UNIVERSITETI I EVROPËS JUGLINDORE
УНИВЕРЗИТЕТ НА ЈУГОИСТОЧНА ЕВРОПА
SOUTH EAST EUROPEAN UNIVERSITY

FAKULTETI I SHKENCAVE DHE TEKNOLOGJIVE BASHKËKOHORE
ФАКУЛТЕТ ЗА СОВРЕМЕНИ НАУКИ И ТЕХНОЛОГИИ
FACULTY OF CONTEMPORARY SCIENCES AND TECHNOLOGIES

POSTGRADUATE STUDIES - SECOND CYCLE

THESIS:

FUNCTION-AS-A-SERVICE PERFORMANCE EVALUATION WITH APPLICATION-LEVEL WORKFLOWS

CANDIDATE:
Blend Hamiti

MENTOR:
Prof. Dr. Besnik Selimi

Tetovo, February 2023

Function-as-a-Service Performance Evaluation with Application-level Workflows

Cloud Computing

Master Thesis

Blend Hamiti

Faculty of Contemporary Sciences and Technologies
South East European University
Tetovo

March 14, 2023

Abstract

The thesis evaluates the performance of Function as a Service in multiple cloud services providers (CSPs) with an application-level benchmark. The evaluated CSPs are Amazon Web Services (AWS), Azure, and Google Cloud Platform (GCP). The benchmark consists of an application that uses different cloud services to replicate the processes that occur when creating a user profile in a typical application. Timestamps are recorded at different steps of the executing application, and the difference between the timestamps is used to calculate the triggering and execution duration of each service. The calculated durations are used to compare the consistency in performance in each platform and the performance variation between platforms. We find that consistency is moderate. We find that there are significant performance differences between the platforms.

Keywords: Cloud Computing, Serverless Computing, Function as a Service, Application Benchmark, Consistency

Përmbledhje

Teza vlerëson performancën e Function as a Service në disa ofrues të shërbimeve cloud (OSHC) në nivel të aplikacionit. OSHC-të e vlerësuara janë Amazon Web Services (AWS), Azure, dhe Google Cloud Platform (GCP). Për të bërë krahasimet e duhura përdoret një aplikacion i përbërë nga shërbime të ndryshme cloud që përsërit proceset që ndodhin gjatë krijimit të një profili përdoruesi në një aplikacion tipik. Vula kohore regjistrohen në hapa të ndryshëm të aplikacionit ekzekutues dhe diferenca midis vulave kohore përdoret për të llogaritur kohëzgjatjen e aktivizimit dhe ekzekutimit të secilit shërbim. Kohëzgjatjet e llogaritura përdoren për të krahasuar konsistencën në performancën e secilës platformë dhe dallimin në performancë midis platformave. Ne gjejmë se konsistenca është mesatare. Ne gjejmë se ka dallime domethënëse në performancë midis platformave.

Абстракт

Тезата го оценува перформансата на Фанкшн-ес-еј-Сервис во повеќе даватели на клауд услуги (ДКУи) на ниво на апликација. Оценетите ДКУи се Амазон Веб Срвисес (АВС), Ежур, и Гугл Клауд Платформ (ГКП). За да се направат потребните споредби, се користи апликација составена од различни клауд услуги која ги реплицира процесите што се случуваат при креирање на кориснички профил во типична апликација. Временски печати се снимаат во различни чекори на апликацијата што се извршува, и разликата помеѓу временските печати се користи за пресметување на времетраењето на активирањето и извршувањето на секоја услуга. Пресметаното времетраење се користи за да се спореди конзистентноста на перформансата во секоја платформа и варијацијата во перформанса помеѓу платформите. Откривме дека конзистентноста е умерена. Откривме дека постојат значителни разлики во перформанса помеѓу платформите.

Declaration of originality

I certify that I am the original author of this work. The copyright is transferred to the University for use for educational and research purposes.

Acknowledgements

I am grateful to my thesis advisor for their guidance and support throughout the selection and composition of my thesis. Their insightful comments were invaluable to me. I also appreciate the assistance of my friend Premt in creating the graphs for my thesis. Lastly, I express my heartfelt thanks to my family for their constant support.

Contents

List of figures

List of tables

Chapter 1

Introduction

These days, most organizations store and process data on the cloud. According to the Flexera Cloud Computing report for 2022, 96% of the organizations surveyed around the world use a cloud [23]. Cloud computing is commonly used to refer to computation services in the cloud. These services include applications, data storage, networking capabilities, etc.

Depending on the level of abstraction, there are three major cloud computing service models. They are: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). In addition to these, there is the Function as a Service (FaaS) model. FaaS allows developers to write event-driven, stateless functions. It makes use of the serverless execution model where applications are packaged into containers that run on-demand and users are charged for actual usage rather than the resources allocated.

There are various performance challenges with FaaS due to its underlying architecture. There is the cold start delay, the heterogeneity of the infrastructure that functions run on, the different event sources that trigger the functions, etc. The aim of the thesis is to evaluate the performance of FaaS in different cloud services providers (CSPs) through a reproducible, application-level benchmark.

Micro-benchmarks evaluate a certain aspect of a platform, e.g. CPU performance, file I/O, network speed. The functions are deployed as FaaS, and are usually triggered through a script. Application-level benchmarks, on the other hand, evaluate performance by measuring various parameters, e.g. response times, of an application that uses FaaS. They usually model real-life use cases and as such evaluate the capabilities of these platforms in a more meaningful way.

The thesis' results could help organizations in choosing a CSP for their serverless functions. The results can also be used as part of a performance overview of the CSPs over time. In addition, the study provides insights for deploying the same application in different CSPs.

1.1 Motivation

Scheuner and Leitner performed a systematic literature review on FaaS performance evaluation, and found some shortcomings [17]. They found that not all FaaS platforms are equally studied, only a few platform configurations are used, micro-benchmarks are more commonly performed than application-level benchmarks, and the results are rarely reproducible. Thus, they propose future studies that can fill these gaps.

Scheuner and Leitner found that AWS was over-studied by a factor of three [17]. The platform configurations used included the language runtime of the function, the function trigger, and external services used as part of the workflow. Regarding the benchmark type, microbenchmarks were performed in two thirds of the studies selected. Finally, when analyzing the reproducibility of the studies, Scheuner and Leitner considered the principles put forward by Papadopoulos et al. and found that the reproducibility principles were generally not followed in the reviewed cases [20].

1.2 Research questions

We propose two hypotheses. We expect the performance of FaaS to differ between consecutive runs in the same CSP, and between CSPs:

Hypothesis 1 Performance of each CSP varies between consecutive runs.

Hypothesis 2 Performance of each CSP is different from the others.

To test the hypotheses, we put forward three research questions. The first question helps us define the setup we use to evaluate performance:

RQ 1 How do we define the benchmark?

RQ 1.1 Which CSPs should be included in the benchmark?

RQ 1.2 What application processes are suitable for testing performance of FaaS?

RQ 1.3 How will we measure performance?

RQ 1.4 How many benchmark runs are needed?

The question is broken down into four sub questions. First, we need to look at the CSPs that are considered in the benchmark. Second, we need to make sure that the deployed application resembles real applications. Then, we need to define the data that is collected, and the number of measurements needed to come to conclusions.

To validate the first hypothesis, we need to define a measure for performance variation:

RQ 2 How can we quantify the performance variation in each CSP?

Finally, we need to select the relevant measurements that allow us to compare the performance of CSPs among each other:

RQ 3 What aspects of the application should be considered to compare performance between CSPs?

1.3 Thesis outline

Chapter 2 provides background information and presents the relevant work. The first section explains the core concepts required to understand the thesis, while the second section provides a summary of FaaS performance evaluation studies that are comparable to our study. Chapter 3 describes the study design and implementation. We first describe the research method, which consists of six phases, and then mention the tools used for the experiment. Chapter 4 presents the study results. The results consist of a table that displays the numeric values of the calculated measurements, and CDF plots that show the empirical distribution of the measurements. Chapter 5 provides answers to the research questions, a discussion of the results, and validation of the hypotheses. We then mention the threats to validity, and the followed reproducibility principles. Finally, Chapter 6 states the conclusions and the suggestions for future work.

Chapter 2

Analysis

This chapter consists of two parts, the thesis background and related work. The background section presents the definitions needed for the rest of the thesis. The related work section comprises studies whose results are comparable to the results of the current study.

2.1 Background

We first explain what a cloud is, the types of clouds, and the advantages and disadvantages of each cloud type. Then, cloud computing, serverless computing, and FaaS, are discussed. The challenges with FaaS, use cases, and limitations are also analyzed. Finally, we give an overview of Infrastructure as Code (IaC).

2.1.1 The cloud

A cloud is a virtual (computing) space which is hosted in a data center, i.e. interconnected network of computers. The computers in data centers are typically called servers. Servers are often dedicated computers, i.e. specialized in one task, and have different hardware from desktop computers which are general-purpose and user-friendly.

Depending on the location of the data center, a cloud is either private or public. A private cloud is also called an on-premises data center, meaning that the organization sets up their own data center and has control over the whole infrastructure. A public cloud is a virtual space in an infrastructure managed by another organization, typically called a provider or cloud services provider

(CSP), and is multi-tenant. A multi-tenant system is physically shared by multiple organizations, but every organization is logically separated.

A system can be deployed on a private cloud, a public cloud, or a mix of one or more private and public clouds. This is called the deployment model of the system. In a hybrid cloud deployment, clouds of both types are used for the system. In a multi-cloud deployment, multiple instances of the same cloud type are used.

Organizations use public clouds to avoid infrastructure set-up and maintenance. There are no upfront expenses for set-up, and services are offered via the pay-per-use model. There are abstraction layers which make developing systems fast and easy, and basic security is included. The drawbacks of public clouds, on the other hand, are vendor and data lock in, and ensuring the NFRs of the system. Due to the lack of standards, moving business logic or data between different platforms may be difficult or impossible [11]. In addition, ensuring data security, performance SLAs, compliance to standards, and system availability may not be possible.

Organizations choose to set-up a private cloud mostly to get control of their data, have ownership over the hardware, and due to legal requirements. This also allows them to ensure the NFRs of the system. Some organizations opt to do computing at the edge to avoid the latency of using public clouds [15]. This is especially helpful for Internet of Things (IoT) networks where tasks are simple and small. In a private cloud, however, organizations need to monitor security (both physical and virtual), and scale infrastructure as needed.

Organizations can still opt in to use a public cloud along a private cloud to avoid replicating their infrastructure and over-provisioning resources. A public cloud helps them deal with cloudbursting, i.e. spikes in demand. Public clouds also have infrastructure in multiple geographical regions. This allows organizations to provide low-latency services to clients distributed over different regions, and create backups of their data.

