtual machine name *	IaaS
Region *	(Europe) North Europe
Availability options	No infrastructure redundancy required
Security type	Standard
Image *	Ubuntu Server 20.04 LTS - x64 Gen2
VM architecture	○ Arm64 [✓ x64]
Run with Azure Spot discount	☐
Size *	Standard_D2s_v3 - 2 vcpus, 8 GiB memory (78,11 US$/month)

Figure 4.4: Picture of some of the chosen options for the IaaS instance

services. Here we chose the relevant cloud service for us, and then configured the settings for this service. In our case the service we needed was called *Virtual Machines*.

Multiple options for different aspects of the VM could be configured. In our case, most of these options were left on their default values, for example Linux as the operating system. Some of the options were however changed to fit our needs. First, the region in which the VM should be located was changed to North Europe. Second, the size of the system was changed. The chosen option was *Standard D2s v3* with 2 virtual cpus and 8 GiB in memory. Figure 4.4 shows a summary of the options chosen, including the size of the VM.
When the VM had been created, a few packages had to be installed before the application could be run on it. During the configuration of the machine a Secure Shell (SSH) key was created, which could then be used together with an SSH client to access the VM and download needed packages for our application.

First, we had to install the runtime environment used for our application, Node 18, on the VM. In order to transfer our application files to the VM we installed the *git* package on the machine. This allowed us to easily fetch all application files by cloning the files from our git repository. Second, when our application was downloaded on the VM, we also had to make sure that the port used for incoming requests in our application was open for communication on the VM. Therefore a new network rule had to be created using Azure's cloud portal in order to allow traffic to our application.

When the port had been opened the configuration was done. The application could be initialized and run in the deployed setting just like during local development. With the application started on the VM, it was open for communication over the internet.

4.2.4.2 Function-as-a-Service

The first part of deploying the FaaS application was to create and configure the Function App. This process closely resembled the one explained in Azure's documentation and was done solely with the use of Visual Studio Code and its extensions (Azure, 2023a).

When creating the Function App, overarching options for the entire project were selected. Some of these were important to maintain parity with the deployed IaaS application. The runtime stack was selected as Node 18, which was the latest version with long term support. The operating system was chosen as Linux and the location was set to North Europe. Finally, the hosting plan was set as Consumption. This hosting plan is Azure's fully serverless option for their functions. With it, the scaling is automatic as load increases and the billing is pay-per-use.

With the Function App created, the next step was to publish the function that existed locally to the deployed application. With the Azure extensions, the newly created Function App was reachable through Visual Studio code. The deployment of the function consisted of selecting the *Deploy* option of the local function, and specifying within which *Function App* it was to be deployed.

When a Function App is deployed, a base Uniform Resource Locator (URL) is created through Azure. Deploying functions triggered by HTTP requests to it will make them accessible on a certain path of this URL, which is the base URL followed by `/api/{functionName}`. Navigating to this address with a browser or performing a GET request in some other manner will invoke the deployed function.

4.3 Performance tests

This section presents the development of the performance tests performed in this study. In Section 4.3.1, the requirements on the performance tests are described. Section 4.3.2 outlines the design of the test cases and Section 4.3.3 explains the specific implementation of the test scripts.

4.3.1 Requirement specification

This section presents the requirement specification that was written before the performance tests were created. Section 4.3.1.1 presents the purpose of the performance tests. In Section 4.3.1.2, the dependencies of the performance tests are explained. Section 4.3.1.3 and Section 4.3.1.4 details the functional and non-functional requirements respectively.

4.3.1.1 Purpose

This is a requirement specification for the performance tests to be used in this study. The purpose of these tests are to measure the performance of the two applications. The resulting data is used for the performance and cost criteria of the comparison model.

4.3.1.2 Dependencies

The performance tests are dependent on software during their development and execution, these dependencies are: *k6* and *Visual Studio Code*.

k6 is the framework used for developing and executing the performance tests. It allows for test scripts to be written in JavaScript. Visual Studio Code is used as the IDE during development.

4.3.1.3 Functional requirements

There are functional requirements on the performance tests. Full requirement specification can be found in Appendix A.1. The tests are to:

- measure the performance of REST applications by sending multiple concurrent HTTP requests to them.
- test the performance at different load levels by varying the number of concurrent requests.

- only send HTTP GET requests.
- check the HTTP status code of the returned response.
- result in detailed reports.

4.3.1.4 Non-functional requirements

There are also a few non-functional requirements on the performance tests. The tests are to:

- be executed locally, not in a distributed manner.
- all be executed from the same computer.

4.3.2 Design

In order to gather data regarding the performance of the two applications, a few test cases had to be created. Early on we had realized that the load tests had to be carefully designed before actually performing them. The main reason being that a small mistake in different quantities could result in us incurring unnecessary costs for the FaaS implementation.

In our tests we needed to access the GET endpoint of the applications and subsequently check the status code of the responses. In order to gather information on how the applications performed under heavy load, we had to be able to send multiple requests to the endpoint at the same time. This we could achieve by using the load testing tool k6.

One of the functional requirements of the performance tests was to test the performance at at different load levels by varying the number of requests. To do this, several test cases were designed. It was decided that one of the cases was to use a load level that matched the limits of the IaaS solution, and one test case was to exceed these limits. Furthermore, two test cases were designed to use load levels below the limits. The exact load levels were decided after the deployment of the applications, as some initial testing was needed before that.

4.3.3 Implementation

The performance tests were performed using a load testing tool called k6, which included functionality for testing the performance of applications under heavy load by writing programmatic test scripts in *JavaScript*. K6 allowed the user to specify options for running test scripts. The options used in our tests were the number of virtual users that should be created and used to execute our scripts, and the number of total iterations to be run.

K6 handled the majority of the settings and configuration, and all the user needed to do was to code the function where the tests themselves were performed. All configuration of input options and output format was optional.

The code for the test cases consists of three parts, the input options, the main test function and the `handleSummary` function specifying the output format. In our case the main test function only consisted of a call to our REST endpoint and a check of the status code of the response. The options specified controlled the number of virtual users created and used to simultaneously run the main function, as well as the number of total iterations of the function performed for the load test. In our case, the number of virtual users corresponded to the number of concurrent HTTP GET requests that were to be sent to the application. The only difference in the other test cases were the option values, virtual users and iterations. The full code for all tests can be found in Appendix A.3.

As described in Section 4.3.2, four test cases with varying load levels were to be implemented. By performing some smaller tests on the deployed IaaS application, an approximate breakpoint where it produced errors was found. This was done by fully implementing the tests without deciding on the load levels. The tests were then executed, starting with a low load level. This was then successively incremented until the application started producing errors. This load level was used as one of the test cases, the other ones were implemented according to the desired load levels.

Finally, the number of virtual users for the different load levels that were decided upon were the following: *1000*, *5000*, *10000* and *15000*. The number of iterations used in the tests were decided to be five times the number of virtual users. This was chosen mainly to ensure that the duration of the tests were long enough.

4.4 Data Collection

In this section, the process of gathering the data for the criteria of the comparison model is explained. Section 4.4.1 details how the performance data was collected. The following section, 4.4.2, describes how the criteria relating to the monetary cost were handled. Finally, Section 4.4.3 explains the way in which the data of the development experience criteria was attained.

4.4.1 Performance

Running the tests explained in Section 4.3.2 produced extensive reports of the results. First, a readable summary was presented in the command line interface where the tests were executed. Additionally, our tests produced a more extensive report in a less readable format. These reports are included in JSON form in Appendix B.

Initially, the tests were run sequentially starting with the one with the lowest load. For both the IaaS and the FaaS applications, this was performed during the same day. As cold starts are a commonly quoted issue with FaaS solutions (Chowhan, 2018), the decision was made to run the FaaS tests a second time. The goal of this second test run was to get performance measurements of the FaaS application when it was experiencing the full effect of cold starts. This time, the test runs were separated by an hour. Azure states that they deallocate resources of their serverless functions after roughly 20 minutes of inactivity (Colby Tresness, 2018). An hour was chosen as the period of separation so that we could be sure that this had occurred.

From the reports, the measurements relevant to this study had to be extracted. There were two criteria pertaining to performance in our comparison model, the average time a request took, and the percentage of the requests that failed. The first one corresponded to the average *iteration_duration* measurement from the k6 reports. The second one was taken from the *checks* measurement.

To summarize, data was collected for two criteria regarding performance, response time and error percentage. This data was gathered from three complete performance tests consisting of four test cases each. One test was performed for the IaaS application and two for the FaaS application.

4.4.2 Cost

The cost criteria of the comparison was defined as the total cost of running our performance tests on the two services. Both of the two services had a type of pay-as-you-go billing policy, although some differences were present. For the IaaS solution, the billing was decided by a hourly cost for running the instance. Meanwhile, the FaaS solution's pricing was decided by the number of function calls, the time spent executing them, and the memory allocated to it during its execution.

Using the metrics available to us through Azure's portal, we could gather data regarding the consumption of our applications. For the IaaS application, the cost management service page showed the total cost for running our instance since the start, as well as the cost for each day of running our instance. Using the daily cost view, we could extract the cost data for the day that we performed the tests on, since we manually started and stopped our instance before and after the tests were performed. The documented hourly cost for the instance size chosen by us could also be found on a Azure instance pricing page (Azure, n.d.-b). This documented hourly cost was then cross referenced with the cost gathered from the cost management view.

To gather the cost of running the tests on the FaaS application, the total number of function executions and the function execution units were extracted from the metrics page. A function execution unit is the time it takes to execute a function, multiplied by the memory that is allocated to it. With the chosen pay-per-use consumption plan, the number of function executions were multiplied with a certain rate to get the first part of the price. The second one was obtained by multiplying the consumed function execution units with a different rate (Azure, n.d.-a). The total price for the user would then be the sum of these two parts.

For the Azure Functions service, there existed a free tier. With this, users get a monthly grant of a number of free function calls and function execution units before the normal billing costs apply. In these cost calculations, this free tier was disregarded and all costs were calculated as if these grants did not exist. As two rounds of tests were run on the FaaS application, it was decided to also split the incurred costs between the two tests to make the comparison with the IaaS solution more accurate.

In summary, cost data was gathered for one criteria, the total cost for the load tests. This was gathered for the three load tests conducted for the performance criteria, using the cost management pages for the two cloud services to check the costs. The end results were three measurements, one for each of the conducted load tests.

4.4.3 Developer effort

The data for the developer effort criteria was gathered by measuring the time to deploy, and the number of code lines for each of the applications. The time to deploy was defined as the time between having the application fully coded, to being fully deployed and ready to be tested. These time measurements were therefore extracted by measuring the active time spent between these two situations. In our case, both deployment phases could be finished in one continuous work session, meaning the data could be gathered without interruption.

To accurately compare these two applications, we used the code formatters Prettier and ESLint for parity between the code of them. By creating a Prettier configuration file that was identical for both applications, formatting rules could be enforced. This included attributes such as the maximum line length and the tab width. For ESLint there existed pre-defined style guides that could be imported. The choice was made to use Google's style guide for this project (Google, n.d.-a). By declaring this in the configuration file of the ESLint plugin, this style was enforced throughout the applications. These configuration files are included in Appendix A.4.

Additionally, some choices were made regarding what parts of the code to include in the results. It was decided that empty lines and the lines initializing the JSON data were to be ignored. Only the lines that we had written ourselves were included. This excluded the files that were automatically generated by frameworks and providers. For the IaaS application this included four *JavaScript* files, and for the FaaS application this included only the function declaration.

To summarize, data for two criteria was gathered regarding developer effort, time to deploy and number of code lines. The data regarding the time to deploy was gathered from the active time spent deploying the applications. And the data regarding the code lines was gathered from the created files for the two different applications. The end results were two measurements for each of the cloud solutions, one for each of the criteria.

Chapter 5

Results

In this chapter, the results of the comparative study are presented. Section 5.1 contains the results regarding the performance aspect. In Section 5.2, the results of the cost analysis are presented. Following this, in Section 5.3 the results pertaining to the developer effort are shown.

5.1 Performance

The performance results consisted of two criteria, the average response time as well as the percentage of failed requests. As mentioned in Section 4.3.3 the research results were gathered from four separate test cases all with differing load level on the applications. Table 5.1 shows the performance results for the IaaS solution. For the first test case, the average response time was fast, and all the requests were successful. The next test raised the response time significantly. The third test, which was designed to produce a load on the limits of what the IaaS solution could handle, did not lead to any failed requests but it did increase the response times notably. Finally, the last test did lead to a small amount of failed requests while further increasing the response times.

Table 5.1: Performance results for IaaS test

Load level (VUs)	**Average response time (ms)**	**Failed requests (%)**
1000	163	0
5000	722	0
10000	2120	0
15000	3620	2.78

The second test performed was on the FaaS solution. The results from this test are contained in Table 5.2. This test was run without any consideration of the concept of cold starts. Notably, all the requests were successful for all load levels. The average response time increased significantly between the test cases, reaching as high as 12.8 seconds for the final one.