There are currently three CSPs that together have a share of 65% of the cloud market in the world [22]. They are Amazon Web Services (AWS), Azure, and Google Cloud Platform (GCP). These three CSPs are the subject of this study.

2.1.2 Cloud computing

Cloud computing refers to the use of virtualized resources in the cloud. Depending on the level of abstraction, there are three main cloud computing service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as

a Service (SaaS). Figure 2.1 shows the abstraction level in each model. The blocks in grey indicate the layers managed by the CSP, and the blocks in white indicate the layers managed by the user.

IaaS are infrastructure resources that are configurable up to (and including) the OS layer. Examples of these services are virtual machines (VMs), storage, and networking. They are usually operated by system administrators.

PaaS services provide a managed platform where organizations can create and deploy applications. The CSP manages hosting, software maintenance, and the runtime. Examples of PaaS services are web servers, databases, and development tools. They are usually operated by developers.

SaaS services are managed applications delivered to users over the internet. These are completed products that users use without worrying about its maintenance and infrastructure management. Examples of SaaS services are games, productivity tools, customer relationship management (CRM) systems, and enterprise resource planning (ERP) systems.

2.1.3 Serverless computing

Serverless computing is an execution model for cloud computing. Applications are packaged into containers that run on-demand, and CSPs dynamically allocate machine resources for them. Since both infrastructure management and scaling are done by the CSP, serverless computing allows cloud computing to be used as a utility [2]. This also leads to users being charged for actual usage rather than the resources allocated.

Eismann et al. did a systematic review of the use cases for serverless computing [19]. Regarding application categories, serverless was mainly used for APIs, stream/async processing, batch tasks, and operations and monitoring at a roughly equal rate. Regarding the application domain, serverless was used for web services, generic tasks, scientific computing, and IoT. Finally, regarding latency, they found that currently there are not many use cases where low latency or a certain degree of stable latency is required. The authors also predict that applications with stricter latency requirements will never make use of serverless computing due to the possible cases of violation.

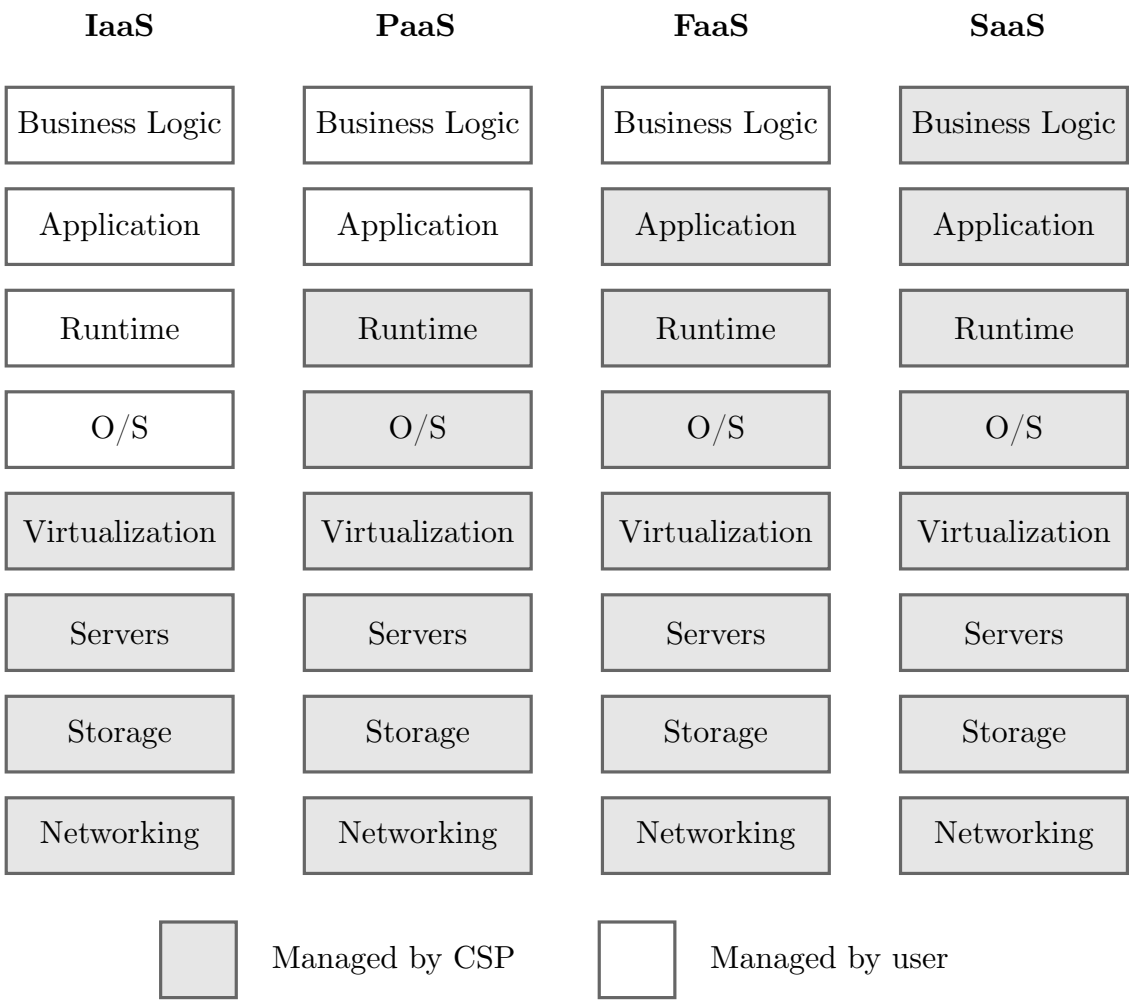

Figure 2.1: Abstraction layers in cloud computing models.

2.1.4 Function as a Service

Function as a Service (FaaS) is a service model that makes use of the serverless execution model. FaaS allows developers to write event-driven, stateless functions. The level of abstraction in the FaaS model is depicted in Figure 2.1.

The programming model of FaaS is made up of an action and a trigger [11]. The trigger is the event that invokes the function, and the action is the function that is executed. The number of the different events that can invoke the functions is related to the self-awareness of the architecture, and thus depends on the CSP [6]. Some common event sources are API gateways, event streams, queues, and storage. Depending on the event source, a function invocation is synchronous or asynchronous.

Being stateless, functions can be scaled horizontally. This makes FaaS suitable for short and bursty loads. FaaS is not suitable, however, for high end computing. Although each platform provides some configuration options, there are only a few CPU and memory size configurations to choose from.

There are tools and frameworks created to aid with developing and deploying serverless applications. Open-source frameworks, such as serverless[1], allow users to define their functions and triggers, and deploy their applications in the cloud. Infrastructure as Code tools allow users to define their functions as part of the infrastructure. Finally, some platforms, such as Azure, provide plugins for popular IDEs to test and deploy functions.

Cold start

There is an additional delay that happens during the function invocation due to the runtime being set-up. This is called the cold start delay. The cold start duration depends on the complexity of the environment [10]. Most platforms claim that they do not charge users for the cold start duration [18]. Manner et al. proved that this is true for AWS [10].

There is a certain amount of time that function containers stay ”warm”. After some time of inactivity, that depends on the specific platform implementation, the containers are destroyed. McGrath and Breener performed a backoff test to measure the amount of time that the containers stay warm and get reused [7]. Azure Functions were affected by cold start latency after only 5 minutes of function idling, while AWS Lambda and Google Cloud Functions did not show cold start signs even after a 30-minute backoff time.

Use cases

Giménez-Alventosa et al. used FaaS to analyze large datasets with the MapReduce model [13]. They used AWS Lambda for the function code and AWS S3 to persist the data. They found FaaS to be a fitting implementation for scientific workloads as long as the constraints of the platform do not interfere with the processing. The constraints were the 15-minute function invocation timeout, the limited memory and disk space configurations, and the inhomogeneous behavior between the machines where functions execute.

[1] https://www.serverless.com/framework/

Ishakian et al. used FaaS to serve a deep learning model [8]. They used AWS Lambda. The response times were acceptable with warm function executions, but cold start latency led to cases of violation for their SLAs. Thus, they proposed that platforms providing FaaS should allow users to specify a minimum period for which function containers are kept warm. They also proposed that platforms should provide more configurations, such as access to GPUs.

Baldini et al. analyzed the use of serverless platforms for mobile application backends [3]. The backend had to support lightweight operations, but with unpredictable bursts. They found that serverless fulfilled their needs. It also simplified their backend architecture and allowed developers to focus on the client program.

Other common FaaS use-cases can be found on CSPs' websites[2,3,4]. The listed examples use FaaS to build web applications and mobile backends, to integrate with third party services and APIs, for chatbots, to process files as they are uploaded, for scheduled tasks, for tasks triggered via a message queue, to do real-time image and video analysis, and real time stream processing of data coming from an IoT network.

The FaaS model is also used for its cost-efficiency. Adzic and Chatley analyzed two use cases of applications being migrated from a monolith to a serverless architecture using FaaS [5]. They noticed cost-savings of 66% and 95% while the applications were roughly using the same computational resources.

Workarounds for FaaS limitations

There have been efforts to overcome the FaaS limitations that are present as a result of the underlying serverless architecture. While they can fulfill specific needs, they do violate one or more FaaS principles as dictated by the serverless manifesto. To the best of our knowledge, the first to create the serverless manifesto were Potes and Nair [4]. Some of the principles are discussed in this paper. They are:

P1 The single responsibility principle. Functions should perform a single action.

P2 The isolation principle. Functions should not call other functions.

P3 The scale-to-zero principle. Functions should not idle.

[2]https://docs.aws.amazon.com/lambda/latest/dg/applications-usecases.html
[3]https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview
[4]https://cloud.google.com/functions

In the industry, there often are workflows that consist of several functions running in succession. CSPs have made services available for function pipelines, such as AWS Step Functions[5], as a way to create and manage these workflows. These services allow state to be stored for the duration of the workflow. This means that the function that coordinates the workflow has to idle while the other functions execute, and they violate the isolation principle.

To mitigate the cold start problem, there are applications that continuously ping functions to keep the function containers warm. This is referred to as the *ping hack*, and one framework that provides this function is Zappa[6]. This does, however, violate the scale-to-zero principle. In addition, the ping hack only makes sure that a fixed number of containers are available, so scaling would nevertheless lead to a cold start.

When FaaS is used to serve deep learning models, the model must be downloaded from persistent storage on each function execution [8]. The downloading process creates a delay on each function call. To avoid this, users have run functions in "long-lived" mode [12], where the function stays idle and waits for new tasks to process. This does violate the scale-to-zero principle.