Table 5.2: Performance results for FaaS test

Load level (VUs)	Average response time (ms)	Failed requests (%)
1000	960	0
5000	459	0
10000	7960	0
15000	12 800	0

Finally a third test was performed, this also on the FaaS solution. During this test the concept of cold starts was considered and the execution of the test cases was adapted to it. The full procedure for running this test was explained in Section 4.4.1. The results from this third test can be found in Table 5.3. The lowest load did not result in any failed requests, but it did come with a high response time. The following test increased the response time significantly, while also causing a significant number of requests to fail. The final two tests both lead to similar response times, reaching more than 21 seconds. They both lead to a high amount of failed requests, especially the last one which had more than 36% of them fail.

Table 5.3: Performance results for FaaS test with cold starts

Load level (VUs)	Average response time (ms)	Failed requests (%)
1000	6230	0
5000	14 800	8.88
10000	21 200	14.8
15000	21 600	36.1

5.2 Cost

The cost results consisted of one measurement for each of the three performed tests. These measurements represented the total incurred monetary cost for running each of the tests. As was explained in Section 4.4.2, the process of producing these measurements differed between the IaaS and FaaS solutions. For the IaaS solution, the incurred monetary cost could be directly extracted from Azure's metrics. This was not the case for the FaaS tests, where some calculations had to be made. Table 5.4 contains the usage data that could be extracted from Azure's metrics and used to calculate the resulting cost.

Table 5.4: Usage data for the tests

Test	**Function executions**	**Function execution units (GB-s)**
FaaS (1st test)	155 000	91 109
FaaS (2nd test)	118 349	67 812

From Azure's documentation, the rates corresponding to the usage data could be found. These were \$0.000016/GB-s for the function execution units and \$0.20 per million function executions (Azure, n.d.-a). For the first test the cost for the function executions was $155000 * 0.20/1000000 = \$0.031$, for the function execution units it was $91109 * 0.000016 = \$1,458$. The resulting cost was the sum of these two, that is $\$0.031 + \$1,458 = \$1.49$. For the second test, the cost for the function executions was $118349 * 0.20/1000000 = \$0.024$ and for the function execution units it was $67812 * 0.000016 = \$1,085$. This lead to the resulting cost being $\$0.024 + \$1,085 = \$1.11$.

Table 5.5 shows the cost of running our tests for both the IaaS and FaaS solution. The table shows that the cost of the IaaS solution ended up being a few times lower than the FaaS ones. The two tests performed on the FaaS solution also differed in cost, where the test adapted to cold starts resulted in a lower cost than the other.

Table 5.5: Cost results for the performed tests

Test	**Cost (\$)**
IaaS	0.45
FaaS (1st test)	1.49
FaaS (2nd test)	1.11

5.3 Developer effort

The results concerning the developer effort consisted of two measured criteria, the time it took to deploy the solutions and the lines of code that were needed. Table 5.6 contains the results for these two criteria. The lines of code were lower for the FaaS solution, being able to achieve the desired functionality with five lines. For the IaaS solution, twenty lines of code were used for the same functionality. The time it took to get the finished local version of the IaaS solution fully deployed was 120 minutes. For the FaaS solution, this time was 10 minutes.

Table 5.6: Developer effort results

Type of service	**Lines of code**	**Time to deploy (min)**
IaaS	20	120
FaaS	5	10

Chapter 6

Analysis and discussion

In this chapter, the results presented in Chapter 5 are analyzed and discussed. In Sections 6.1, 6.2 and 6.3 the performance, cost and developer effort results are discussed and analyzed respectively. The chapter ends with Section 6.4 presenting how we combated the validity threats and analyzing the validity of our results.

6.1 Performance

The results for the performance measurements clearly show that the IaaS solution had shorter average response times for all the tests. This was contrary to our expectations that the FaaS would respond quicker. For the first round of tests of the FaaS solution, presented in Table 5.2, the average response time reached above ten seconds for the highest load. These tests were run without taking the concept of cold starts into consideration. Some initial trial tests had already been performed on the application before the actual testing took place. This meant that the first test with a low load level was not performed on an application which had had its resources completely deallocated. The following tests were run sequentially, so the resources from the previous test were still allocated when a new one was performed. Consequently, the results in Table 5.2 show the results for an active application that is subjected to a steadily increasing load.

The second round of tests of the FaaS application were constructed to maximize the effects of cold starts. This was done by making sure that the application was in an idle state before every individual test. The results in Table 5.3 show a dramatic increase in response times compared to the previous round of tests. A common scenario which is affected by cold starts is a sudden spike in function calls, which can lead to noticeable overhead in their execution time (Chowhan, 2018). The high response times of this third test indicate that this is what occurred.

Regarding the amount of failed requests, the IaaS application only showed these for the highest load level. This was contrary to what our initial testing had showed, where the previous level had produced failed requests as well. Cloud services tend to show a varying performance, with daily fluctuations being common (Iosup, Yigitbasi, & Epema, 2011). It is possible that this deviation was caused by such a variation.

For the FaaS application, the first test did not produce any failed requests. This result illustrates one of the fundamental differences between the two solutions, their scalability. As the IaaS solution reached its limits for the higher load tests, it would need to be scaled to accommodate more traffic. Scaling it could be done in a horizontal manner, by adding more VMs, or in a vertical one, by increasing the capacity of the existing machine. Both of these lead to a step wise scaling pattern where the capacity of the application increases in discrete stages (Chowhan, 2018). As the FaaS can scale in a granular way, on a per-function-execution basis, a limit is not reached when the load is increased gradually.

The second test on the FaaS solution produced a high amount of failed requests. When Azure scales their functions up, it is done by adding more running instances of the *Function Application*. This is handled by a *scale controller*. There exists an upper limit on how quickly this can happen, the *new instance rate* (Azure, 2023b). It is probable that the load pattern of the second test brought the application to this limit, as each test demanded that it scaled from an idle state to the desired load. This limit on the scalability of the FaaS application was not something that had been predicted by us. The result shows that the hands-off approach for scaling a serverless solution can bring other issues, as the manner in which it is scaled and the limitations on it are, to a greater extent, controlled by the provider.

The results concerning the performance of the two solutions show that the IaaS one had a faster response time for all the test cases. This resembles the findings of Malla and Christensen, 2019, which also found that an IaaS solution could be up to 1.65 times faster than a FaaS one. The workload used in that study differs greatly from the one used here, focusing on a smaller amount of concurrent requests with more demanding computations. With this, the effects of cold starts for the FaaS solution are more prominent in this study, as many more containers needed to be initialized in a short amount of time. This was the case for both of the tests performed on the FaaS application, but the effect was understandably higher in the second one.

6.2 Cost

While inspecting the cost results, presented in Table 5.5, we can clearly deduce that the IaaS solution had a significantly lower cost than the FaaS solution of around three times as much. While a FaaS solution often is considered the cheaper solution between these two, with the high loads the applications experienced during our tests the fact that the actual cost of the solutions is as presented is not surprising. The IaaS pricing entails paying a set price per day of running an instance, contrasted by the FaaS solution requiring payment depending on the number of requests and execution time of your functions. This means that the load on the IaaS application has no effects on the resulting costs while the FaaS application's cost increases depending on the load.

The hypothesis regarding this criteria, presented in Section 3.4.2.2, was also consistent with the results gathered from our tests. The main reason why the outcome was predictable was that the situations of the tests were, from the cost perspective, favorable for the IaaS solution. This is because the tests were performed during a short time, meaning a minimal uptime for the instances, in turn resulting in low costs for the IaaS solution. The tests also meant that the applications were bombarded with many requests resulting in the FaaS solution increasing in price.

The second test of the FaaS solution lead to a lower cost than the first one. This further highlights one of the key differences between the two applications, that one only pays for the actual function executions for a FaaS deployment. As the second test produced many failed requests, these never reached the stage where they invoked the function and therefore never incurred a charge for the user.

The results concerning the costs of the two solutions show that the IaaS solution had a cost of less than half of the FaaS solution. These findings do not correspond to the results of previous research, Malla and Christensen, 2019;

Villamizar et al., 2016, which both found that the FaaS solution had a lower cost than the IaaS solution. The reason for the deviation of our results could be that the test performed by us included many requests. This resulting in many function invocations for the FaaS solution. As mentioned earlier, this meant that the cost of the FaaS solution increased for each function invocation and the time spent executing that function, while the cost of the IaaS stayed the same regardless of the number of requests.

6.3 Developer effort

The results regarding the developer effort for the two solutions, presented in Section 5.3, clearly show that the FaaS solution is a better candidate in this aspect. The table shows us that the IaaS application had four times as many lines of code than the FaaS application, which matches with our hypothesis discussed in Section 3.4.2.3. The fact that the FaaS application is so much shorter is not surprising when the main part of the application is automatically generated configuration files, which we did not count for the lines of code criteria. Also, the IaaS application includes a lot of boilerplate code required for the application to be complete, which is especially apparent when the functionality of the application is as simple as ours was.

The second criteria for developer effort, time to deploy, also shows that FaaS had a more than ten times shorter time to deploy. That the IaaS application would result in a longer time to deploy was expected by us and apparent in our hypothesis as well, also presented in Section 3.4.2.3. But the difference was not expected to be on this scale, rather we were expecting the IaaS application to take a few times longer than the FaaS application, not as much as ten times.

During the deployment of the IaaS application, time had to be spent on many stages of deployment such as configuration, SSH key handling, dependency installation, network configuration, and more. This led to the deployment taking much longer than we anticipated. In contrast the deployment of the FaaS application was much simpler, only including two steps, creating the function application and deploying the functions to it. The fact that the FaaS solution had plugins available for the IDE used during development, Visual Studio Code, also resulted in an easier and quicker deployment process.

It is important to consider that the results for both criteria in this category are highly specific to the context of this study. First, the number of lines of code was greatly influenced by the specific structure and functionality of the applications. Second, the time it took to deploy the applications was dependent on the previous experience of us as developers. Finally, the concept of developer experience is broad, and this study only focused on a small part of it, which we named developer effort (Fagerholm & Münch, 2012). They can not be used to draw far-reaching conclusions about the general developer experience of the two offerings. What they can do is provide insight into how the experience was like for us, as novices in the field of cloud computing, when working with them.

6.4 Validity analysis

In this section, the validity threats and how they were met, is analyzed. It covers the threats presented in Section 3.6 by first dealing with the credibility in Section 6.4.1. Following this, the transferability is analyzed in Section 6.4.2. In the next one, Section 6.4.3, the dependability is handled. Finally, in Section 6.4.4, the confirmability of the research is considered.

6.4.1 Credibility

The research of this thesis aimed to attain a high level of credibility by using three strategies. First, the methods used to attain the underlying data and the metrics of the data were based on previous research. This was especially true for the performance data, which was attained through the use of a widely used tool. The metrics of the developer effort aspect were an exception to this, as previous research that measured this could not be found. The credibility of the conclusions drawn from these measurements were therefore lower.

Second, a thorough description of the phenomenon being studied was included. Section 3.2.2 and 3.4.2 contains clear definitions of what the concepts of performance, cost and developer effort meant in the context of this thesis.

Finally, the results of the comparative study were compared to results from similar previous research in Sections 6.1, 6.2 and 6.3. A high consistency with previous research gave the results of this study a higher credibility.

6.4.2 Transferability

To combat the threat to the study's transferability, a strategy mentioned by Guba, 1981 was employed, to develop a thick description of the context. In Chapter 4 the applications, deployments and load tests were explained in detail. With this, a reader could get a clear understanding of the context and make judgements about the fittingness with other contexts. Additionally, the choice was made to use the established REST paradigm for the applications of the study to increase the transferability of the results.

6.4.3 Dependability

In the case of this thesis, some of the phenomena that were studied were changing due to factors outside of our control, such as the cloud services performance varying due to the time of day. Factors like this were mitigated by performing the tests with this in mind. Additionally, the subjective nature of some of the criteria in the comparison contributed to a lower repeatability.

Two strategies mentioned by Shenton, 2004 were employed to increase the dependability. First, the research design and implementation were thoroughly described in Chapter 3. Second, the operational detail of the gathering of data was clearly explained in Section 4.4. Together, these steps enabled the research to be repeated in a similar manner, although not necessarily with the same results.

6.4.4 Confirmability

Guba, 1981 mentions several strategies that can be employed to ensure a high confirmability, two of these were used in this thesis' study.

The first one was to include reflective commentary by revealing the predispositions that lead to what results were gathered. Section 3.4.2 included the motivations for the criteria of the comparative study and what was expected from them. In Chapter 6, the preliminary theories were included, even in the case where the results did not support them.

Second, an attempt was made to include an audit trail. This was done by including all the gathered performance data in Appendix B, and Chapter 6 included clear explanations of how this data led to the conclusions that were drawn.