Palade et al. claim that efficient and effective function composition is still an open research question for FaaS [15]. There are cases in the industry where a single function fulfills all tasks for an application, e.g. a function that acts like a web server. Although this might be easier to manage and allow code reuse, it does violate the single responsibility principle. The functions should be lightweight, execute quickly, and be easy to scale. Instead, to address reuse, some platforms, such as AWS Lambda, provide function-layer sharing, where common dependencies can be packed into a layer to be used across functions. Fox et al. further claim that "FaaS could further help users by making libraries easier to use as one needn't put library routines in one's code; just invoke them as FaaS." [6].

2.1.5 Infrastructure as Code

Infrastructure as Code (IaC) is the provisioning of cloud resources via code. This allows the infrastructure to be version controlled and reproduced with ease. Developers are able to collaborate more easily, and human errors in manual

[5]https://aws.amazon.com/step-functions/
[6]https://github.com/zappa/Zappa

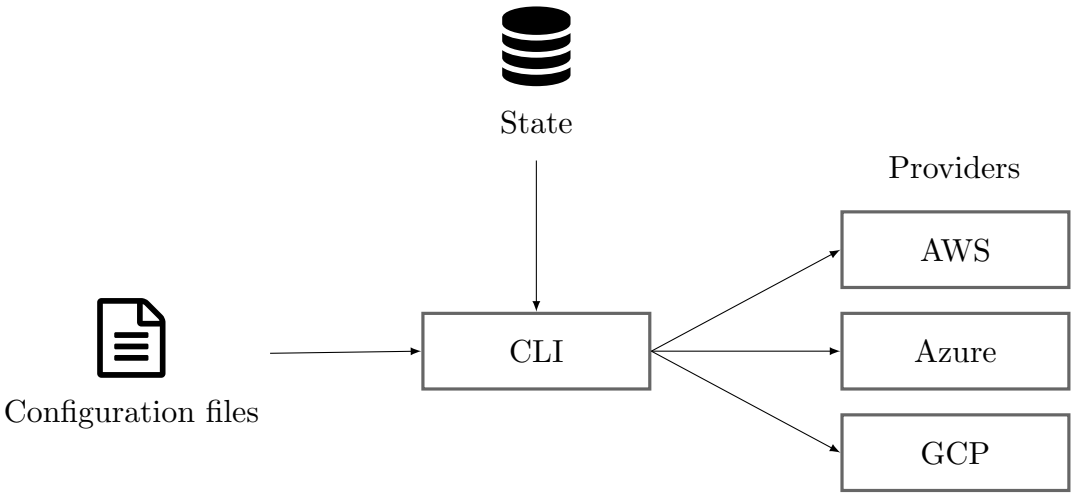

Figure 2.2: Process flow of Terraform and Pulumi.

configurations are eliminated. IaC tools, such as Terraform[7] and Pulumi[8], are used to deploy the configured infrastructure in the cloud. Terraform is the chosen tool for this study due to its well-written documentation and large community support.

The infrastructure code can be written in a declarative, or an imperative style. With declarative languages, the user specifies the desired state of the infrastructure, and the system then takes the necessary steps to achieve that state. It is easier to update the infrastructure, but there is less flexibility. With imperative languages, on the other hand, the user specifies the steps that need to be executed to achieve the desired state. The code is easier to write and understand, and can also be more efficient.

Terraform

Terraform is an open-source tool. It uses a declarative approach, and the resources of the infrastructure are described in the HashiCorp Configuration Language (HCL). Plugins, called providers, allow Terraform to interact with CSPs. A state file keeps track of the resources in the cloud and is used to bind the defined resources in code to those in the cloud. Figure 2.2 shows a diagram of the process flow for Terraform and Pulumi.

[7]https://www.terraform.io/
[8]https://www.pulumi.com/

Pulumi

Pulumi is also an open-source tool. Like Terraform, it is a declarative tool that allows the user to define the desired end state of the infrastructure. However, users use an object-oriented programming model to define those resources. The supported languages are JavaScript, Python, Go, etc.

The current state of the infrastructure is stored in a state file. A deployment engine runs the program and compares the current state to the desired state and then makes the necessary changes. Resource providers, which are tied to the chosen language, are used to handle the communication with CSPs.

2.2 Related work

We discuss several studies that relate to the work in the thesis. The studies chosen do not present comparable numeric data, but rather a relative comparison of the performance between CSPs. Thus, two or more of the CSPs analyzed in the current study need to be present for the study to be relevant. In addition, since each study provides a measure for a certain aspect of the current application, the studies are grouped by the aspect that they measure. For each study, the goal, the experiment setup, and the attained results are stated.

2.2.1 HTTP trigger duration

McGrath and Breener developed a tool to measure platform overhead using a function that immediately returns [7]. They performed a concurrency test. AWS Lambda and Azure Functions had similar overhead, while GCP Cloud Functions had twice as much. In AWS Lambda and GCP Cloud Functions, executions per second scaled linearly with the number of concurrent requests, while that was not the case with Azure Functions. The authors invoked the functions synchronously via HTTP triggers, from virtual machines set up in the same region as the infrastructure, and measured response times. Although this should have eliminated any network latency, we are curious to see how the response times compare with execution time logs from the platforms.

2.2.2 Function execution duration

Zhang et al. studied the execution duration and cost of functions with varying memory configurations [16]. With respect to cost, they found that the memory configuration of a function is not trivial. They tested both AWS Lambda and GCP Cloud Functions. AWS Lambda functions had lower execution times for all memory sizes, while GCP Cloud Functions instances were more stable, even with small memory configurations. In AWS Lambda, there was more variation in execution time as memory size decreases. AWS Lambda functions had different execution durations at different times of the day. The authors explain that execution duration varies due to the heterogeneous infrastructure on which the functions run on.

2.2.3 HTTP trigger and function execution duration

Ivan et al. compared the performance, cost, and scalability of an application deployed in virtual machines and as FaaS [14]. The application was a Web API. It consisted of four endpoints, a relational database, and file storage. The methods simulated a real application flow.

The application was deployed in virtual machines both as a monolith and with a microservices architecture. They found that the monolithic application performs better at very low loads, but performance degrades rapidly as load increases. The application with a microservices architecture did not perform as well, but performance degraded slower as load increased. The FaaS application performs better at high loads due to its scaling capabilities. The response time was the same for all load sizes.

The FaaS-based application was deployed both in AWS and Azure using the platform's respective services. The response times in the AWS implementation were on average twice as low as the response times in the Azure implementation. The AWS implementation also had higher concurrency limits.

2.2.4 Cold start duration

Manner et al. studied FaaS cold start times in AWS and Azure [10]. They found that platforms do not log the cold start overhead in the function execution time. Thus, they compared the response times on the client side with the platform logs to deduce the cold start time, and measured the increased overhead between the

cold and warm function executions. They found that AWS Lambda consistently had lower cold start overhead than Azure Functions.

2.2.5 Elasticity

Lee et al. designed a study to evaluate the elasticity of different FaaS platforms [9]. The platforms tested were AWS Lambda, Azure functions, and GCP Cloud Functions. They performed distributed data processing with concurrent invocations, and found that AWS Lambda has the lowest concurrency overhead with respect to CPU performance, file I/O, and network bandwidth. They also performed the same operations in virtual machines, and found that FaaS was able to scale well. The FaaS implementation proved to be more cost-effective.

Chapter 3

Design

This chapter describes the experiment design. The experiment consists of building and deploying the same application in three CSPs, and measuring its performance. We first describe the application infrastructure, and then define the benchmarking process. Finally, we list the tools used for the experiment.

3.1 Research method

An application that makes use of several cloud services is deployed in multiple CSPs. The application is defined in configuration files, and resembles real production applications. A mock client program is used to trigger the application processes, and timestamps are recorded for all relevant events for analysis.

The experiment consists of six phases, as shown in Figure 3.1. In the first phase, the processes that make up the application and the infrastructure are defined. In the second phase, the benchmarking process is defined. In the third phase, the application is deployed. In the fourth phase, the benchmark is executed. In the fifth phase, the experiment data is collected. In the sixth phase, the collected data is analyzed.

3.1.1 Defining the application

The application replicates the processes that take place when creating a user in a typical application. The first process starts with a user submitting a web form to an endpoint. A new user record is added to the database, and the profile picture is uploaded to persistent storage. The user is then notified whether the user account was created, and this marks the end of the first process. The second process starts

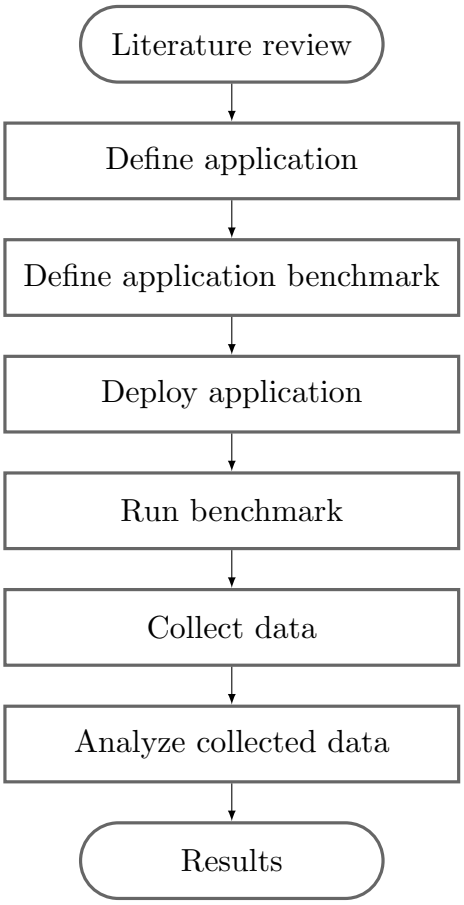

Figure 3.1: Experiment phases.

when the system detects a newly uploaded picture. It then creates a thumbnail version of the picture, and stores it in persistent storage. The first process is synchronous, and the second process is asynchronous.

The application is implemented in three platforms: AWS, Azure, and GCP. The cloud services used are API management, FaaS, object Storage and RDBMS. They are shown in Figure 3.2 and are also described below. The specific services used in each platform are shown in Table 3.1. In addition, a logging service is used. This is discussed in section 3.1.5.

RDBMS

An RDBMS service is used to deploy a MySQL database. The database contains a single table that stores user records. A user records consists of an ID, email, name, and password. The statement used to create the table can be found in Appendix A.1.

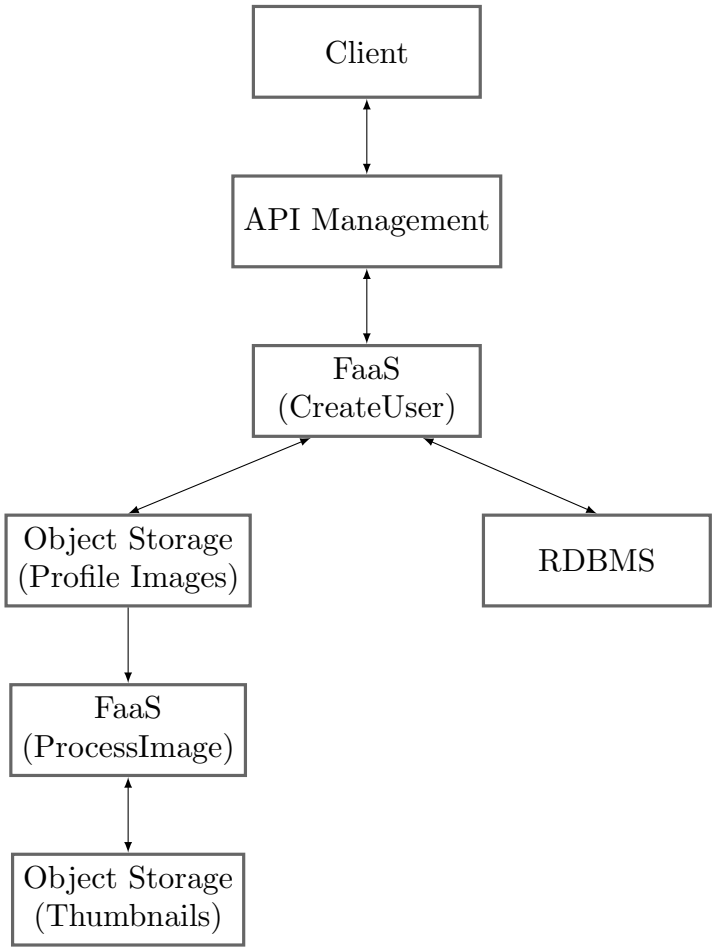

Figure 3.2: Application infrastructure.

Object storage

Object storage is used to store the profile images. Object-based storage is used to store unstructured data, such as binary data, in the form of objects. An object storage service manages storage and access to data. We use two storage containers in the application, one for the original profile images, and one for the thumbnails.

API management

The API management service is used to deploy an API gateway. An API gateway is a reverse proxy server that routes clients' requests to the backend services. API management tools do authorization of requests, monitoring of requests, traffic throttling to prevent overloads of backend resources, etc.

In our application, except for monitoring of requests, we do not make use of any of the other API management features. The API gateway routes requests

Service type	AWS	Azure	GCP
API management	API Gateway	API Management	API Gateway
FaaS	Lambda	Functions	Cloud Functions
Logging	CloudWatch	Cloud Logging	Monitor
Object storage	S3	Blob Storage	Cloud Storage
RDBMS	RDS	Database for MySQL	Cloud SQL

Table 3.1: The services used for the application in each CSP.

to an HTTP API that has a single configured route - '/users'. The route only responds to requests that use the POST verb. A request to this route triggers the CreateUser function.

FaaS

The FaaS service is used to deploy two functions, the CreateUser function and the ProcessImage function. These functions carry out the application processes.

The CreateUser function parses and transforms the form input, and stores the data in persistent storage. The form submitted by the client is in the form-data format. The form fields include a name, email, password, and profile image field. The function creates a hash of the user's entered password and creates a record in the database. The profile image is uploaded to the respective storage container. Each profile image is stored in a directory named after the user's ID from the database record.

The CreateUser function sends a response to the API gateway with a message on whether a user was created successfully. A user is not created if the provided email address is not unique. The API gateway returns this response to the client.

The ProcessImage function creates a thumbnail version of the original profile image. It is triggered when a new file is uploaded to the original profile images storage container. It then downloads the image and creates a thumbnail, which is a 200x200 pixels image. The thumbnail is stored in the thumbnails container.

3.1.2 Defining the application benchmark

The benchmark produces 100 measurements for both application processes. A measurement consists of 10 timestamps, as depicted in Figure 3.3. The meaning of each timestamp is as follows:

- t_1: The client sends a request to the API endpoint. This marks the start of the synchronous process from the client's perspective.
- t_2: The API gateway receives the request. This marks the start of the synchronous process from the platform's perspective.
- t_3: The CreateUser function starts executing.
- t_4: The profile image is uploaded.
- t_5: The CreateUser function finishes executing.
- t_6: The API gateway receives a response. This marks the end of the synchronous process from the platform's perspective.
- t_7: The client receives a response. This marks the end of the synchronous process from the client's perspective.
- t_8: The ProcessImage function starts executing. This marks the start of the asynchronous process.
- t_9: The thumbnail image is uploaded.
- t_{10}: The ProcessImage function finishes executing. This marks the end of the asynchronous process.

We then define several time durations, i.e. time intervals, that are of interest to us. The duration $t_a t_b$ for each pair (t_a, t_b) is calculated as follows:

$$t_a t_b = t_b - t_a \tag{3.1}$$

The defined durations are as follows:

- $t_1 t_7$: The time it takes for the synchronous process to complete from the client's perspective.
- $t_a t_b$: The time it takes for the platform to complete the synchronous process.
- $t_1 t_2$: The time it takes for the request to be sent from the client to the API gateway.
- $t_6 t_7$: The time it takes for the client to receive the API gateway response.

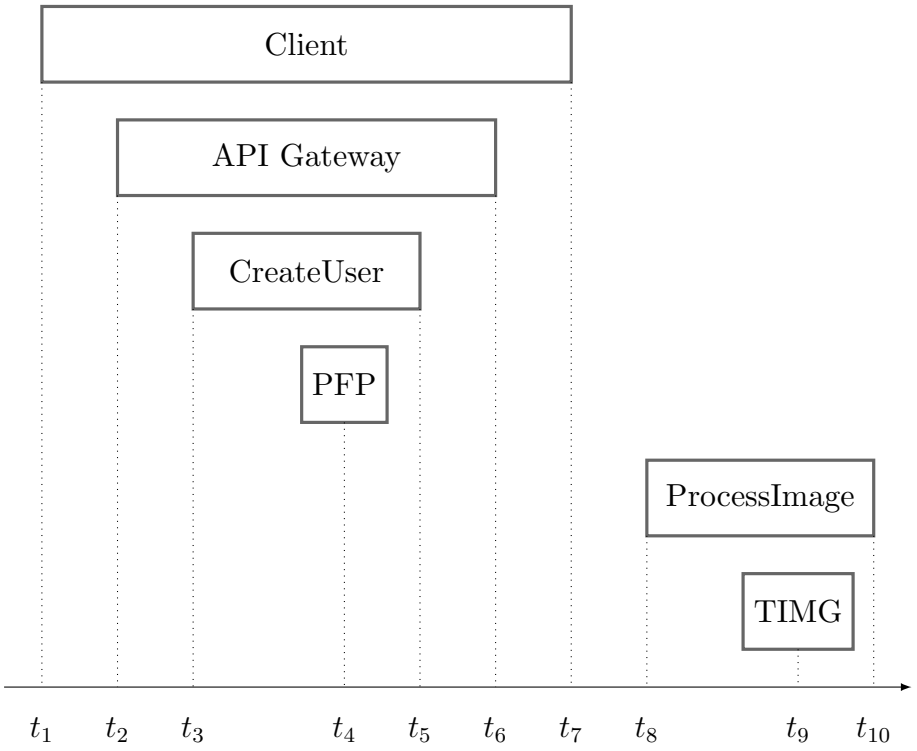

Figure 3.3: Application processes reference.

- t_2t_3: The time it takes for the API gateway to trigger the CreateUser function after receiving the client's request.
- t_3t_5: The execution duration of the CreateUser function.
- t_5t_6: The time it takes for the CreateUser function to pass the response to the API gateway.
- t_4t_8: The time it takes for the platform to discover the uploaded picture and trigger the ProcessImage function.
- t_8t_{10}: The execution duration of the ProcessImage function.
- t_4t_9: The time it takes for the thumbnail image to be generated from the time the original profile image is uploaded.

3.1.3 Deploying the application

The application infrastructure is deployed with Terraform. A separate workspace is created for each platform. The official providers for each CSP

(hashicorp/aws[1], hashicorp/google[2], hashicorp/azurerm[3]) are used. Each provider has authentication instructions and extensive documentation on how to provision resources in the CSP.

The code for the infrastructure is published on Github[4]. There is a 'README' file in each directory where specific steps on how to deploy the application in each CSP are mentioned. Authentication for all Terraform providers is done via environment variables. The variables are defined in a text file named '.env'. Each provider uses a different set of variables for authentication. An example '.env' file, named '.env.example', is included in each setup.

All cloud resources are declared using the HashiCorp Configuration Language (HCL). Resource declarations are separated into files based on which application component they belong to, i.e. storage, database, API, functions. The function code for FaaS is packaged in a .zip file with all its dependencies using the hashicorp/archive[5] provider. There are several Terraform output variables defined in each setup. Among them, a variable for the API gateway URL is found which is used in the program described in Section 3.1.4.

There is an additional step done after the terraform configuration is applied to finish deploying the application. Specifically, after the database is deployed, we create the database tables manually. The credentials to connect to the database are found in the respective resource declarations. The SQL statements used can be found both in the 'README' file and in Appendix A.1.

Application configuration

In this section, we describe the properties shared by all platforms and those that differ between them.

FaaS Function Code. The FaaS function code has the same structure in each CSP. There are slight differences, however, due to the different SDK that each platform provides. For common libraries, the same library versions are used across the functions.

FaaS Function Memory Configuration. The functions are configured with a memory of 512MB, except for the functions in Azure where memory cannot be

[1]https://registry.terraform.io/providers/hashicorp/aws/4.8.0
[2]https://registry.terraform.io/providers/hashicorp/google/4.25.0
[3]https://registry.terraform.io/providers/hashicorp/azurerm/3.5.0
[4]https://github.com/blendhamiti/faas-benchmark
[5]https://registry.terraform.io/providers/hashicorp/archive/2.2.0

configured. Azure scales memory resources as needed to a maximum of 1.5GB in the consumption plan[6], which is the fully serverless option of FaaS in the platform.

FaaS Function Runtime. All functions run on the Linux operation system and use the Node.js 14 runtime. However, this is not true in Azure, where functions run on the Windows operating system. The dependencies packaged with the function code need to be installed for the operating system where the function runs. The dependencies for the functions in Azure are thus different from those in the other CSPs.

FaaS Function Triggers. The CreateUser function is triggered via an event-based trigger from the API management service. In AWS, the function is only assigned an endpoint when the API gateway trigger is configured. In GCP and Azure, the function is assigned two endpoints, one endpoint that belongs to the function and another endpoint that is associated with the function and is managed by the API gateway. On the other hand, the ProcessImage function is triggered from the object storage service. In AWS and GCP, the trigger is an event-based trigger. In Azure, the trigger relies on polling the storage container for updates at regular intervals.

DMBS. The DBMS used in each platform is MySQL 5.7. The database is deployed in a virtual private cloud (VPC) in AWS and GCP. In Azure, the database is not deployed in a VPC due to the basic tier of the service not supporting such a deployment.