Chapter 7

Conclusions and Future work

Cloud computing is a collection of services used by many companies and individuals today, and with the extensive usage of cloud computing comes many considerations of what type of service to use. These complicated decisions require comparisons of the different types of cloud services but the problem is that we lack up to date comparisons of FaaS and traditional IaaS solutions. The purpose of this thesis is therefore to provide such a comparison of Azure's cloud solutions. While the goal of this thesis is to provide information for future cloud choices and research. To achieve this goal an extensive literature study was conducted, as well as measurements of the cloud options' performance, cost and developer effort. The result of this thesis is a comparison of Azure's FaaS and IaaS implementations.

7.1 Conclusions

The practical part of the research began with a literature study. Previous research was consulted and used as a basis for deciding how the thesis' comparison was to be performed. Three factors of the existing research served as a motivation for this investigation. First, most of it was several years old, in a field that moves quickly. The performance and pricing of the cloud services change rapidly. Second, most of it focused on other solutions than the serverless options. Finally, in a field where the choice of cloud provider is so important, no previous comparisons of Azure's serverless option with their traditional IaaS one could be found. Therefore it could be concluded early in the investigation that the main problem was the lack of up-to-date comparisons of Azure's serverless FaaS solution to their IaaS one.

The comparison dealt with three main aspects of the solutions, their performance, cost, and the developer effort required to develop them. By subjecting the solutions to heavy loads, and measuring their response time and fault rate, their performance could be established. The cost aspect was analyzed by looking at the incurred monetary costs for the solutions, while executing the performance tests. The developer effort involved with the two options was compared by looking at the amount of code needed to achieve the desired functionality, and the time it took to deploy them. The conclusion was drawn that this approach could be used to achieve the purpose of comparing the serverless solution to the IaaS one.

The results concerning the performance of the two solutions showed that the IaaS one performed better. It resulted in significantly lower response times for each test case. The amount of failed requests was also low for the IaaS one, only appearing for the highest load level. This highlighted one key difference between the two services, the scalability of the FaaS one. As the amount of concurrent requests became high enough, the IaaS solution reached its limit while the FaaS one could scale up and handle the high load.

The high response times of the FaaS solution during the first test highlighted a different aspect of its scalability. Although all of the incoming requests could be fulfilled, it took a long time. It was theorized that this was due to the concept of cold starts. When the application was subjected to a high number of concurrent requests, new function containers needed to be initialized. The rate at which this could happen seemed to be a limiting factor.

To further investigate this, a second test was performed on the FaaS application. It was designed to maximize this effect by letting the application reach an idle state between each test case. The results of the second test lead to even higher response times and many failed requests. It showed that although the FaaS had the ability to automatically scale up, this ability was limited. When subjected to high enough load levels, it was not able to scale up sufficiently fast.

The results relating to the monetary cost of the two cloud services showed that the IaaS one was cheaper. Both tests performed on the FaaS solution incurred a higher cost than it. The fact that the cost of running the two tests on the FaaS application were different, highlighted the pay-per-use pricing model of it. Only the executed function calls lead to a monetary cost. The failed requests of the second test, that did not invoke the function, had no associated cost. The cost of running the tests on the IaaS application did not depend on the number of requests, only on the time it was deployed. The traffic generated with the performance tests consisted of a high amount of requests, arriving during a short time period. It could be concluded that the pricing model of the IaaS service was favourable for this type of traffic.

The results highlighting the developer effort of the two cloud services showed that the FaaS solution produced significantly lower values than the IaaS one, both for number of code lines and time to deploy. One of the main reasons for investigating this criteria was the alleged benefits of the serverless solutions. Looking at the results gathered from our study we can conclude that these claims held true. The authors of this study had no prior experience working with serverless cloud solutions, and even then the development and deployment procedures were easy to follow and perform. It is however important to remember that the results regarding developer effort are highly specific to the context of this study, meaning that they are hard to generalize and transfer to other contexts.

The purpose of this study was to compare Azure's FaaS offering with their IaaS one, focusing on their performance, cost and developer effort. This was achieved, as the produced results covered all of these aspects. The main goal of this thesis was to help software professionals in choosing a cloud service. This goal was achieved by analyzing the results and formulating the following guidelines:

- For the types of traffic used in this study, Azure's FaaS offering can not be recommended above their IaaS one if the response times are of value.
- For applications with a short deployment window and a high amount of traffic, Azure's IaaS option is cheaper and therefore recommended.
- The comparatively low effort involved with developing a FaaS application through Azure is an advantage. If these aspects are valued, the FaaS solution is highly recommended.

7.2 Future work

Even though the comparison performed in this thesis can be used as indication of how cloud services compare to each other, there are still many things that warrant future work. Most of the limitations placed upon this thesis can be further researched in future work. The results of the comparative study performed in this thesis is limited to one cloud provider, Azure, and limited to two of their cloud offerings, IaaS and FaaS. This is one part that can warrant future work concerned with different cloud providers and different cloud offerings.

The programs created for the comparative study is another part that can be further researched. The programs was limited in their functionality and only represented one type of workload. This could be built upon in other research, where more advanced programs could be used. Different workloads could be used for more in-depth testing of the implementations of the cloud provider's services.

The performed comparative study only concerns itself with three aspects of the cloud solutions, performance, cost and developer effort. There are many more aspects of the cloud services that can be explored and evaluated. One of the biggest being the security of the cloud solutions, which can greatly affect the choice of cloud services in the industry. Another aspect that can be explored are the effects on the performance of the cloud services when they are accessed from different parts of the globe, testing how well the solutions can be distributed across larger areas.

References

AWS. (n.d.). *Fast NoSQL key-value database – amazon DynamoDB – amazon web services* [Amazon web services, inc.]. Retrieved April 10, 2023, from https://aws.amazon.com/dynamodb/

Azure. (n.d.-a). *Pricing - functions | microsoft azure*. Retrieved May 7, 2023, from https://azure.microsoft.com/en-us/pricing/details/functions/

Azure. (n.d.-b). *Pricing - linux virtual machines | microsoft azure*. Retrieved May 7, 2023, from https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/

Azure. (n.d.-c). *Pricing overview—how azure pricing works | microsoft azure*. Retrieved April 7, 2023, from https://azure.microsoft.com/en-us/pricing/

Azure. (2023a, February 14). *Develop azure functions by using visual studio code*. Retrieved May 7, 2023, from https://learn.microsoft.com/en-us/azure/azure-functions/functions-develop-vs-code

Azure. (2023b, April 7). *Event-driven scaling in azure functions*. Retrieved May 9, 2023, from https://learn.microsoft.com/en-us/azure/azure-functions/event-driven-scaling

Bhandari, P. (2020a, June 12). *What is quantitative research? | definition, uses & methods* [Scribbr]. Retrieved April 24, 2023, from https://www.scribbr.com/methodology/quantitative-research/

Bhandari, P. (2020b, June 19). *What is qualitative research? | methods & examples* [Scribbr]. Retrieved April 24, 2023, from https://www.scribbr.com/methodology/qualitative-research/

Chowhan, K. (2018). *Hands-on serverless computing: Build, run, and orchestrate serverless applications using AWS lambda, microsoft azure functions, and google cloud functions*. Packt Publishing.

Colby Tresness. (2018, July 2). *Understanding serverless cold start | Azure-bloggen och uppdateringar | Microsoft Azure*. Retrieved May 7, 2023, from https://azure.microsoft.com/sv-se/blog/understanding-serverless-cold-start/

Dahal, P., & Prasai, R. (2022, July 12). *Representative service providers and their selection in cloud computing domain; a comprehensive overview*. https://doi.org/10.20944/preprints202207.0190.v1

Express. (n.d.). *Express examples*. Retrieved May 7, 2023, from https://expressjs.com/en/starter/examples.html

Fagerholm, F., & Münch, J. (2012). Developer experience: Concept and definition. *2012 International Conference on Software and System Process (ICSSP)*, 73–77. https://doi.org/10.1109/ICSSP.2012.6225984

Fisher, C. (2018). Cloud versus on-premise computing. *American Journal of Industrial and Business Management*, *08*(9), 1991–2006. https://doi.org/10.4236/ajibm.2018.89133

Galante, G., & de Bona, L. C. E. (2012). A survey on cloud computing elasticity. *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, 263–270. https://doi.org/10.1109/UCC.2012.30

Gartner. (n.d.). *Gartner forecasts worldwide public cloud end-user spending to reach nearly $500 billion in 2022* [Gartner]. Retrieved March 28, 2023, from https://www.gartner.com/en/newsroom/press-releases/2022-04-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-500-billion-in-2022

Google. (n.d.-a). *Google JavaScript style guide*. Retrieved May 19, 2023, from https://google.github.io/styleguide/jsguide.html

Google. (n.d.-b). *Pricing overview* [Google cloud]. Retrieved April 7, 2023, from https://cloud.google.com/pricing

Guba, E. G. (1981). ERIC/ECTJ annual review paper: Criteria for assessing the trustworthiness of naturalistic inquiries [Publisher: Springer]. *Educational Communication and Technology*, *29*(2), 75–91. Retrieved April 20, 2023, from https://www.jstor.org/stable/30219811

HPE. (n.d.). *What is on-premises vs. cloud? | glossary*. Retrieved April 5, 2023, from https://www.hpe.com/us/en/what-is/on-premises-vs-cloud.html

IBM. (n.d.). *IaaS vs. PaaS vs. SaaS | IBM*. Retrieved March 28, 2023, from https://www.ibm.com/topics/iaas-paas-saas

Iosup, A., Yigitbasi, N., & Epema, D. (2011). On the performance variability of production cloud services, 104–113. https://doi.org/10.1109/CCGrid.2011.22

Jääskeläinen, P. (2019). *Comparing cloud architectures in terms of performance and scalability*. Retrieved March 28, 2023, from http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-254615

Kang, E., & Hwang, H.-J. (2021). Ethical conducts in qualitative research methodology :participant observation and interview process. *Journal of Research and Publication Ethics*, *2*(2), 5–10. https://doi.org/10.15722/JRPE.2.2.202109.5

Kaplunovich, A. (2019). ToLambda–automatic path to serverless architectures. *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, 1–8. https://doi.org/10.1109/IWoR.2019.00008

Kranz, J. J., Hanelt, A., & Kolbe, L. M. (2016). Understanding the influence of absorptive capacity and ambidexterity on the process of business model change - the case of on-premise and cloud-computing software: Understanding the dynamics of business model change. *Information Systems Journal*, *26*(5), 477–517. https://doi.org/10.1111/isj.12102

Lupia, A., & Elman, C. (2014). Openness in political science: Data access and research transparency: Introduction [Publisher: Cambridge University Press]. *PS: Political Science & Politics*, *47*(1), 19–42. https://doi.org/10.1017/S1049096513001716

Malla, S., & Christensen, K. (2019). HPC in the cloud: Performance comparison of function as a service (FaaS) vs infrastructure as a service (IaaS). *Internet Technology Letters*, *3*, e137. https://doi.org/10.1002/itl2.137

Marin, E., Perino, D., & Di Pietro, R. (2022, January 27). Serverless computing: A security perspective. Retrieved March 28, 2023, from http://arxiv.org/abs/2107.03832

Millnert, V., & Eker, J. (2020). HoloScale: Horizontal and vertical scaling of cloud resources. *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, 196–205. https://doi.org/10.1109/UCC48980.2020.00038

Miri, S. M., & Dehdashti Shahrokh, Z. (2019). A short introduction to comparative research.

Mozilla. (2023, February 24). *Express/node introduction - learn web development | MDN*. Retrieved May 7, 2023, from https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction

Neumann, A., Laranjeiro, N., & Bernardino, J. (2021). An analysis of public REST web service APIs. *IEEE Transactions on Services Computing*, *14*(4), 957–970. https://doi.org/10.1109/TSC.2018.2847344

Newman, S. (2021). *Building microservices: Designing fine-grained systems* (Second edition).

Pautasso, C., Wilde, E., & Alarcón, R. (Eds.). (2014). *REST: Advanced research topics and practical applications* [OCLC: ocn857849618]. Springer.

Roberts, M. (2018, May 22). *Serverless architectures* [Martinfowler.com]. Retrieved April 7, 2023, from https://martinfowler.com/articles/serverless.html

Sbarski, P., Cui, Y., & Nair, A. (2022). *Serverless architectures on AWS* (Second edition). Manning Publications Co.