API Gateway Routes. The API routes of the API gateways in all platforms are the same. In GCP, the API is defined through an OpenAPI document.

A limitation of Terraform

Terraform state management does not work for the API gateway configuration resource of GCP. The API gateway configuration resource[7] needs to be given a new value for the ”id” property every time that the OpenAPI specification document, which describes the API structure, is updated. To overcome this, we calculate a hash for the OpenAPI document and use that value as part of the resource ”id”.

[6]https://docs.microsoft.com/en-us/azure/azure-functions/consumption-plan

[7]https://registry.terraform.io/providers/hashicorp/google/4.25.0/docs/resources/api_api_config

Cost

AWS has free tiers for most available services for one year after creating the account. A free tier means that either the basic version of a service can be used indefinitely, or that there is a free (monthly) usage limit for the service. GCP and Azure have fewer services with a free tier, but compensate for it by giving users free credits to be spent in three months and one year, respectively.

The deployed application uses the free tiers of the services whenever possible. Using that, and the free credits, the monetary cost of deploying the applications and running the benchmark is zero. In Azure, all services consume credits except for the Functions and the Cloud Logging service. In GCP, the only service that consumes credits is Cloud SQL.

3.1.4 Benchmark execution

A mock client program is built to trigger the application processes. The application processes are executed 101 times. The results of the first execution are ignored due to the cold start delay of FaaS. Thus, only 100 executions are valid.

The program records the time for each request and response. A comma-separated values (CSV) file is generated with columns for all timestamps mentioned in Section 3.1.2. The columns for t_1 and t_7 are automatically filled as the program is running, while the rest have to be filled manually after the program finishes executing. The relation between the timestamps defined in Section 3.1.2 and the headers in the CSV file can be found in Appendix A.2.

The program waits for 10 seconds in between executions, i.e. after an execution finishes and before the next is triggered. The reason for this is twofold. First, it allows us to avoid mistakes in the manual process of matching platform logs to client request and response timestamps (described in Section 3.1.5) by bundling logs for each execution together. Second, it avoids some processes of the workflow executing for a specific duration due to the underlying implementation consisting of periodic processes. This is true for durations t_4t_8 and t_4t_9 of the application in Azure, due to the object storage trigger relying on polling the storage container for new objects at regular intervals.

3.1.5 Data collection

The timestamps defined in Section 3.1.2, expect for t_1 and t_7, are extracted from the platform logs. The collected data in the CSV file is published to Zenodo [21].

The logging service used in each platform is mentioned in Table 3.1. Logging services are used to monitor the system by recording events and application logs. To collect the timestamps, we query two sources of logs: API requests logs, and function execution logs. Timestamps t_2 and t_6 are taken from the API requests logs, and timestamps t_3, t_4, t_5, t_8, t_9, and t_{10} are taken from the respective function execution logs.

Each CSP requires us to query the logs in a different way. In AWS CloudWatch, we use the filters in the UI to query the data. In GCP Monitor, we use queries written in the Monitoring Query Language (MQL) to retrieve the logs. In Azure Cloud Logging, we use the filters in the UI but then intercept the response of the network request that retrieves the data. This step is required to get the precise timestamp for each log entry.

Once we find the logs, we export them to a file for further processing before filling in the CSV file created by the program in Section 3.1.4. We use simultaneous editing to separate the required timestamps from the other data, and manually normalize the format of the timestamps where needed. For t_6, the logs provide a duration with reference to t_2 instead of a timestamp. Thus, we calculate the timestamp ourselves.

3.1.6 Analyzing the collected data

We calculate the mean and standard deviation of the durations defined in section 3.1.2 for each platform. In addition, we calculate the cumulative distribution function (CDF) of the durations. Both help us discuss the variation in each platform's performance and between platforms.

3.2 Tools

The FaaS function code is written in the JavaScript programming language. The mock program used to trigger the application processes is also written in the JavaScript programming language. Data parsing, transformations, and analysis

are done in the Python programming language. The Visual Studio Code[8] editor is used to write the code. In addition, we use the editor plugins for Terraform[9], to write the infrastructure code, and Azure Functions[10], to test the functions' code locally.

[8]https://code.visualstudio.com/
[9]https://marketplace.visualstudio.com/items?itemName=HashiCorp.terraform
[10]https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurefunctions

Chapter 4

Results

The data collected from the experiment consists of timestamps for each step of the application processes. To present the collected data, several time intervals are defined as differences between two timestamps. The defined time intervals, or durations, provide useful insights for the performance of the application. In this chapter, the experiment results are presented in terms of the defined durations. The first section shows the average value and the spread of each duration, while the second section presents cumulative distribution function (CDF) plots of each duration.

4.1 Calculated durations

We calculate the duration values, as defined in Section 3.1.2, from the collected timestamps of each execution. Then, for each platform, we calculate the mean and standard deviation (STD) of each duration. These are presented in Table 4.1. The values for the mean and the STD are shown in seconds.

To interpret and compare the STD values, we calculate the coefficient of variation (COV) for each duration. This is also shown in Table 4.1. The COV is shown as a percentage and is calculated as follows:

$$COV = \frac{STD}{mean} \cdot 100\% \tag{4.1}$$

	AWS			Azure			GCP		
	Mean	STD	COV	Mean	STD	COV	Mean	STD	COV
t_1t_7	2.74	0.64	23.2%	2.52	0.54	21.2%	1.41	0.52	36.8%
t_2t_6	0.56	0.06	11.5%	1.69	0.55	32.6%	0.62	0.18	29.6%
t_1t_2	2.00	0.62	31.1%	0.69	0.15	21.6%	0.64	0.36	55.8%
t_6t_7	0.18	0.07	39.7%	0.14	0.08	52.3%	0.15	0.06	38.0%
t_2t_3	0.06	0.01	18.1%	1.29	0.54	42.2%	0.02	0.02	78.6%
t_3t_5	0.51	0.06	12.2%	0.41	0.05	11.6%	0.59	0.17	29.2%
t_5t_6	-0.01	0.01	53.1%	-0.01	0.01	152.3%	0.00	0.00	40.4%
t_4t_8	0.98	0.34	34.5%	5.28	2.95	55.8%	0.41	0.49	118.7%
t_8t_{10}	0.43	0.04	10.0%	0.19	0.02	9.3%	1.05	0.17	16.3%
t_4t_9	1.41	0.34	24.1%	5.47	2.95	54.0%	1.46	0.59	40.6%

Table 4.1: Mean, standard deviation (STD), and coefficient of variation (COV) of the durations defined in Section 3.1.2, for each platform, in seconds.

4.2 Empirical distribution of the durations

Cumulative distribution function (CDF) plots are composed for the defined durations. The CDF plots are organized into four categories based on the performance aspect that they represent. These categories are: synchronous process, network latency, triggering and execution of CreateUser, and triggering and execution of ProcessImage. In the following sections, we present the CDF plots for each category and the respective time durations they encompass.

For all plots, the x-axis represents time in seconds, and the y-axis represents the cumulative probability. The plot title indicates the duration it depicts. Furthermore, the values shown in the x-axis are the same for each plot. This helps when comparing the plots with each other.

4.2.1 Synchronous process

The durations related to the whole synchronous process are t_1t_7, and t_2t_6. The CDF plots for these durations are shown in Figure 4.1. t_1t_7 (left) shows the time it takes for the synchronous process to complete from the client's perspective. t_2t_6 (right) shows the time it takes for the platform to complete the synchronous process.

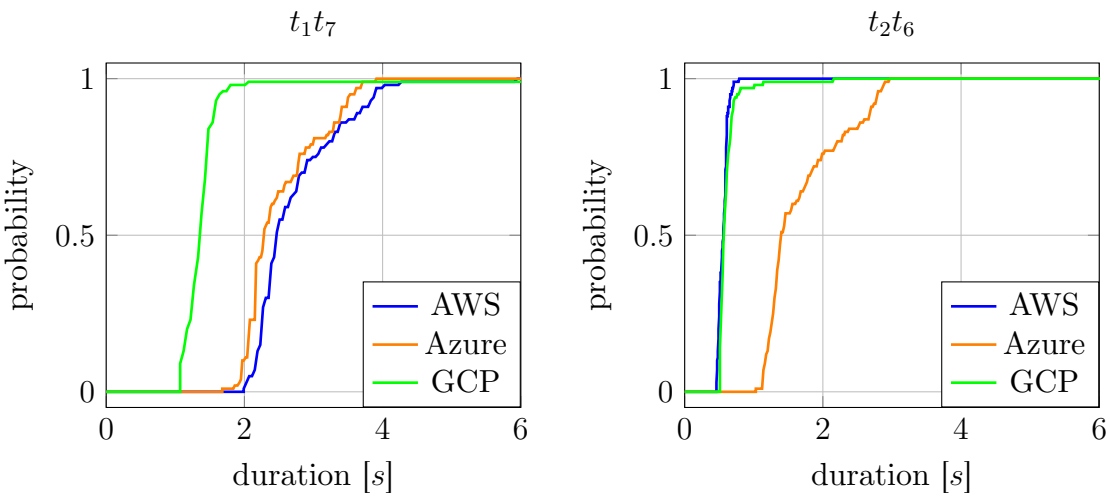

Figure 4.1: CDF plots of the durations related to the whole synchronous process.

4.2.2 Network latency

CDF plots of the durations related to the network latency between the client and the platform (t_1t_2, and t_6t_7) are shown in Figure 4.2. t_1t_2 (left) shows the time it takes for the request to be sent from the client to the API gateway. t_6t_7 (right) shows the time it takes for the client to receive the API gateway response.

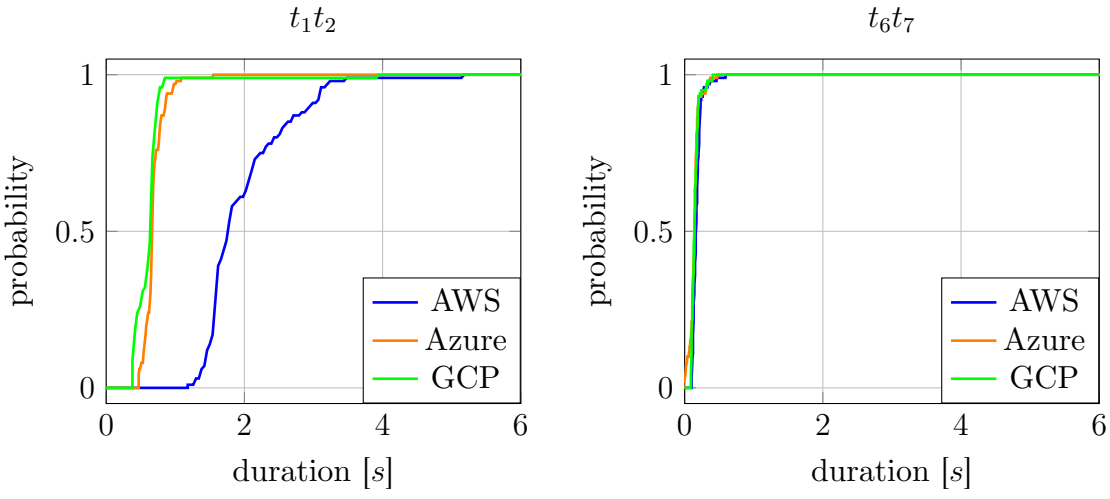

Figure 4.2: CDF plots of the durations related to the network latency between the client and the platform.

4.2.3 Triggering and execution of CreateUser

Figure 4.3 contains the CDF plots of the durations related to the triggering and execution of the CreateUser function (t_2t_3, t_3t_5, and t_5t_6). t_2t_3 (top-left) shows the time it takes for the API gateway to trigger the CreateUser function after receiving the client's request. t_3t_5 (top-right) shows the execution duration of the CreateUser function. t_5t_6 (bottom) shows the time it takes for the CreateUser function to pass the response to the API gateway.

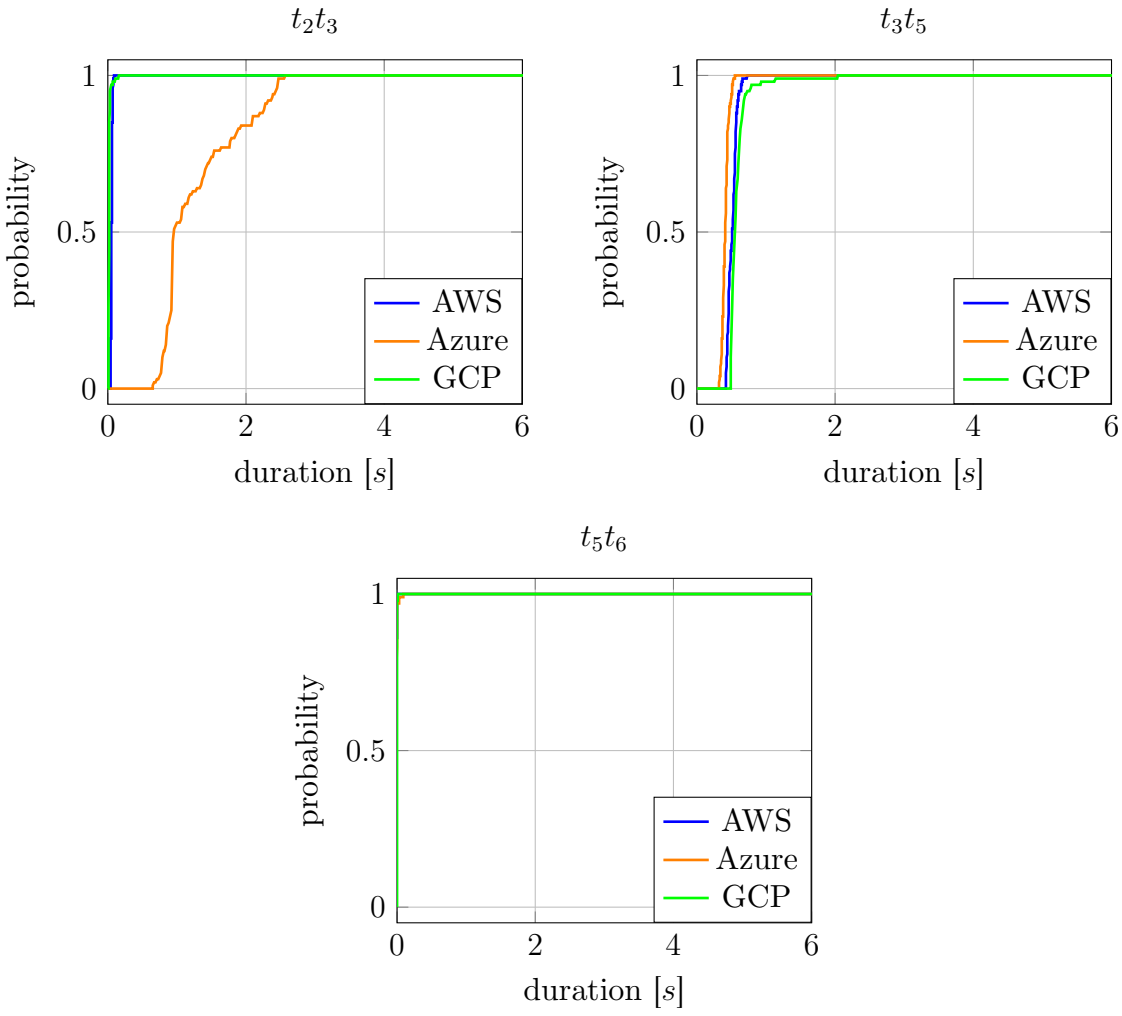

Figure 4.3: CDF plots of the durations related to the triggering and execution of the CreateUser function.

4.2.4 Triggering and execution of ProcessImage

The CDF plots of the durations related to the triggering and execution of the ProcessImage function (t_4t_8, t_8t_{10}, and t_4t_9) are presented in Figure 4.4. t_4t_8 (top-left) shows the time it takes for the platform to discover the uploaded picture and trigger the ProcessImage function. t_8t_{10} (top-right) shows the execution duration of the ProcessImage function. t_4t_9 (bottom) shows the time it takes for the thumbnail image to be generated from the time the original profile image is uploaded.

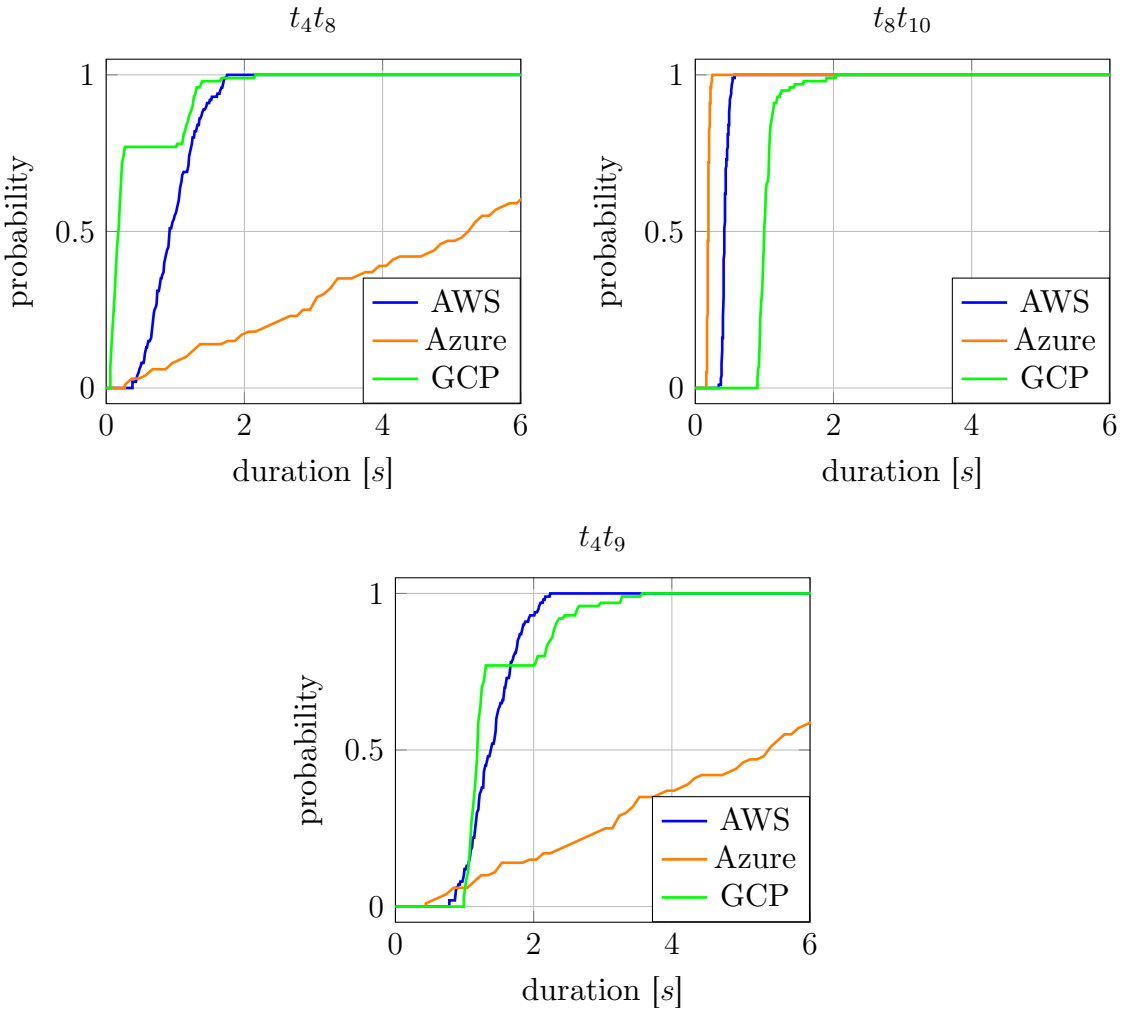

Figure 4.4: CDF plots of the durations related to the triggering and execution of the ProcessImage function.

Chapter 5

Discussion

In this chapter, we discuss the study results. We first answer the research questions, which in turn help us interpret the results of the experiment and validate our hypotheses. For each hypothesis, we consider both our results and results of other similar studies. In addition, we discuss the threats to validity, and the reproducibility of the study.

5.1 Answering the research questions

This section presents the answers to the research questions stated in Chapter 1. The first research question is related to the research method, and is answered in Chapters 2 and 3. The second and third question are directly related to the hypotheses and how we interpret the experiment results to validate them.

5.1.1 Research question 1

To define the application used for performance evaluation, we need to choose the CSPs to be evaluated, the application processes, the data to be collected, and the application execution plan.

To decide which CSPs are included in the benchmark (RQ 1.1), we look at the cloud computing market share. This is presented in Section 2.1.2. We only consider the top three CSPs due to their popularity, which affects documentation and tooling, and due to the increased effort needed to deploy and test the application in more CSPs.

To devise the application processes that are used to evaluate performance (RQ 1.2), we look at other performance evaluation studies, as well as use cases from

the real world. Most studies we refer to are from the list of studies that Scheuner and Leitner considered in their systematic review [17]. The use cases from the real world were taken from CSP websites, or other surveys, such as the one by Eismann et al. [19].

To decide which data points are collected (RQ 1.3), we look at the accessible logs on each platform. In Section 3.1.2, we set out to record two timestamps for each service execution, with the timestamps marking the beginning and the end of that service execution. This is, however, not possible for image uploads to object storage where a single timestamp to mark the upload time is reported. Moreover, the precision of the timestamp in the object storage service logs is lower than the other reported timestamps, with the time in the timestamp rounded to the nearest second. Thus, as the upload timestamp, we use the time in the upload response received in the respective function that triggers the upload.