Shenton, A. (2004). Strategies for ensuring trustworthiness in qualitative research projects. *Education for Information*, *22*, 63–75. https://doi.org/10.3233/EFI-2004-22201

Taherdoost, H. (2016). Sampling methods in research methodology; how to choose a sampling technique for research. *International Journal of Academic Research in Management*, *5*, 18–27. https://doi.org/10.2139/ssrn.3205035

Team Cleo. (n.d.). *Blog: On premise vs. cloud: Key differences, benefits and risks* [Cleo]. Retrieved April 7, 2023, from https://www.cleo.com/blog/knowledge-base-on-premise-vs-cloud

Tim, H. W., & Rana, M. E. (2022). A review of cloud computing on sustainable development: Contribution, exploration and potential challenges. *2022 International Conference on Data Analytics for Business and Industry (ICDABI)*, 176–183. https://doi.org/10.1109/ICDABI56818.2022.10041482

Villamizar, M., Garces, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A., & Lang, M. (2016). Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 179–182. https://doi.org/10.1109/CCGrid.2016.37

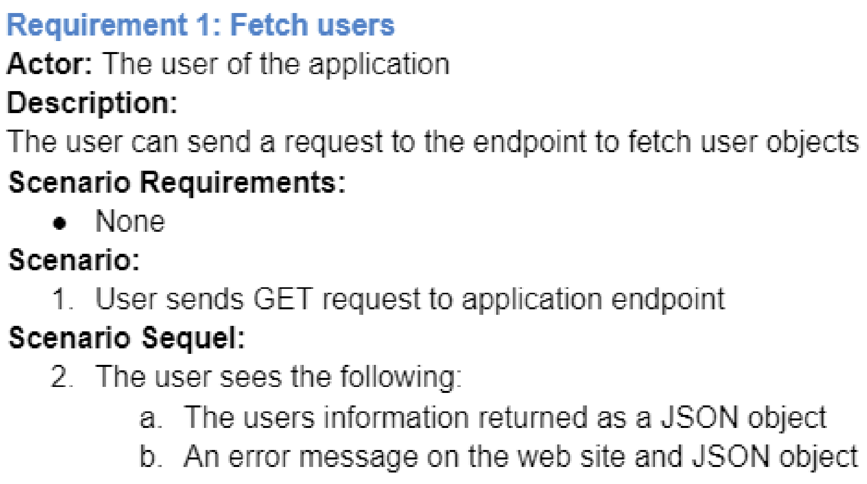
Appendix A

Requirements and Code

This appendix includes the requirements and code for the developed applications, performance tests and configuration files.

A.1 Requirement specifications

This section includes the requirement specifications for the applications and the performance tests.



Requirement 1: Fetch users
Actor: The user of the application
Description:
The user can send a request to the endpoint to fetch user objects
Scenario Requirements:

- None

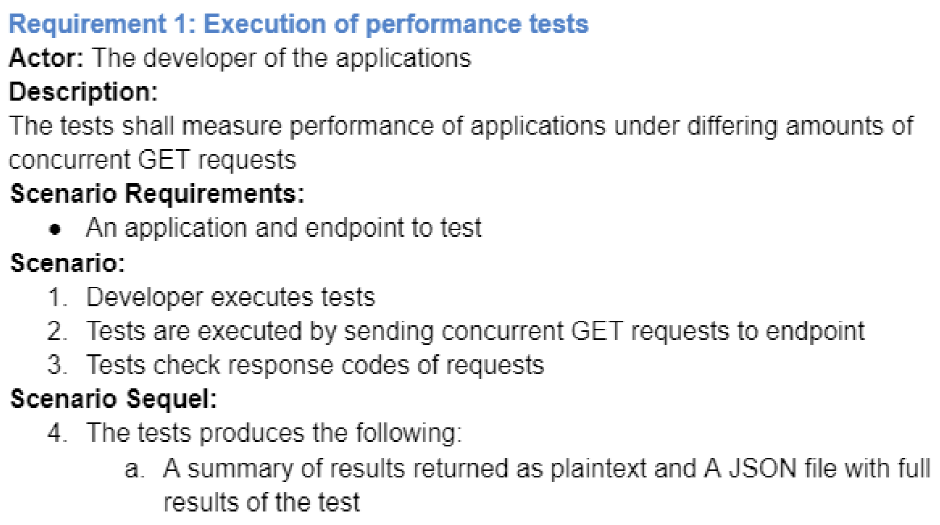
Scenario:

1. User sends GET request to application endpoint

Scenario Sequel:

2. The user sees the following:
    a. The users information returned as a JSON object
    b. An error message on the web site and JSON object

Figure A.1: Requirement specification for the applications



Requirement 1: Execution of performance tests
Actor: The developer of the applications
Description:
The tests shall measure performance of applications under differing amounts of concurrent GET requests
Scenario Requirements:
- An application and endpoint to test

Scenario:
1. Developer executes tests
2. Tests are executed by sending concurrent GET requests to endpoint
3. Tests check response codes of requests

Scenario Sequel:
4. The tests produces the following:
    a. A summary of results returned as plaintext and A JSON file with full results of the test

Figure A.2: Requirement specification for the performance tests

A.2 Application code

This section includes the code for the two created applications.

A.2.1 Infrastructure-as-a-service code

This section includes the code for the IaaS application.

Listing A.1: Code from `server.js`

```
const app = require('./app');

const PORT = process.env.PORT || 5001;

app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Listing A.2: Code from `app/index.js`

```
const express = require('express');
const app = express();
const routes = require('./routes');

app.use(express.json());

app.use(routes);

module.exports = app;
```

Listing A.3: Code from `app/controller.js`

```
const getUsers = (req, res) => {
  const data = [
    {
      id: '1',
      firstName: 'Kevyn',
      lastName: 'Freeman',
      age: '49',
      role: 'Media Relations',
    },
    /*
    19 additional user resources of the same format
    are included here.
    */
  ];

  res.send(data);
};

module.exports = { getUsers };
```

Listing A.4: Code from `app/routes.js`

```
const express = require('express');
const router = express.Router();
const { getUsers } = require('./controller');

router.get('/user', getUsers);

module.exports = router;
```

A.2.2 Function-as-a-service code

This section includes the code for the FaaS application.

Listing A.5: Code from the `getUsers()` function

```
module.exports = async function (context, req) {
  const data = [
    {
      id: '1',
      firstName: 'Kevyn',
      lastName: 'Freeman',
      age: '49',
      role: 'Media Relations',
    },
    /*
    19 additional user resources of the same format
    are included here.
    */
  ];

  context.res = {
    body: JSON.stringify(data),
  };
};
```

A.3 Performance tests

This section includes the code for the performance tests. The only difference between the test for IaaS and FaaS is the url accessed in the `http.get()` function, therefore only one of each test case has been included.

Listing A.6: Code from `low.js`

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { textSummary } from
'https://jslib.k6.io/k6-summary/0.0.2/index.js';

export const options = {
  vus: 1000,
  iterations: 5000,
  discardResponseBodies: true,
};
```

```

export default function () {
  const res = http.get(/*URL*/);
  sleep(1);
  check(res, {
    'status was 200': (r) => r.status == 200,
  });
}

export function handleSummary(data) {
  return {
    stdout: textSummary(data,
  { indent: ' ', enableColors: true }),
    'summary-low.json': JSON.stringify(data),
  };
}
```

Listing A.7: Code from `mid.js`

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { textSummary } from
'https://jslib.k6.io/k6-summary/0.0.2/index.js';

export const options = {
  vus: 5000,
  iterations: 25000,
  discardResponseBodies: true,
};

export default function () {
  const res = http.get(/*URL*/);
  sleep(1);
  check(res, {
    'status was 200': (r) => r.status == 200,
  });
}

export function handleSummary(data) {
  return {
    stdout: textSummary(data,
  { indent: ' ', enableColors: true }),
    'summary-mid.json': JSON.stringify(data),
  };
}
```

Listing A.8: Code from `high.js`

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { textSummary } from
'https://jslib.k6.io/k6-summary/0.0.2/index.js';

export const options = {
  vus: 10000,
  iterations: 50000,
  discardResponseBodies: true,
};

export default function () {
  const res = http.get(/*URL*/);
  sleep(1);
  check(res, {
    'status was 200': (r) => r.status == 200,
  });
}

export function handleSummary(data) {
  return {
    stdout: textSummary(data,
  { indent: ' ', enableColors: true }),
    'summary-high.json': JSON.stringify(data),
  };
}
```

Listing A.9: Code from `extreme.js`

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { textSummary } from
'https://jslib.k6.io/k6-summary/0.0.2/index.js';
```

```

export const options = {
  vus: 15000,
  iterations: 75000,
  discardResponseBodies: true,
};

export default function () {
  const res = http.get(/*URL*/);
  sleep(1);
  check(res, {
    'status was 200': (r) => r.status == 200,
  });
}

export function handleSummary(data) {
  return {
    stdout: textSummary(data,
  { indent: ' ', enableColors: true }),
    'summary-extreme.json': JSON.stringify(data),
  };
}
```

A.4 Configuration files

This section includes the configuration code for the code formatters.

A.4.1 ESLint configuration

Listing A.10: Code from `.eslintrc.js`

```
module.exports = {
  env: {
    commonjs: true,
    es6: true,
    node: true,
  },
  extends: [
    'plugin:node/recommended',
    'eslint-config-prettier',
    'eslint:recommended',
    'google'
  ],
  parserOptions: {
    ecmaVersion: 2020,
  },
  rules: {},
};
```

Listing A.11: Code from `.prettierrc.cjs`

```
module.exports = {
  trailingComma: 'all',
  tabWidth: 2,
  semi: true,
  singleQuote: true,
  printWidth: 100,
  bracketSpacing: true,
};
```

Appendix B

JSON reports

This appendix includes the JSON reports from the performed performance tests. Each of the listings represents one test case.

B.1 Infrastructure-as-a-service

This section includes the JSON reports for the test performed on the IaaS application.

Listing B.1: Report from `summary-low.json`

```
{
  "root_group": {
    "groups": [],
    "checks": [
      {
        "name": "status was 200",
        "path": "::status was 200",
        "id": "1461660757a913d4fb82ac4c5e1009de",
        "passes": 5000,
        "fails": 0
      }
    ],
    "name": "",
    "path": "",
    "id": "d41d8cd98f00b204e9800998ecf8427e"
  },
  "options": {
    "summaryTrendStats": ["avg", "min", "med", "max",
       ↪ "p(90)", "p(95)"],
    "summaryTimeUnit": "",
    "noColor": false
  },
  "state": { "isStdOutTTY": true, "isStdErrTTY": true,
     ↪ "testRunDurationMs": 6093.8253 },
  "metrics": {
    "data_sent": {
      "type": "counter",
      "contains": "data",
      "values": { "count": 445000, "rate": 73024
         ↪ .73866456264 }
    },
    "checks": {
      "contains": "default",
      "values": { "rate": 1, "passes": 5000, "fails": 0
         ↪ },
      "type": "rate"
    },
    "http_req_blocked": {
      "values": {
        "p(90)": 89.6286,
        "p(95)": 99.964535,
        "avg": 18.503060059999996,
        "min": 0,
        "med": 0,
        "max": 135.5765
      },
      "type": "trend",
      "contains": "time"
    },
    "vus_max": {
      "type": "gauge",
      "contains": "default",
      "values": { "value": 1000, "min": 1000, "max": 100
         ↪ 0 }
    },
    "http_req_failed": {
      "type": "rate",
      "contains": "default",
    "values": { "rate": 0, "passes": 0, "fails": 5000
        ↪ }
  },
  "http_req_waiting": {
    "type": "trend",
    "contains": "time",
    "values": {
      "avg": 140.36816729999953,
      "min": 53.1853,
      "med": 80.7299,
      "max": 516.4187,
      "p(90)": 387.8138300000006,
      "p(95)": 479.67321000000015
    }
  },
  "http_req_receiving": {
    "values": {
      "p(90)": 0,
      "p(95)": 0.5201150000000001,
      "avg": 0.05705918000000005,
      "min": 0,
      "med": 0,
      "max": 1.2735
    },
    "type": "trend",
    "contains": "time"
  },
  "iterations": {
    "type": "counter",
    "contains": "default",
    "values": { "count": 5000, "rate": 820
        ↪ .5026816242993 }
  },
  "http_req_tls_handshaking": {
    "type": "trend",
    "contains": "time",
    "values": { "med": 0, "max": 0, "p(90)": 0,
        ↪ "p(95)": 0, "avg": 0, "min": 0 }
  },
  "http_req_duration{expected_response:true}": {
    "contains": "time",
    "values": {
        "min": 53.1853,
        "med": 80.78545,
        "max": 516.8827,
        "p(90)": 387.8674900000008,
        "p(95)": 479.67321000000015,
        "avg": 140.4825427399994
      },
      "type": "trend"
    },
    "http_req_sending": {
      "type": "trend",
      "contains": "time",
      "values": {
        "avg": 0.05731626,
        "min": 0,
        "med": 0,
        "max": 4.3367,
        "p(90)": 0,
        "p(95)": 0.5181
      }
    },
    "http_reqs": {
      "type": "counter",
      "contains": "default",
      "values": { "count": 5000, "rate": 820
         ↪ .5026816242993 }
    },
    "http_req_connecting": {
      "type": "trend",
      "contains": "time",
      "values": {
        "min": 0,
        "med": 0,
        "max": 124.7499,
        "p(90)": 89.3796,
        "p(95)": 99.8007,
        "avg": 18.44318145999999
      }
    },
    "iteration_duration": {
      "contains": "time",
      "values": {
        "min": 1054.4327,
        "med": 1085.5115,
        "max": 1642.0851,
        "p(90)": 1478.31917,
        "p(95)": 1572.3136100000002,
        "avg": 1163.4842260800033
      },
      "type": "trend"
    },
    "http_req_duration": {
      "type": "trend",
      "contains": "time",
      "values": {
        "max": 516.8827,
        "p(90)": 387.8674900000008,
        "p(95)": 479.67321000000015,
        "avg": 140.4825427399994,
        "min": 53.1853,
        "med": 80.78545
      }
    },
    "vus": {
      "type": "gauge",
      "contains": "default",
      "values": { "max": 1000, "value": 813, "min": 813
         ↪ }
    },
    "data_received": {
      "type": "counter",
      "contains": "data",
      "values": { "count": 9925000, "rate": 1628697
         ↪ .823024234 }
    }
  }
}
```

Listing B.2: Report from `summary-mid.json`