To decide on the application execution plan (RQ 1.4), we consider the practicality of data collection. First, there are limits to how many results the CSPs' logging service shows at a time, especially when no keyword filters are used, and the logs in the UI either appear in small blocks (of 50 or 100 logs) or are loaded progressively as you scroll down the page. Second, when filling in the timestamps in the CSV file, we need to check that the timestamps are ordered correctly within their own column, and that all timestamps in a row belong to a single program execution. Third, since we use simultaneous editing to process the exported logs files, very large files can make the text editing software non-responsive.

5.1.2 Research question 2

We use consistency to talk about performance variation. To compare the consistency in performance for each platform, we use the calculated COV value of the durations in Section 4.1. The lower the COV, the more consistent the performance is. However, the value of the COV is influenced by the mean value. Thus, the mean value has to be considered as well when comparing the consistency of one platform to another. We consider COV values lower than 15% to signify high consistency in performance, values lower than 40% to signify moderate consistency, and other values to show low consistency.

5.1.3 Research question 3

To discuss the performance variation between platforms, we look at the mean and standard deviation value of the durations in Section 4.1, and the CDF plots in Section 4.2. We do not discuss performance consistency for durations that are dependent on the client's network latency, and the plots for those durations are presented only for the sake of completeness. We also compare our findings to other studies. Due to the different experiment setups, performance values from other studies cannot be compared directly. Thus, we look at how platforms perform compared to each other instead.

5.2 Validating the hypotheses

In this section, we validate the hypotheses. For each hypothesis, we discuss our results, draw conclusions from them, and then compare our findings to those of other studies from literature.

5.2.1 Hypothesis 1

For durations related to the triggering and execution of the CreateUser function, we look at durations t_2t_3, and t_3t_5. Duration t_5t_6 is very close to zero in all platforms, and a discussion of performance consistency is not applicable. Regarding the triggering of the function, performance is most consistent in AWS, with a COV of 18.1%, followed by Azure and GCP. Azure has a COV of 42.2%, which is quite high considering the mean value for the duration. While in GCP, even though the COV is 78.6%, the mean value for the duration is extremely low, at only 0.02 seconds. Regarding the execution duration of the function, performance is highly consistent in AWS and Azure, with COV values of around 10%, while it is slightly worse in GCP, with a COV of 29.2%.

For durations related to the triggering and execution of the ProcessImage function, we look at durations t_4t_8, and t_8t_{10}. Duration t_4t_9 is the sum of the two previous durations. Regarding the triggering of the function, performance is fairly consistent in AWS, with a COV of 34.5%, and highly inconsistent in GCP, with a COV of 118.7%. A discussion of consistency for this duration is not applicable for the performance in Azure due to the trigger relying on polling instead of being an event-based trigger. Regarding the execution duration of the

function, performance is highly consistent in all three platforms, with COV values of 10% to 15%.

Conclusion

To determine performance variation in each platform, we classify their performance consistency according to the criteria defined in Section 5.1.2. This is presented in Table 5.1. For each function, we classify consistency for the function trigger duration and the function execution duration. The function trigger consistency in all platforms is moderate to low. The function execution consistency is high in all platforms.

	Sync. Function		Async. Function	
	Trigger	Execution	Trigger	Execution
AWS	moderate	high	moderate	high
Azure	low	high	N/A	high
GCP	low	moderate	low	high

Table 5.1: Consistency classification of the trigger duration and execution duration for both synchronous and asynchronous functions in each platform.

Results from literature

There is a study, described in Section 2.2.2, that compares the consistency in performance of AWS and GCP. The authors of the study found that performance of FaaS function in GCP was more stable than those in AWS. In our study, however, we find the opposite.

5.2.2 Hypothesis 2

For durations related to the triggering and execution of the CreateUser function, we look at durations t_2t_3, t_3t_5, and t_5t_6. The function is triggered almost instantaneously in AWS and GCP, with the duration being less than 0.1 seconds. In Azure, however, triggering roughly takes 1 to 2 seconds. The execution duration of the functions, on the other hand, is similar in all platforms, with the duration being around 0.5 seconds. Lastly, passing the response of the function back to the API gateway is instantaneous in all platforms, with the duration being almost zero.

For durations related to the triggering and execution of ProcessImage function, we look at durations t_4t_8, t_8t_{10}. The function is triggered the fastest in GCP, with a duration ranging from 0 to 1 seconds. In AWS, the function trigger takes around 1 second. In Azure, the trigger duration forms a uniform distribution with a range of 0 to 10 seconds. This is due to the trigger mechanism, which uses polling with an interval of 10 seconds. The execution duration of the function, however, is the lowest in Azure, with a duration of 0.2 seconds. The function execution takes twice as long in AWS, and five times as long in GCP.

To discuss the synchronous process as a whole, we look at duration t_2t_6. AWS and GCP perform similarly, with the duration of the process being around 0.5 seconds. The duration of the process in Azure, however, is two to four times higher. For the asynchronous process, duration t_4t_9, we find that GCP performs the best, and AWS performs almost as well. The duration of the asynchronous process in Azure is not comparable to that in the other platforms due to the different triggering mechanism of the ProcessImage function.

Conclusion

To decide whether there are performance differences between the platforms, we look at the durations of the functions' trigger and functions' execution. These durations are summarized in Table 5.2. We find that the function trigger durations are much higher in Azure compared to those in the other platforms. The function execution durations, on the other hand, are almost the same in all platforms.

	Sync. Function		Async. Function	
	Trigger	Execution	Trigger	Execution
AWS	0.06 ± 0.01	0.51 ± 0.06	0.98 ± 0.34	0.43 ± 0.04
Azure	1.29 ± 0.54	0.41 ± 0.05	5.28 ± 2.95	0.19 ± 0.02
GCP	0.02 ± 0.02	0.59 ± 0.17	0.41 ± 0.49	1.05 ± 0.17

Table 5.2: The trigger duration and execution duration for both synchronous and asynchronous functions in each platform in seconds.

Results from literature

We consider the study in Section 2.2.1 to compare the trigger duration of the CreateUser function. The authors of the study found that AWS and Azure had a similar trigger duration, while the trigger in GCP took twice as long. In our

findings, we find AWS and GCP to have a very short trigger duration, while Azure has a duration that is 10 to 20 times longer.

The study described in Section 2.2.2 compares the execution duration of a function in AWS and GCP. In their study, the authors found that the function in AWS had lower execution times than the functions in GCP. In our study, we also find that the functions in AWS have a lower execution duration than those in GCP. There is a slight difference in the durations of the CreateUser function, but a significant difference in those of the ProcessImage function.

To compare our results to the study in Section 2.2.3, we look at the measured durations of the synchronous process. The study found that the duration in Azure was twice as long as that of AWS. In our study, we also see that the duration of the synchronous process as a whole in Azure was two to four times longer than that in AWS.

5.3 Threats to validity

This section presents the threats to the validity of the study and our attempts to mitigate them. The threats to validity are separated into three categories: construct validity, internal validity, and external validity [1]. Construct validity is about correctly interpreting and measuring the concepts that are set out to be measured. Internal validity is about the validity of the results that are generated by the study. External validity is about generalizing and comparing the results of the study to other studies.

5.3.1 Construct validity

For the purposes of the study, we need to make sure that our application resembles real production applications. Thus, to design the application for the benchmark, we refer to FaaS use cases mentioned on CSPs' websites, as well as other studies that perform FaaS benchmarks [9], [14]. A drawback of our application is that it uses very few cloud services. However, a larger application that spans more services would increase both the application deployment time and the data collection efforts.

5.3.2 Internal validity

The services or service tiers used for the application are not identical among CSPs. This is unavoidable since each platform develops their services and service configurations independently and there are no standards at large. To mitigate such effects, we try to match the configurations between CSPs to their best extent. These configurations are explained in Section 3.1.3.

The application database is publicly accessible. This is a deviation from an application in production, where the database would not be accessible from the internet, and is done for practical reasons. Apart from the CreateUser function which needs access to the database, we also need access to the database to create the required tables for the application after the database is deployed.

The application API and FaaS functions can be accessed without any form of authorization. These are also deviations from an application in production and are done to reduce the deployment effort of an application with proper authorization configuration. To the best of our knowledge, having authorization configured does not impact performance in any meaningful way.

The manual collection and processing of the experiment data is prone to human error. To avoid such mistakes, we add a delay of 10 seconds between each benchmark run, to be able to examine the timestamps. However, at the same time, the delays of 10 seconds may eliminate some concurrency effects which would be present in the original application. Thus, to completely mitigate these threats, in the future, we need to develop tools to collect and process the data at each step.

5.3.3 External validity

There might not be enough measurements to make reasonable conclusions. The number of measurements is limited due to the manual process of data collection (described in Section 3.1.5) and the cost of running the infrastructure (described in Section 3.1.3). The current number of measurements is decided through a trial-and-error process that examines the relation between the number of measurements and the variability of the results.

5.4 Reproducibility principles

In this section, we list the eight reproducibility principles for performance evaluation in cloud computing by Papadopoulos et al., and discuss how or whether each principle is applied in our experiment [20].

P1: *Repeated experiments. Decide how many repetitions with the same configuration of the experiment should be run, and then quantify the confidence in the final result.*

We run the application processes 100 times in each platform and measure all the relevant properties defined in the benchmark. A target population is not defined, and the sample size of the experiment is chosen such that the most measurements are made while it is still practical to collect the experiment data (see Section 5.1.1). Aggregated values (mean and standard deviation) for the durations specified in the benchmark are calculated.

P2: *Workload and configuration coverage. Experiments should be conducted in different configurations of relevant parameters.*

We test the application with a single type of workload. Due to the benchmark being an application-level benchmark, we do not consider different configurations of the services that make up the application.

P3: *Experimental setup description. Description of the hardware and software setup used to carry out the experiments, and of other relevant environmental parameters, must be provided. This description should include the operating system and software versions, and all the information related to the configuration of each experiment.*

We mention the relevant application configurations for each platform in Section 3.1.3. In addition, the infrastructure code provides a complete description of the deployed application (the cloud resources used, resource configurations, environment variables, etc.).

P4: *Open access artifact. At least a representative subset of the developed software and data (e.g., workload traces, configuration files, experimental protocol, evaluation scripts) used for the experiment should be made available to the scientific community.*

The configuration files used to deploy the infrastructure in each platform are published. The benchmark results are also published.

P5: *Probabilistic result description. Report a full characterization of the empirical distribution of the measured performance, including aggregated values and variations around the aggregation, with the confidence that the results lend to these values.*

The mean and standard deviation for all durations specified in the benchmark are calculated. The empirical distribution of each duration is also visualized with a CDF plot.

P6: *Statistical evaluation. When comparing different approaches, provide a statistical evaluation of the significance of the obtained results.*

Since there are no quantitative decisions mentioned in the hypotheses, we do not perform any statistical tests.

P7: *Measurement units. Report the corresponding unit of measurement for all reported quantities.*

The CSV file with the experiment results (mentioned in Section 3.1.4) specifies a measurement unit in the header of each column. In the tables, the numerical values are associated by the corresponding unit, or the units are reported in the table caption. For plots, the units are shown in the axis labels.

P8: *Cost. The experiment should include the cost model used or assumed for the experiment and the actual charged cost.*

We assumed that the free service tiers and free credits awarded by the platform would cover the expenses of running the infrastructure and we did not calculate the cost of the experiment beforehand. The resulting cost is described in Section 3.1.3. The credits spent in Azure and GCP were $74.48 and $21.43, respectively.

Chapter 6

Conclusion

The study aims to evaluate the performance of different Function as a Service (FaaS) platforms. An application-level benchmark is performed in Amazon Web Services (AWS), Azure, and Google Cloud Platform (GCP). The application consists of two processes, motivated from real-world use cases. The first process creates a user account with a profile picture, while the second process creates a thumbnail of the uploaded profile picture. The first process is synchronous, and the second process is asynchronous.

The application infrastructure is defined as code with Terraform. The results of the benchmark constitute of timestamps for each part of the executing processes. Time durations are calculated from the measured timestamps and cumulative distribution function (CDF) plots are created. These are used to validate the two proposed hypotheses.

In the first hypothesis, we propose that the performance in each cloud services provider (CSP) varies between consecutive runs. To validate this hypothesis, we quantify the consistency in performance in each platform. Overall, we find that the function trigger consistency is moderate, while the function execution consistency is high in all platforms. For both the function trigger and function execution, the consistency is the highest in AWS.

In the second hypothesis, we propose that performance differs between CSPs. To validate the hypothesis, we compare function trigger duration and execution duration across platforms. We find a significant difference in function trigger duration between AWS and GCP on one hand, and Azure on the other. The trigger duration in Azure is considerably longer compared to those in the other platforms. For function execution, we find that all platforms have similar durations.

When looking at the application as a whole, we find that AWS and GCP perform similarly, and Azure lags behind. We may draw these conclusions since performance in all platforms is reasonably consistent. Our conclusions are in line with some, but not all, of the other research findings. Although, it is true that the studies' research methodologies and configurations vary greatly.

6.1 Future work

To further validate the results of this study, it is recommended to re-run the benchmark using the reproducibility package that is provided. The current study fails to discuss the performance of the function triggering in Azure. Thus, a critical improvement to the benchmark is to use the appropriate event-based Azure storage trigger. Additionally, to improve the infrastructure creation process, the manual steps involved in the creation of the database tables could be automated.

To improve FaaS benchmarking, it is advisable to expand the testing coverage and extent. This can be achieved by incorporating testing on a wider range of cloud platforms, including Alibaba Cloud and IBM Cloud, in addition to the current platforms. Furthermore, adding more processes to the benchmark application will provide a better understanding of how well each platform performs in different real-life situations.

Finally, a general challenge in performing benchmarks in public clouds is the impracticality of collecting data from logs. While each CSP provides performance metrics in some form, these are not as flexible as user-generated logs. Thus, improving data collection from logs would enable researchers to run benchmarks more extensively. This is also a necessary step towards standardizing a benchmark.

Bibliography

[1] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting Empirical Methods for Software Engineering Research," en, in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds., London: Springer, 2008, pp. 285–311, ISBN: 978-1-84800-044-5. DOI: 10.1007/978-1-84800-044-5_11. [Online]. Available: https://doi.org/10.1007/978-1-84800-044-5_11 (visited on 09/11/2022).

[2] M. Armbrust, A. Fox, R. Griffith, *et al.*, "A view of cloud computing," en, *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1721654.1721672. [Online]. Available: https://dl.acm.org/doi/10.1145/1721654.1721672 (visited on 09/09/2022).

[3] I. Baldini, P. Castro, P. Cheng, *et al.*, "Cloud-Native, Event-Based Programming for Mobile Applications," in *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, May 2016, pp. 287–288.

[4] D. Potes and A. Nair, *AWS re:Invent 2016: Building Complex Serverless Applications (GPST404)*, Nov. 2016. [Online]. Available: https://www.slideshare.net/AmazonWebServices/aws-reinvent-2016-building-complex-serverless-applications-gpst404 (visited on 09/08/2022).

[5] G. Adzic and R. Chatley, "Serverless computing: Economic and architectural impact," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, New York, NY, USA: Association for Computing Machinery, Aug. 2017, pp. 884–889, ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3117767. [Online]. Available: https://doi.org/10.1145/3106237.3117767 (visited on 08/06/2022).

[6] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research," 2017, arXiv:1708.08028 [cs]. DOI: 10.13140/RG.2.2.15007.

87206. [Online]. Available: http://arxiv.org/abs/1708.08028 (visited on 07/31/2022).

[7] G. McGrath and P. R. Brenner, "Serverless Computing: Design, Implementation, and Performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, ISSN: 2332-5666, Jun. 2017, pp. 405–410. DOI: 10.1109/ICDCSW.2017.36.

[8] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving Deep Learning Models in a Serverless Platform," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, Apr. 2018, pp. 257–262. DOI: 10.1109/IC2E.2018.00052.

[9] H. Lee, K. Satyam, and G. Fox, "Evaluation of Production Serverless Computing Environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, ISSN: 2159-6190, Jul. 2018, pp. 442–450. DOI: 10.1109/CLOUD.2018.00062.

[10] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold Start Influencing Factors in Function as a Service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, Dec. 2018, pp. 181–188. DOI: 10.1109/UCC-Companion.2018.00054.

[11] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, Nov. 2019, ISSN: 0001-0782. DOI: 10.1145/3368454. [Online]. Available: https://doi.org/10.1145/3368454 (visited on 05/30/2022).

[12] S. Fouladi, F. Romero, D. Iter, *et al.*, "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers," en, 2019, pp. 475–488, ISBN: 978-1-939133-03-8. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/fouladi (visited on 05/30/2022).

[13] V. Giménez-Alventosa, G. Moltó, and M. Caballer, "A framework and a performance assessment for serverless MapReduce on AWS Lambda," en, *Future Generation Computer Systems*, vol. 97, pp. 259–274, Aug. 2019, ISSN: 0167-739X. DOI: 10.1016/j.future.2019.02.057. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18325172 (visited on 08/01/2022).

[14] C. Ivan, R. Vasile, and V. Dadarlat, "Serverless Computing: An Investigation of Deployment Environments for Web APIs," en, *Computers*, vol. 8, no. 2, p. 50, Jun. 2019, Number: 2 Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 2073-431X. DOI: 10.3390/computers8020050. [Online]. Available: https://www.mdpi.com/2073-431X/8/2/50 (visited on 05/30/2022).

[15] A. Palade, A. Kazmi, and S. Clarke, "An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge," May 2019. DOI: 10.1109/SERVICES.2019.00057.

[16] M. Zhang, Y. Zhu, C. Zhang, and J. Liu, "Video processing with serverless computing: A measurement study," in *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, ser. NOSSDAV '19, New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 61–66, ISBN: 978-1-4503-6298-6. DOI: 10.1145/3304112.3325608. [Online]. Available: https://doi.org/10.1145/3304112.3325608 (visited on 07/31/2022).

[17] J. Scheuner and P. Leitner, "Function-as-a-Service performance evaluation: A multivocal literature review," en, *Journal of Systems and Software*, vol. 170, p. 110 708, Dec. 2020, ISSN: 0164-1212. DOI: 10.1016/j.jss.2020.110708. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301527 (visited on 06/27/2022).

[18] J. Beswick, *Operating Lambda: Performance optimization – Part 1 | AWS Compute Blog*, en-US, Section: AWS Lambda, Apr. 2021. [Online]. Available: https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1/ (visited on 09/11/2022).

[19] S. Eismann, J. Scheuner, E. van Eyk, *et al.*, "A Review of Serverless Use Cases and their Characteristics," arXiv, Tech. Rep. arXiv:2008.11110, Jan. 2021, arXiv:2008.11110 [cs] type: article. DOI: 10.48550/arXiv.2008.11110. [Online]. Available: http://arxiv.org/abs/2008.11110 (visited on 06/01/2022).

[20] A. V. Papadopoulos, L. Versluis, A. Bauer, *et al.*, "Methodological Principles for Reproducible Performance Evaluation in Cloud Computing," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1528–1543, Aug.

2021, Conference Name: IEEE Transactions on Software Engineering, ISSN: 1939-3520. DOI: 10.1109/TSE.2019.2927908.

[21] B. Hamiti, *Resources for "Function-as-a-Service Performance Evaluation with Application-level Workflows"*, version 0.1, Zenodo, Sep. 2022. DOI: 10.5281/zenodo.7112754. [Online]. Available: https://doi.org/10.5281/zenodo.7112754.

[22] N. Reno, *Q2 Cloud Market Grows by 29% Despite Strong Currency Headwinds; Amazon Increases its Share*, Jul. 2022. [Online]. Available: https://www.srgresearch.com/articles/q2-cloud-market-grows-by-29-despite-strong-currency-headwinds-amazon-increases-its-share (visited on 10/19/2022).

[23] "Flexera 2022 State of the Cloud Report," Tech. Rep. [Online]. Available: https://www.flexera.com/blog/cloud/cloud-computing-trends-2022-state-of-the-cloud-report/.

Appendix A

Experiment details

A.1 Application database

The following statement is used to create the database table used in the application:

```
CREATE TABLE `seeu_db`.`users` (
  `user_id` int(11) NOT NULL AUTO_INCREMENT,
  `email` varchar(90) NOT NULL,
  `password` varchar(255) NOT NULL,
  `name` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`user_id`),
  UNIQUE KEY `email_UNIQUE` (`email`)
);
```

A.2 Application benchmark results

Table A.1 shows the relation between the headers of the CSV file where experiment results are collected, and the timestamps defined in Section 3.1.2.

File header	Timestamp
$client\_request\_timestamp$	t_1
$client\_response\_timestamp$	t_7
$apiGateway\_request\_timestamp$	t_2
$apiGateway\_response\_latency\_ms$	t_6
$createUser\_start\_timestamp$	t_3
$createUser\_end\_timestamp$	t_5
$profileImage\_upload\_timestamp$	t_4
$processImage\_start\_timestamp$	t_8
$processImage\_end\_timestamp$	t_{10}
$thumbnail\_upload\_timestamp$	t_9

Table A.1: Relation between the results' file headers and the timestamps defined in the thesis document.