```
{
  "metrics": {
    "http_req_blocked": {
      "type": "trend",
    "contains": "time",
    "values": {
      "min": 0,
      "med": 0,
      "max": 3155.441,
      "p(90)": 295.2844,
      "p(95)": 415.591,
      "avg": 87.21730280800013
    }
  },
  "data_sent": {
    "type": "counter",
    "contains": "data",
    "values": { "rate": 232673.36848231437, "count": 2
       ↪ 225000 }
  },
  "data_received": {
    "type": "counter",
    "contains": "data",
    "values": { "count": 49625000, "rate": 5189400
       ↪ .409408921 }
  },
  "http_req_waiting": {
    "type": "trend",
    "contains": "time",
    "values": {
      "p(95)": 2100.9078,
      "avg": 630.9935573679952,
      "min": 239.1769,
      "med": 451.7557,
      "max": 5653.034,
      "p(90)": 998.5320200000002
    }
  },
  "http_req_tls_handshaking": {
    "type": "trend",
    "contains": "time",
    "values": { "avg": 0, "min": 0, "med": 0, "max": 0
       ↪ , "p(90)": 0, "p(95)": 0 }
  },
  "http_req_sending": {
    "contains": "time",
    "values": {
      "avg": 0.09411943999999943,
      "min": 0,
      "med": 0,
      "max": 28.009,
      "p(90)": 0,
      "p(95)": 0.9984
    },
    "type": "trend"
  },
  "http_req_receiving": {
    "contains": "time",
    "values": {
      "avg": 0.04499121999999989,
      "min": 0,
      "med": 0,
      "max": 3.0013,
      "p(90)": 0,
      "p(95)": 0.5065
    },
    "type": "trend"
  },
  "iteration_duration": {
    "type": "trend",
    "contains": "time",
    "values": {
      "min": 1298.3158,
      "med": 1457.4780500000002,
      "max": 7075.9589,
      "p(90)": 2389.055330000001,
      "p(95)": 3430.6613,
      "avg": 1721.727530547958
    }
  },
  "vus": {
    "type": "gauge",
    "contains": "default",
    "values": { "value": 2965, "min": 0, "max": 5000 }
  },
  "http_reqs": {
    "type": "counter",
    "contains": "default",
    "values": { "count": 25000, "rate": 2614
        ↪ .30751103724 }
  },
  "iterations": {
    "type": "counter",
    "contains": "default",
    "values": { "count": 25000, "rate": 2614
        ↪ .30751103724 }
  },
  "vus_max": {
    "type": "gauge",
    "contains": "default",
    "values": { "value": 5000, "min": 4182, "max": 500
        ↪ 0 }
  },
  "http_req_duration": {
    "type": "trend",
    "contains": "time",
    "values": {
      "med": 451.76545,
      "max": 5653.034,
      "p(90)": 998.5320200000002,
      "p(95)": 2100.9314999999997,
      "avg": 631.132668027992,
      "min": 246.1862
    }
  },
  "http_req_failed": {
    "type": "rate",
    "contains": "default",
    "values": { "rate": 0, "passes": 0, "fails": 25000
        ↪  }
  },
  "http_req_connecting": {
    "type": "trend",
    "contains": "time",
    "values": {
      "p(90)": 294.8,
      "p(95)": 415.29263,
      "avg": 87.12672747200014,
      "min": 0,
      "med": 0,
        "max": 3104.6131
      }
    },
    "http_req_duration{expected_response:true}": {
      "type": "trend",
      "contains": "time",
      "values": {
        "avg": 631.132668027992,
        "min": 246.1862,
        "med": 451.76545,
        "max": 5653.034,
        "p(90)": 998.5320200000002,
        "p(95)": 2100.9314999999997
      }
    },
    "checks": {
      "type": "rate",
      "contains": "default",
      "values": { "rate": 1, "passes": 25000, "fails": 0
         ↪  }
    }
  },
  "root_group": {
    "name": "",
    "path": "",
    "id": "d41d8cd98f00b204e9800998ecf8427e",
    "groups": [],
    "checks": [
      {
        "name": "status was 200",
        "path": "::status was 200",
        "id": "1461660757a913d4fb82ac4c5e1009de",
        "passes": 25000,
        "fails": 0
      }
    ]
  },
  "options": {
    "summaryTrendStats": ["avg", "min", "med", "max",
      ↪ "p(90)", "p(95)"],
    "summaryTimeUnit": "",
    "noColor": false
  },
  "state": { "isStdOutTTY": true, "isStdErrTTY": true,
     ↪ "testRunDurationMs": 9562.7618 }
}
```

Listing B.3: Report from `summary-high.json`

```
{
  "root_group": {
    "checks": [
      {
        "passes": 50000,
        "fails": 0,
        "name": "status was 200",
        "path": "::status was 200",
        "id": "1461660757a913d4fb82ac4c5e1009de"
      }
    ],
    "name": "",
    "path": "",
    "id": "d41d8cd98f00b204e9800998ecf8427e",
    "groups": []
  },
  "options": {
    "summaryTrendStats": ["avg", "min", "med", "max",
       ↪ "p(90)", "p(95)"],
    "summaryTimeUnit": "",
    "noColor": false
  },
  "state": { "isStdOutTTY": true, "isStdErrTTY": true,
     ↪ "testRunDurationMs": 17547.929 },
  "metrics": {
    "data_received": {
      "values": { "count": 99250000, "rate": 5655938
         ↪ .08819263 },
      "type": "counter",
      "contains": "data"
    },
    "http_req_connecting": {
      "type": "trend",
      "contains": "time",
      "values": {
        "max": 15109.2146,
        "p(90)": 510.6856,
        "p(95)": 1439.9875649999997,
        "avg": 233.5562314479987,
        "min": 0,
        "med": 0
      }
    },
    "http_req_failed": {
      "type": "rate",
      "contains": "default",
      "values": { "fails": 50000, "rate": 0, "passes": 0
         ↪   }
    },
    "iterations": {
      "type": "counter",
      "contains": "default",
      "values": { "rate": 2849.339087250695, "count": 50
         ↪ 000 }
    },
    "vus": {
      "type": "gauge",
      "contains": "default",
      "values": { "min": 0, "max": 10000, "value": 1298
         ↪ }
    },
    "iteration_duration": {
      "type": "trend",
      "contains": "time",
      "values": {
        "p(90)": 3544.7563299999997,
        "p(95)": 16142.145189999974,
        "avg": 3122.1618649599714,
        "min": 1507.6385,
        "med": 1660.1537,
        "max": 16973.1723
      }
    },
    "http_req_duration{expected_response:true}": {
      "type": "trend",
      "contains": "time",
      "values": {
        "max": 15433.3075,
        "p(90)": 2078.8187,
        "p(95)": 13486.369065,
        "avg": 1885.4504456559973,
        "min": 342.1129,
        "med": 650.6042
      }
    },
    "http_req_waiting": {
      "type": "trend",
      "contains": "time",
      "values": {
        "avg": 1884.9894708059937,
        "min": 341.1161,
        "med": 650.5754,
        "max": 15433.2551,
        "p(90)": 2077.8430399999997,
        "p(95)": 13486.369065
      }
    },
    "http_req_blocked": {
      "type": "trend",
      "contains": "time",
      "values": {
        "max": 15109.2146,
        "p(90)": 513.6863,
        "p(95)": 1440.2071699999988,
        "avg": 233.6996455359989,
        "min": 0,
        "med": 0
      }
    },
    "data_sent": {
      "type": "counter",
      "contains": "data",
      "values": { "count": 4450000, "rate": 253591
          ↪ .17876531184 }
    },
    "vus_max": {
      "type": "gauge",
      "contains": "default",
      "values": { "value": 10000, "min": 4157, "max": 10
```

```
     ↪ 000 }
},
"http_reqs": {
  "contains": "default",
  "values": { "count": 50000, "rate": 2849
     ↪ .339087250695 },
  "type": "counter"
},
"http_req_sending": {
  "type": "trend",
  "contains": "time",
  "values": {
    "avg": 0.41714854000000073,
    "min": 0,
    "med": 0,
    "max": 268.0342,
    "p(90)": 0,
    "p(95)": 0.9994
  }
},
"http_req_tls_handshaking": {
  "contains": "time",
  "values": { "avg": 0, "min": 0, "med": 0, "max": 0
     ↪ , "p(90)": 0, "p(95)": 0 },
  "type": "trend"
},
"checks": {
  "type": "rate",
  "contains": "default",
  "values": { "passes": 50000, "fails": 0, "rate": 1
     ↪  }
},
"http_req_duration": {
  "contains": "time",
  "values": {
    "p(90)": 2078.8187,
    "p(95)": 13486.369065,
    "avg": 1885.4504456559973,
    "min": 342.1129,
    "med": 650.6042,
    "max": 15433.3075
  },
      "type": "trend"
    },
    "http_req_receiving": {
      "type": "trend",
      "contains": "time",
      "values": {
        "avg": 0.0438263099999999,
        "min": 0,
        "med": 0,
        "max": 4.0002,
        "p(90)": 0,
        "p(95)": 0.5065
      }
    }
  }
}
```

Listing B.4: Report from `summary-extreme.json`

```
{
  "state": { "isStdOutTTY": true, "isStdErrTTY": true,
     ↪ "testRunDurationMs": 62133.8654 },
  "metrics": {
    "vus_max": {
      "values": { "min": 4095, "max": 15000, "value": 15
         ↪ 000 },
      "type": "gauge",
      "contains": "default"
    },
    "checks": {
      "type": "rate",
      "contains": "default",
      "values": { "rate": 0.97224, "passes": 72918,
         ↪ "fails": 2082 }
    },
    "data_sent": {
      "values": { "count": 6551023, "rate": 105434
         ↪ .01666428434 },
      "type": "counter",
      "contains": "data"
    },
    "http_reqs": {
```

```
    "type": "counter",
    "contains": "default",
    "values": { "count": 75000, "rate": 1207
        ↪ .071208545799 }
  },
  "http_req_sending": {
    "contains": "time",
    "values": {
      "max": 37.0087,
      "p(90)": 0,
      "p(95)": 0.5394,
      "avg": 0.08330681199999987,
      "min": 0,
      "med": 0
    },
    "type": "trend"
  },
  "http_req_receiving": {
    "contains": "time",
    "values": {
      "avg": 0.040763317333333355,
      "min": 0,
      "med": 0,
      "max": 4.9991,
      "p(90)": 0,
      "p(95)": 0.5049
    },
    "type": "trend"
  },
  "iterations": {
    "type": "counter",
    "contains": "default",
    "values": { "count": 75000, "rate": 1207
        ↪ .071208545799 }
  },
  "http_req_connecting": {
    "type": "trend",
    "contains": "time",
    "values": {
      "avg": 317.2866820413354,
      "min": 0,
      "med": 0,
        "max": 15213.2045,
        "p(90)": 645.4086,
        "p(95)": 1576.6838500000015
      }
    },
    "data_received": {
      "contains": "data",
      "values": { "count": 144742230, "rate": 2329522
         ↪ .3799161864 },
      "type": "counter"
    },
    "http_req_blocked": {
      "type": "trend",
      "contains": "time",
      "values": {
        "min": 0,
        "med": 0,
        "max": 15213.2045,
        "p(90)": 646.2619300000007,
        "p(95)": 1577.1833450000004,
        "avg": 317.3651776533357
      }
    },
    "iteration_duration": {
      "type": "trend",
      "contains": "time",
      "values": {
        "max": 61197.416,
        "p(90)": 5872.8919,
        "p(95)": 21927.156925000003,
        "avg": 4622.157962073321,
        "min": 1196.2001,
        "med": 2404.4231499999996
      }
    },
    "http_req_duration{expected_response:true}": {
      "type": "trend",
      "contains": "time",
      "values": {
        "max": 29981.84,
        "p(90)": 2491.8648600000006,
        "p(95)": 13924.772224999999,
        "avg": 2461.2217018774713,
        "min": 194.1971,
        "med": 1391.6203500000001
      }
    },
    "http_req_failed": {
      "type": "rate",
      "contains": "default",
      "values": { "fails": 72918, "rate": 0.02776,
          ↪ "passes": 2082 }
    },
    "vus": {
      "type": "gauge",
      "contains": "default",
      "values": { "value": 589, "min": 0, "max": 15000 }
    },
    "http_req_tls_handshaking": {
      "type": "trend",
      "contains": "time",
      "values": { "min": 0, "med": 0, "max": 0,
          ↪ "p(90)": 0, "p(95)": 0, "avg": 0 }
    },
    "http_req_waiting": {
      "type": "trend",
      "contains": "time",
      "values": {
        "avg": 2896.212860426673,
        "min": 0,
        "med": 1390.3688,
        "max": 59662.5722,
        "p(90)": 2527.1993,
        "p(95)": 17215.9394
      }
    },
    "http_req_duration": {
      "type": "trend",
      "contains": "time",
      "values": {
        "max": 59663.5743,
        "p(90)": 2527.207540000001,
        "p(95)": 17215.9394,
        "avg": 2896.3369305560254,
        "min": 0,
        "med": 1390.3869
      }
    }
  },
  "root_group": {
    "name": "",
    "path": "",
    "id": "d41d8cd98f00b204e9800998ecf8427e",
    "groups": [],
    "checks": [
      {
        "name": "status was 200",
        "path": "::status was 200",
        "id": "1461660757a913d4fb82ac4c5e1009de",
        "passes": 72918,
        "fails": 2082
      }
    ]
  },
  "options": {
    "summaryTrendStats": ["avg", "min", "med", "max",
       ↪ "p(90)", "p(95)"],
    "summaryTimeUnit": "",
    "noColor": false
  }
}
```

B.2 Function-as-a-service first test

This section includes the JSON reports for the first test performed on the FaaS application.

Listing B.5: Report from `summary-low.json`

```
{
  "state": { "isStdOutTTY": true, "isStdErrTTY": true,
     ↪ "testRunDurationMs": 10720.1865 },
  "metrics": {
    "http_req_failed": {
      "type": "rate",
      "contains": "default",
    "values": { "passes": 0, "fails": 5000, "rate": 0
        ↪ }
  },
  "http_reqs": {
    "values": { "count": 5000, "rate": 466
        ↪ .40979613554293 },
    "type": "counter",
    "contains": "default"
  },
  "http_req_tls_handshaking": {
    "type": "trend",
    "contains": "time",
    "values": {
      "avg": 127.08198816000021,
      "min": 0,
      "med": 0,
      "max": 920.6003,
      "p(90)": 571.46721,
      "p(95)": 731.9477800000002
    }
  },
  "data_received": {
    "type": "counter",
    "contains": "data",
    "values": { "count": 16220685, "rate": 1513097
        ↪ .276805772 }
  },
  "http_req_sending": {
    "type": "trend",
    "contains": "time",
    "values": {
      "min": 0,
      "med": 0,
      "max": 19.2526,
      "p(90)": 0,
      "p(95)": 0.7377850000000051,
      "avg": 0.11336897999999983
    }
  },
  "vus": {
    "type": "gauge",
    "contains": "default",
    "values": { "value": 547, "min": 547, "max": 1000
       ↪ }
  },
  "checks": {
    "values": { "rate": 1, "passes": 5000, "fails": 0
       ↪ },
    "type": "rate",
    "contains": "default"
  },
  "data_sent": {
    "values": { "count": 1109000, "rate": 103449
       ↪ .69278286342 },
    "type": "counter",
    "contains": "data"
  },
  "vus_max": {
    "values": { "value": 1000, "min": 1000, "max": 100
       ↪ 0 },
    "type": "gauge",
    "contains": "default"
  },
  "http_req_connecting": {
    "type": "trend",
    "contains": "time",
    "values": {
      "p(95)": 134.4036,
      "avg": 25.75818146000001,
      "min": 0,
      "med": 0,
      "max": 1072.1544,
      "p(90)": 114.6592
    }
  },
  "http_req_duration{expected_response:true}": {
    "type": "trend",
    "contains": "time",
    "values": {
      "min": 122.612,
      "med": 729.2302,
      "max": 2134.1172,
      "p(90)": 1247.2387700000002,
      "p(95)": 1407.7396350000001,
        "avg": 786.7311432599987
      }
    },
    "http_req_waiting": {
      "contains": "time",
      "values": {
        "med": 726.4905,
        "max": 2134.1172,
        "p(90)": 1243.0506800000003,
        "p(95)": 1401.5792450000001,
        "avg": 783.3184245399988,
        "min": 122.612
      },
      "type": "trend"
    },
    "http_req_receiving": {
      "contains": "time",
      "values": {
        "avg": 3.2993497400000007,
        "min": 0,
        "med": 0,
        "max": 190.5748,
        "p(90)": 9.169960000000003,
        "p(95)": 15.864950000000004
      },
      "type": "trend"
    },
    "http_req_duration": {
      "type": "trend",
      "contains": "time",
      "values": {
        "p(95)": 1407.7396350000001,
        "avg": 786.7311432599987,
        "min": 122.612,
        "med": 729.2302,
        "max": 2134.1172,
        "p(90)": 1247.2387700000002
      }
    },
    "iteration_duration": {
      "type": "trend",
      "contains": "time",
      "values": {
        "p(95)": 3316.2812,
        "avg": 1960.455812779994,
        "min": 1129.4309,
        "med": 1749.61295,
        "max": 4142.0479,
        "p(90)": 3021.78902
      }
    },
    "iterations": {
      "values": { "count": 5000, "rate": 466
        ↪ .40979613554293 },
      "type": "counter",
      "contains": "default"
    },
    "http_req_blocked": {
      "type": "trend",
      "contains": "time",
      "values": {
        "avg": 169.50883707999975,
        "min": 0,
        "med": 0,
        "max": 1303.0391,
        "p(90)": 801.8560800000008,
        "p(95)": 944.2359200000001
      }
    }
  },
  "root_group": {
    "path": "",
    "id": "d41d8cd98f00b204e9800998ecf8427e",
    "groups": [],
    "checks": [
      {
        "fails": 0,
        "name": "status was 200",
        "path": "::status was 200",
        "id": "1461660757a913d4fb82ac4c5e1009de",
        "passes": 5000
      }
    ],
    "name": ""
  },
  "options": {
    "summaryTrendStats": ["avg", "min", "med", "max",
       ↪ "p(90)", "p(95)"],
    "summaryTimeUnit": "",
    "noColor": false
  }
}
```

Listing B.6: Report from `summary-mid.json`

```
{
  "root_group": {
    "groups": [],
    "checks": [
      {
        "id": "1461660757a913d4fb82ac4c5e1009de",
        "passes": 25000,
        "fails": 0,
        "name": "status was 200",
        "path": "::status was 200"
      }
    ],
    "name": "",
    "path": "",
    "id": "d41d8cd98f00b204e9800998ecf8427e"
  },
  "options": {
    "summaryTrendStats": ["avg", "min", "med", "max",
       ↪ "p(90)", "p(95)"],
    "summaryTimeUnit": "",
    "noColor": false
  },
  "state": { "isStdOutTTY": true, "isStdErrTTY": true,
     ↪ "testRunDurationMs": 30162.0013 },
  "metrics": {
    "http_reqs": {
      "type": "counter",
      "contains": "default",
      "values": { "count": 25000, "rate": 828
         ↪ .8574670938696 }
    },
"checks": {
  "type": "rate",
  "contains": "default",
  "values": { "rate": 1, "passes": 25000, "fails": 0
     ↪  }
},
"data_sent": {
  "type": "counter",
  "contains": "data",
  "values": { "count": 5545000, "rate": 183840
     ↪ .58620142026 }
},
"iteration_duration": {
  "type": "trend",
  "contains": "time",
  "values": {
    "p(90)": 7849.175690000003,
    "p(95)": 9456.181449999998,
    "avg": 5588.025128936025,
    "min": 1556.4811,
    "med": 5393.17705,
    "max": 14250.42
  }
},
"http_req_receiving": {
  "type": "trend",
  "contains": "time",
  "values": {
    "p(95)": 20.362689999999994,
    "avg": 4.000139684000021,
    "min": 0,
    "med": 0,
    "max": 315.1398,
    "p(90)": 11.213410000000076
  }
},
"vus": {
  "type": "gauge",
  "contains": "default",
  "values": { "value": 42, "min": 0, "max": 5000 }
},
"http_req_duration{expected_response:true}": {
    "values": {
      "avg": 3874.7407081079914,
      "min": 553.1214,
      "med": 4023.5836,
      "max": 8885.7618,
      "p(90)": 5208.236130000001,
      "p(95)": 5517.816445
    },
    "type": "trend",
    "contains": "time"
  },
  "http_req_connecting": {
    "type": "trend",
    "contains": "time",
    "values": {
      "p(95)": 437.04913,
      "avg": 76.07368014800005,
      "min": 0,
      "med": 0,
      "max": 3908.8349,
      "p(90)": 320.3707
    }
  },
  "http_req_blocked": {
    "contains": "time",
    "values": {
      "min": 0,
      "med": 0,
      "max": 7511.5783,
      "p(90)": 3645.681630000001,
      "p(95)": 4303.92373,
      "avg": 709.8268970400012
    },
    "type": "trend"
  },
  "http_req_tls_handshaking": {
    "contains": "time",
    "values": {
      "avg": 625.8205384999999,
      "min": 0,
      "med": 0,
      "max": 7143.7628,
        "p(90)": 3124.7244500000056,
        "p(95)": 3824.402025
      },
      "type": "trend"
    },
    "http_req_failed": {
      "type": "rate",
      "contains": "default",
      "values": { "rate": 0, "passes": 0, "fails": 25000
         ↪  }
    },
    "http_req_duration": {
      "type": "trend",
      "contains": "time",
      "values": {
        "p(90)": 5208.236130000001,
        "p(95)": 5517.816445,
        "avg": 3874.7407081079914,
        "min": 553.1214,
        "med": 4023.5836,
        "max": 8885.7618
      }
    },
    "vus_max": {
      "type": "gauge",
      "contains": "default",
      "values": { "value": 5000, "min": 4902, "max": 500
         ↪ 0 }
    },
    "data_received": {
      "type": "counter",
      "contains": "data",
      "values": { "count": 81098455, "rate": 2688762
         ↪ .3998610466 }
    },
    "http_req_waiting": {
      "type": "trend",
      "contains": "time",
      "values": {
        "min": 552.658,
        "med": 4018.90885,
        "max": 8884.6794,
        "p(90)": 5205.214840000001,
        "p(95)": 5515.036405,
        "avg": 3870.169297728028
      }
    },
    "http_req_sending": {
      "type": "trend",
      "contains": "time",
      "values": {
        "avg": 0.5712706959999989,
        "min": 0,
        "med": 0,
        "max": 83.5399,
        "p(90)": 0.9914,
        "p(95)": 3.6553
      }
    },
    "iterations": {
      "type": "counter",
      "contains": "default",
      "values": { "count": 25000, "rate": 828
          ↪ .8574670938696 }
    }
  }
}
```

Listing B.7: Report from `summary-high.json`

```
{
  "root_group": {
    "groups": [],
    "checks": [
      {
        "name": "status was 200",
        "path": "::status was 200",
        "id": "1461660757a913d4fb82ac4c5e1009de",
        "passes": 50000,
        "fails": 0
      }
    ],
    "name": "",
    "path": "",
    "id": "d41d8cd98f00b204e9800998ecf8427e"
  },
  "options": {
    "summaryTrendStats": ["avg", "min", "med", "max",
       ↪ "p(90)", "p(95)"],
    "summaryTimeUnit": "",
    "noColor": false
  },
  "state": { "isStdOutTTY": true, "isStdErrTTY": true,
     ↪ "testRunDurationMs": 48500.0885 },
  "metrics": {
    "vus": {
      "contains": "default",
      "values": { "value": 200, "min": 0, "max": 10000 }
         ↪ ,
      "type": "gauge"
    },
    "http_reqs": {
      "type": "counter",
      "contains": "default",
      "values": { "count": 50000, "rate": 1030
         ↪ .925953877383 }
    },
    "iteration_duration": {
      "type": "trend",
      "contains": "time",
      "values": {
        "avg": 8958.709428105909,
        "min": 1654.0299,
        "med": 8486.665649999999,
        "max": 25095.0167,
        "p(90)": 13099.022099999998,
        "p(95)": 15991.933489999992
      }
    },
    "http_req_waiting": {
      "values": {
        "avg": 6642.9558768919815,
        "min": 644.7198,
        "med": 7005.00695,
        "max": 17678.0705,
        "p(90)": 8607.7685,
    "p(95)": 8989.769465
  },
  "type": "trend",
  "contains": "time"
},
"vus_max": {
  "type": "gauge",
  "contains": "default",
  "values": { "value": 10000, "min": 4776, "max": 10
     ↪ 000 }
},
"checks": {
  "type": "rate",
  "contains": "default",
  "values": { "rate": 1, "passes": 50000, "fails": 0
     ↪  }
},
"http_req_sending": {
  "type": "trend",
  "contains": "time",
  "values": {
    "p(90)": 0.9995,
    "p(95)": 3.0683499999999952,
    "avg": 0.4952737179999956,
    "min": 0,
    "med": 0,
    "max": 39.3392
  }
},
"http_req_receiving": {
  "type": "trend",
  "contains": "time",
  "values": {
    "p(95)": 18.905839999999987,
    "avg": 3.648183946000002,
    "min": 0,
    "med": 0,
    "max": 834.7309,
    "p(90)": 8.99
  }
},
"http_req_failed": {
    "type": "rate",
    "contains": "default",
    "values": { "rate": 0, "passes": 0, "fails": 50000
       ↪  }
  },
  "http_req_tls_handshaking": {
    "type": "trend",
    "contains": "time",
    "values": {
      "p(90)": 4436.211219999999,
      "p(95)": 6394.673144999999,
      "avg": 993.0642930740025,
      "min": 0,
      "med": 0,
      "max": 12713.3075
    }
  },
  "http_req_connecting": {
    "type": "trend",
    "contains": "time",
    "values": {
      "max": 15106.4563,
      "p(90)": 666.2860099999999,
      "p(95)": 2413.0297549999977,
      "avg": 309.3775218579995,
      "min": 0,
      "med": 0
    }
  },
  "data_received": {
    "type": "counter",
    "contains": "data",
    "values": { "count": 162202393, "rate": 3344373
       ↪ .134494384 }
  },
  "iterations": {
    "contains": "default",
    "values": { "count": 50000, "rate": 1030
       ↪ .925953877383 },
    "type": "counter"
  },
  "http_req_duration": {
    "type": "trend",
    "contains": "time",
    "values": {
      "med": 7008.6891,
      "max": 17678.9518,
      "p(90)": 8612.13736,
      "p(95)": 8995.08048,
      "avg": 6647.0993345559855,
      "min": 651.4214
    }
  },
  "http_req_duration{expected_response:true}": {
    "type": "trend",
    "contains": "time",
    "values": {
      "max": 17678.9518,
      "p(90)": 8612.13736,
      "p(95)": 8995.08048,
      "avg": 6647.0993345559855,
      "min": 651.4214,
      "med": 7008.6891
    }
  },
  "http_req_blocked": {
    "values": {
      "p(90)": 6886.973019999999,
      "p(95)": 8210.805934999998,
      "avg": 1308.2230632439944,
      "min": 0,
      "med": 0,
      "max": 15391.4889
    },
    "type": "trend",
    "contains": "time"
  },
  "data_sent": {
    "contains": "data",
    "values": { "count": 11090000, "rate": 228659
      ↪ .3765700036 },
    "type": "counter"
  }
}
```

```
}
```

Listing B.8: Report from `summary-extreme.json`

```
{
  "root_group": {
    "name": "",
    "path": "",
    "id": "d41d8cd98f00b204e9800998ecf8427e",
    "groups": [],
    "checks": [
      {
        "name": "status was 200",
        "path": "::status was 200",
        "id": "1461660757a913d4fb82ac4c5e1009de",
        "passes": 75000,
        "fails": 0
      }
    ]
  },
  "options": {
    "summaryTrendStats": ["avg", "min", "med", "max",
       ↪ "p(90)", "p(95)"],
    "summaryTimeUnit": "",
    "noColor": false
  },
  "state": { "isStdOutTTY": true, "isStdErrTTY": true,
     ↪ "testRunDurationMs": 74364.8665 },
  "metrics": {
    "iterations": {
      "type": "counter",
      "contains": "default",
      "values": { "count": 75000, "rate": 1008
         ↪ .5407737536918 }
    },
    "http_req_duration": {
      "type": "trend",
      "contains": "time",
      "values": {
        "avg": 10431.003687946655,
        "min": 902.0804,
        "med": 11135.5738,
        "max": 19249.3024,
        "p(90)": 14009.72365,
        "p(95)": 14872.63292
      }
    },
    "vus": {
      "type": "gauge",
      "contains": "default",
      "values": { "value": 39, "min": 0, "max": 15000 }
    },
    "vus_max": {
      "type": "gauge",
      "contains": "default",
      "values": { "value": 15000, "min": 4420, "max": 15
         ↪ 000 }
    },
    "http_req_duration{expected_response:true}": {
      "contains": "time",
      "values": {
        "p(95)": 14872.63292,
        "avg": 10431.003687946655,
        "min": 902.0804,
        "med": 11135.5738,
        "max": 19249.3024,
        "p(90)": 14009.72365
      },
      "type": "trend"
    },
    "http_req_receiving": {
      "type": "trend",
      "contains": "time",
      "values": {
        "avg": 3.7990345093333326,
        "min": 0,
        "med": 0,
        "max": 544.7778,
        "p(90)": 9.0547,
        "p(95)": 19.349405
      }
    },
    "data_received": {
      "values": { "count": 243296439, "rate": 3271658
```

```
        ↪ .3845410384 },
      "type": "counter",
      "contains": "data"
    },
    "http_req_blocked": {
      "type": "trend",
      "contains": "time",
      "values": {
        "max": 33346.7815,
        "p(90)": 12696.352000000003,
        "p(95)": 13689.240355,
        "avg": 2352.709745714659,
        "min": 0,
        "med": 0
      }
    },
    "iteration_duration": {
      "type": "trend",
      "contains": "time",
      "values": {
        "avg": 13787.192940438692,
        "min": 1904.2482,
        "med": 13257.98765,
        "max": 49944.7998,
        "p(90)": 19760.62324000001,
        "p(95)": 24909.140295
      }
    },
    "checks": {
      "type": "rate",
      "contains": "default",
      "values": { "rate": 1, "passes": 75000, "fails": 0
        ↪  }
    },
    "data_sent": {
      "type": "counter",
      "contains": "data",
      "values": { "rate": 223694.34361856885, "count": 1
        ↪ 6635000 }
    },
    "http_req_tls_handshaking": {
      "type": "trend",
    "contains": "time",
    "values": {
      "p(90)": 8054.17485,
      "p(95)": 10270.062275,
      "avg": 1626.794902075999,
      "min": 0,
      "med": 0,
      "max": 31864.6095
    }
  },
  "http_req_connecting": {
    "type": "trend",
    "contains": "time",
    "values": {
      "min": 0,
      "med": 0,
      "max": 15911.2243,
      "p(90)": 1382.4961,
      "p(95)": 6024.064080000003,
      "avg": 716.757614048004
    }
  },
  "http_req_sending": {
    "values": {
      "med": 0,
      "max": 46.0673,
      "p(90)": 1.0006,
      "p(95)": 3.9989,
      "avg": 0.567867719999997,
      "min": 0
    },
    "type": "trend",
    "contains": "time"
  },
  "http_req_failed": {
    "type": "rate",
    "contains": "default",
    "values": { "passes": 0, "fails": 75000, "rate": 0
      ↪   }
  },
  "http_reqs": {
    "values": { "count": 75000, "rate": 1008
```

```
        ↪ .5407737536918 },
      "type": "counter",
      "contains": "default"
    },
    "http_req_waiting": {
      "type": "trend",
      "contains": "time",
      "values": {
        "p(90)": 14004.83583,
        "p(95)": 14867.68577,
        "avg": 10426.636785717255,
        "min": 895.8309,
        "med": 11130.538250000001,
        "max": 19249.3024
      }
    }
  }
}
```

B.3 Function-as-a-service second test

This section includes the JSON reports for the second test performed on the FaaS application, focused on the effects of cold starts.

Listing B.9: Report from `summary-low-cs.json`

```
{
  "options": {
    "summaryTrendStats": ["avg", "min", "med", "max",
      ↪ "p(90)", "p(95)"],
    "summaryTimeUnit": "",
    "noColor": false
  },
  "state": { "isStdOutTTY": true, "isStdErrTTY": true,
    ↪ "testRunDurationMs": 38005.9903 },
  "metrics": {
    "vus_max": {
      "type": "gauge",
      "contains": "default",
      "values": { "min": 1000, "max": 1000, "value": 100
        ↪ 0 }
    },
"http_req_failed": {
  "type": "rate",
  "contains": "default",
  "values": { "rate": 0, "passes": 0, "fails": 5000
     ↪ }
},
"vus": {
  "contains": "default",
  "values": { "value": 32, "min": 32, "max": 1000 },
  "type": "gauge"
},
"iterations": {
  "type": "counter",
  "contains": "default",
  "values": { "count": 5000, "rate": 131
     ↪ .55820860165824 }
},
"http_reqs": {
  "type": "counter",
  "contains": "default",
  "values": { "count": 5000, "rate": 131
     ↪ .55820860165824 }
},
"iteration_duration": {
  "contains": "time",
  "values": {
    "avg": 7228.33648000003,
    "min": 1433.7879,
    "med": 5383.89595,
    "max": 22115.5299,
    "p(90)": 14785.201440000006,
    "p(95)": 17801.432
  },
  "type": "trend"
},
"http_req_receiving": {
  "type": "trend",
  "contains": "time",
  "values": {
    "avg": 3.5025612000000046,
    "min": 0,
    "med": 0,
        "max": 186.3514,
        "p(90)": 9.449290000000001,
        "p(95)": 17.037025000000003
      }
    },
    "http_req_duration{expected_response:true}": {
      "type": "trend",
      "contains": "time",
      "values": {
        "max": 20100.7539,
        "p(90)": 13026.138710000001,
        "p(95)": 15788.269820000001,
        "avg": 6014.108323380006,
        "min": 429.5608,
        "med": 4363.4167
      }
    },
    "http_req_blocked": {
      "type": "trend",
      "contains": "time",
      "values": {
        "med": 0,
        "max": 1233.1412,
        "p(90)": 1011.9250000000002,
        "p(95)": 1047.849335,
        "avg": 208.79735558000027,
        "min": 0
      }
    },
    "http_req_connecting": {
      "type": "trend",
      "contains": "time",
      "values": {
        "p(90)": 129.71833,
        "p(95)": 151.3786,
        "avg": 26.735437420000025,
        "min": 0,
        "med": 0,
        "max": 238.6006
      }
    },
    "data_sent": {
```

```
    "type": "counter",
    "contains": "data",
    "values": { "count": 1109000, "rate": 29179
        ↪ .6106678478 }
  },
  "http_req_duration": {
    "type": "trend",
    "contains": "time",
    "values": {
      "min": 429.5608,
      "med": 4363.4167,
      "max": 20100.7539,
      "p(90)": 13026.138710000001,
      "p(95)": 15788.269820000001,
      "avg": 6014.108323380006
    }
  },
  "http_req_tls_handshaking": {
    "type": "trend",
    "contains": "time",
    "values": {
      "max": 1010.2132,
      "p(90)": 800.4747,
      "p(95)": 815.6917,
      "avg": 162.8805984799998,
      "min": 0,
      "med": 0
    }
  },
  "checks": {
    "type": "rate",
    "contains": "default",
    "values": { "rate": 1, "passes": 5000, "fails": 0
        ↪ }
  },
  "data_received": {
    "type": "counter",
    "contains": "data",
    "values": { "count": 16209050, "rate": 426486
        ↪ .71622694173 }
  },
  "http_req_sending": {
        "contains": "time",
        "values": {
          "avg": 0.12210992000000004,
          "min": 0,
          "med": 0,
          "max": 25.5194,
          "p(90)": 0,
          "p(95)": 0.9819
        },
        "type": "trend"
      },
      "http_req_waiting": {
        "type": "trend",
        "contains": "time",
        "values": {
          "p(90)": 13025.53357,
          "p(95)": 15787.939315000003,
          "avg": 6010.483652260008,
          "min": 429.5608,
          "med": 4360.9462,
          "max": 20096.7541
        }
      }
    },
    "root_group": {
      "name": "",
      "path": "",
      "id": "d41d8cd98f00b204e9800998ecf8427e",
      "groups": [],
      "checks": [
        {
          "id": "1461660757a913d4fb82ac4c5e1009de",
          "passes": 5000,
          "fails": 0,
          "name": "status was 200",
          "path": "::status was 200"
        }
      ]
    }
}
```

Listing B.10: Report from `summary-mid-cs.json`

```
{
  "root_group": {
    "name": "",
    "path": "",
    "id": "d41d8cd98f00b204e9800998ecf8427e",
    "groups": [],
    "checks": [
      {
        "name": "status was 200",
        "path": "::status was 200",
        "id": "1461660757a913d4fb82ac4c5e1009de",
        "passes": 22781,
        "fails": 2219
      }
    ]
  },
  "options": {
    "summaryTrendStats": ["avg", "min", "med", "max",
       ↪ "p(90)", "p(95)"],
    "summaryTimeUnit": "",
    "noColor": false
  },
  "state": { "isStdOutTTY": true, "isStdErrTTY": true,
     ↪ "testRunDurationMs": 84535.9198 },
  "metrics": {
    "iteration_duration": {
      "type": "trend",
      "contains": "time",
      "values": {
        "min": 1065.0831,
        "med": 13260.1146,
        "max": 53185.534,
        "p(90)": 40872.28198,
        "p(95)": 41459.534799999994,
        "avg": 15767.499265775998
      }
    },
    "http_req_tls_handshaking": {
      "type": "trend",
      "contains": "time",
      "values": {
        "max": 6906.9068,
        "p(90)": 3880.4081000000006,
        "p(95)": 4391.707625,
        "avg": 768.2713301720024,
        "min": 0,
        "med": 0
      }
    },
    "http_req_receiving": {
      "type": "trend",
      "contains": "time",
      "values": {
        "med": 0,
        "max": 341.7664,
        "p(90)": 8.504420000000001,
        "p(95)": 16.022739999999995,
        "avg": 3.2749226359999932,
        "min": 0
      }
    },
    "iterations": {
      "contains": "default",
      "values": { "count": 25000, "rate": 295
          ↪ .73227639974175 },
      "type": "counter"
    },
    "http_req_waiting": {
      "type": "trend",
      "contains": "time",
      "values": {
        "p(95)": 35321.0508,
        "avg": 13891.682027011966,
        "min": 60.7449,
        "med": 12109.277750000001,
        "max": 46974.3225,
        "p(90)": 35071.03296
      }
    },
    "http_req_blocked": {
      "contains": "time",
      "values": {
        "p(95)": 5096.7343,
        "avg": 867.2968929080005,
      "min": 0,
      "med": 0,
      "max": 7233.4191,
      "p(90)": 4382.59838
    },
    "type": "trend"
  },
  "data_sent": {
    "type": "counter",
    "contains": "data",
    "values": { "count": 5545000, "rate": 65593
       ↪ .41890546272 }
  },
  "http_req_duration": {
    "type": "trend",
    "contains": "time",
    "values": {
      "avg": 13895.424437935875,
      "min": 62.5355,
      "med": 12114.31175,
      "max": 46981.8792,
      "p(90)": 35072.19472000001,
      "p(95)": 35322.0514
    }
  },
  "http_req_connecting": {
    "type": "trend",
    "contains": "time",
    "values": {
      "med": 0,
      "max": 697.7022,
      "p(90)": 354.1587400000023,
      "p(95)": 488.033805,
      "avg": 74.95804263200006,
      "min": 0
    }
  },
  "http_reqs": {
    "type": "counter",
    "contains": "default",
    "values": { "count": 25000, "rate": 295
       ↪ .73227639974175 }
    },
    "vus_max": {
      "type": "gauge",
      "contains": "default",
      "values": { "value": 5000, "min": 5000, "max": 500
         ↪ 0 }
    },
    "http_req_failed": {
      "type": "rate",
      "contains": "default",
      "values": { "rate": 0.08876, "passes": 2219,
         ↪ "fails": 22781 }
    },
    "vus": {
      "type": "gauge",
      "contains": "default",
      "values": { "value": 158, "min": 158, "max": 5000
         ↪ }
    },
    "http_req_duration{expected_response:true}": {
      "type": "trend",
      "contains": "time",
      "values": {
        "p(90)": 17690.0294,
        "p(95)": 30281.0625,
        "avg": 11877.448480387111,
        "min": 62.5355,
        "med": 11382.6418,
        "max": 46981.8792
      }
    },
    "data_received": {
      "values": { "count": 80180562, "rate": 948479
         ↪ .2049308252 },
      "type": "counter",
      "contains": "data"
    },
    "http_req_sending": {
      "type": "trend",
      "contains": "time",
      "values": {
        "p(90)": 0.6914100000000024,
        "p(95)": 2.0465599999999626,
        "avg": 0.4674882879999962,
        "min": 0,
        "med": 0,
        "max": 36.8756
      }
    },
    "checks": {
      "type": "rate",
      "contains": "default",
      "values": { "passes": 22781, "fails": 2219,
         ↪ "rate": 0.91124 }
    }
  }
}
```

Listing B.11: Report from `summary-high-cs.json`

```
{
  "metrics": {
    "checks": {
      "type": "rate",
      "contains": "default",
      "values": { "rate": 0.8522, "passes": 42610,
         ↪ "fails": 7390 }
    },
    "vus_max": {
      "type": "gauge",
      "contains": "default",
      "values": { "value": 10000, "min": 5287, "max": 10
         ↪ 000 }
    },
    "http_req_receiving": {
      "type": "trend",
      "contains": "time",
      "values": {
        "p(90)": 8.016769999999996,
        "p(95)": 16.027464999999996,
        "avg": 3.1523863639999776,
        "min": 0,
        "med": 0,
        "max": 902.0161
    }
  },
  "iterations": {
    "type": "counter",
    "contains": "default",
    "values": { "count": 50000, "rate": 386
        ↪ .1630983911739 }
  },
  "vus": {
    "type": "gauge",
    "contains": "default",
    "values": { "min": 0, "max": 10000, "value": 85 }
  },
  "http_req_failed": {
    "type": "rate",
    "contains": "default",
    "values": { "rate": 0.1478, "passes": 7390,
        ↪ "fails": 42610 }
  },
  "data_sent": {
    "type": "counter",
    "contains": "data",
    "values": { "count": 11221520, "rate": 86666
        ↪ .73863717052 }
  },
  "http_req_duration{expected_response:true}": {
    "type": "trend",
    "contains": "time",
    "values": {
      "avg": 17191.217868068492,
      "min": 64.236,
      "med": 17275.06855,
      "max": 56029.3063,
      "p(90)": 34100.05332,
      "p(95)": 42355.268124999995
    }
  },
  "http_req_waiting": {
    "type": "trend",
    "contains": "time",
    "values": {
      "avg": 19942.40207738391,
      "min": 64.0495,
      "med": 20453.60735,
      "max": 56318.1569,
      "p(90)": 35793.02469,
      "p(95)": 42179.86781999999
    }
  },
  "iteration_duration": {
    "contains": "time",
    "values": {
      "avg": 22223.051389623473,
      "min": 1068.1265,
      "med": 21682.4156,
      "max": 61079.3384,
      "p(90)": 43803.32307,
      "p(95)": 45602.928765000004
    },
    "type": "trend"
  },
  "data_received": {
    "type": "counter",
    "contains": "data",
    "values": { "rate": 1238802.8707926986, "count": 1
       ↪ 60398919 }
  },
  "http_req_connecting": {
    "type": "trend",
    "contains": "time",
    "values": {
      "p(95)": 2243.999899999998,
      "avg": 267.27414026399805,
      "min": 0,
      "med": 0,
      "max": 4091.2539,
      "p(90)": 651.25324
    }
  },
  "http_reqs": {
    "type": "counter",
    "contains": "default",
    "values": { "count": 50000, "rate": 386
       ↪ .1630983911739 }
    },
    "http_req_sending": {
      "type": "trend",
      "contains": "time",
      "values": {
        "med": 0,
        "max": 102.0284,
        "p(90)": 0.5943599999999977,
        "p(95)": 2.0005,
        "avg": 0.6145548659999986,
        "min": 0
      }
    },
    "http_req_tls_handshaking": {
      "values": {
        "min": 0,
        "med": 0,
        "max": 12386.4498,
        "p(90)": 4289.26061,
        "p(95)": 5948.804739999998,
        "avg": 983.7220015520023
      },
      "type": "trend",
      "contains": "time"
    },
    "http_req_duration": {
      "type": "trend",
      "contains": "time",
      "values": {
        "med": 20458.3538,
        "max": 56318.1569,
        "p(90)": 35794.3694,
        "p(95)": 42182.57002,
        "avg": 19946.169018613906,
        "min": 64.236
      }
    },
    "http_req_blocked": {
      "type": "trend",
      "contains": "time",
      "values": {
        "avg": 1271.9864004400051,
          "min": 0,
          "med": 0,
          "max": 12984.9255,
          "p(90)": 6426.418249999999,
          "p(95)": 8171.33927
        }
      }
    },
    "root_group": {
      "path": "",
      "id": "d41d8cd98f00b204e9800998ecf8427e",
      "groups": [],
      "checks": [
        {
          "path": "::status was 200",
          "id": "1461660757a913d4fb82ac4c5e1009de",
          "passes": 42610,
          "fails": 7390,
          "name": "status was 200"
        }
      ],
      "name": ""
    },
    "options": {
      "summaryTrendStats": ["avg", "min", "med", "max",
         ↪ "p(90)", "p(95)"],
      "summaryTimeUnit": "",
      "noColor": false
    },
    "state": { "isStdOutTTY": true, "isStdErrTTY": true,
       ↪ "testRunDurationMs": 129478.9694 }
}
```

Listing B.12: Report from `summary-extreme-cs.json`

```
{
    "options": {
      "summaryTrendStats": ["avg", "min", "med", "max",
         ↪ "p(90)", "p(95)"],
      "summaryTimeUnit": "",
      "noColor": false
    },
"state": { "testRunDurationMs": 131943.4724,
    ↪ "isStdOutTTY": true, "isStdErrTTY": true },
"metrics": {
  "http_req_blocked": {
    "contains": "time",
    "values": {
      "avg": 2138.7075253893227,
      "min": 0,
      "med": 0,
      "max": 15973.0723,
      "p(90)": 11755.882580000001,
      "p(95)": 12732.0112
    },
    "type": "trend"
  },
  "vus_max": {
    "type": "gauge",
    "contains": "default",
    "values": { "value": 15000, "min": 5771, "max": 15
        ↪ 000 }
  },
  "http_req_receiving": {
    "values": {
      "max": 655.3063,
      "p(90)": 6.5188,
      "p(95)": 13.212400000000029,
      "avg": 2.4422041080000074,
      "min": 0,
      "med": 0
    },
    "type": "trend",
    "contains": "time"
  },
  "http_req_tls_handshaking": {
    "type": "trend",
    "contains": "time",
    "values": {
      "avg": 1554.8014949679962,
      "min": 0,
      "med": 0,
      "max": 14712.8259,
      "p(90)": 7824.288870000017,
      "p(95)": 10634.73118
    }
  },
  "http_req_waiting": {
    "type": "trend",
    "contains": "time",
    "values": {
      "med": 22862.12385,
      "max": 60005.8953,
      "p(90)": 39018.2799,
      "p(95)": 44870.95601500005,
      "avg": 19465.382570942806,
      "min": 0
    }
  },
  "http_req_duration": {
    "contains": "time",
    "values": {
      "avg": 19468.26747831079,
      "min": 0,
      "med": 22862.12385,
      "max": 60005.8953,
      "p(90)": 39025.83376,
      "p(95)": 44885.44857000001
    },
    "type": "trend"
  },
  "data_received": {
    "contains": "data",
    "values": { "count": 208103337, "rate": 1577215
       ↪ .8577812298 },
    "type": "counter"
  },
  "iterations": {
    "type": "counter",
    "contains": "default",
    "values": { "count": 75000, "rate": 568
       ↪ .4252402622079 }
  },
  "http_req_duration{expected_response:true}": {
    "contains": "time",
    "values": {
        "med": 24494.5585,
        "max": 59992.5854,
        "p(90)": 39902.416769999996,
        "p(95)": 43925.3683,
        "avg": 21497.857275753828,
        "min": 69.1057
      },
      "type": "trend"
    },
    "http_req_connecting": {
      "values": {
        "med": 0,
        "max": 15352.9349,
        "p(90)": 1316.17695,
        "p(95)": 3060.839045000006,
        "avg": 576.8669205973348,
        "min": 0
      },
      "type": "trend",
      "contains": "time"
    },
    "checks": {
      "type": "rate",
      "contains": "default",
      "values": { "fails": 27042, "rate": 0.63944,
         ↪ "passes": 47958 }
    },
    "http_req_sending": {
      "contains": "time",
      "values": {
        "p(90)": 0.6399500000000203,
        "p(95)": 2.0069,
        "avg": 0.4427032600000003,
        "min": 0,
        "med": 0,
        "max": 58.8432
      },
      "type": "trend"
    },
    "vus": {
      "type": "gauge",
      "contains": "default",
    "values": { "value": 59, "min": 0, "max": 15000 }
  },
  "http_reqs": {
    "type": "counter",
    "contains": "default",
    "values": { "rate": 568.4252402622079, "count": 75
      ↪ 000 }
  },
  "http_req_failed": {
    "type": "rate",
    "contains": "default",
    "values": { "rate": 0.36056, "passes": 27042,
      ↪ "fails": 47958 }
  },
  "iteration_duration": {
    "type": "trend",
    "contains": "time",
    "values": {
      "min": 1000.9921,
      "med": 25127.9883,
      "max": 61098.7345,
      "p(90)": 47785.477750000005,
      "p(95)": 50092.22874500001,
      "avg": 22617.240952045442
    }
  },
  "data_sent": {
    "type": "counter",
    "contains": "data",
    "values": { "count": 14897900, "rate": 112911
      ↪ .23182536464 }
  }
},
"root_group": {
  "name": "",
  "path": "",
  "id": "d41d8cd98f00b204e9800998ecf8427e",
  "groups": [],
  "checks": [
    {
      "name": "status was 200",
      "path": "::status was 200",
        "id": "1461660757a913d4fb82ac4c5e1009de",
        "passes": 47958,
        "fails": 27042
      }
    ]
  }
}
```

TRITA-EECS-EX- 2023:411