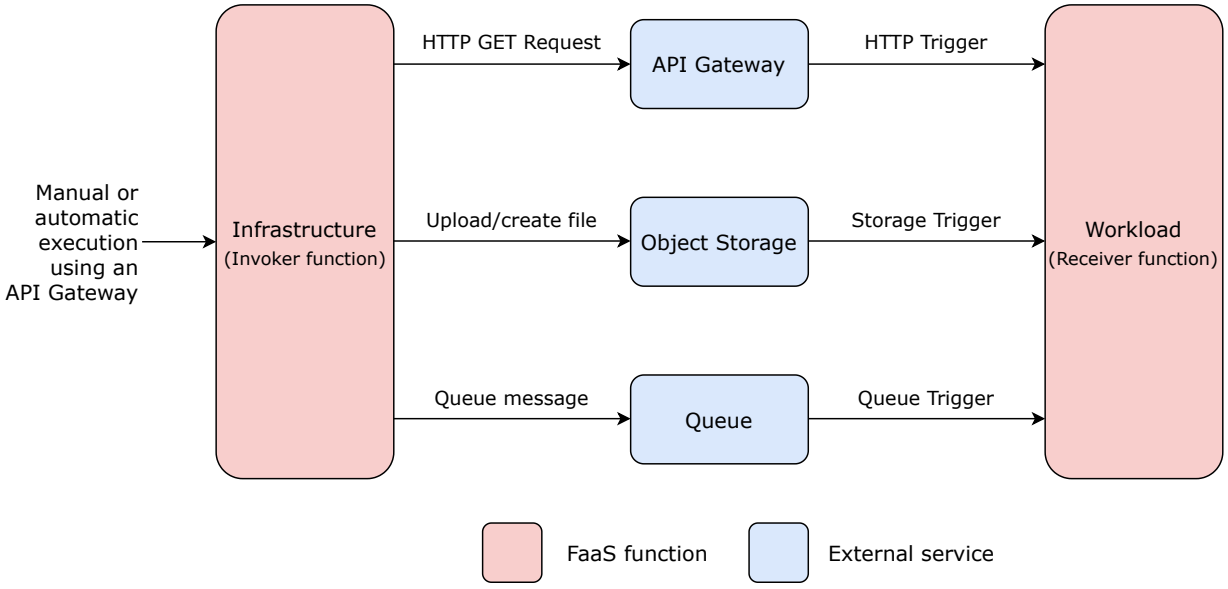
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Performance Comparison of Function-as-a-Service Triggers

A Cross-Platform Performance Study of Function Triggers in Function-as-a-Service

Master's thesis in Computer science and engineering

Marcus Bertilsson & Oskar Grönqvist

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Performance Comparison of Function-as-a-Service Triggers

A Cross-Platform Performance Study of Function Triggers in Function-as-a-Service

MARCUS BERTILSSON & OSKAR GRÖNQVIST

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Performance Comparison of Function-as-a-Service Triggers

A Cross-Platform Performance Study of Function Triggers in Function-as-a-Service

MARCUS BERTILSSON & OSKAR GRÖNQVIST

Supervisor: Joel Scheuner, Department of Computer Science and Engineering
Examiner: Jennifer Horkoff, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A high-level model of the benchmark architecture/structure.

Typeset in LaTeX
Gothenburg, Sweden 2021

Performance Comparison of Function-as-a-Service Triggers

A Cross-Platform Performance Study of Function Triggers in Function-as-a-Service

MARCUS BERTILSSON & OSKAR GRÖNQVIST
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Cloud computing paved the way for how servers are handled and maintained, and recent developments in cloud computing have established a new paradigm shift towards serverless computing. Through a subset called Function-as-a-Service (FaaS), most operational concerns are abstracted away and allows developers to focus entirely on the code (i.e. functions) to be executed. FaaS functions are triggered by events (called triggers) and there are many types of triggers offered by each provider. This thesis studied the latency of three trigger types through a trace-based approach. The three triggers were HTTP triggers, storage triggers, and queue triggers. To further contrast previous work, the comparisons were also made across two providers, Amazon Web Services (AWS) and Microsoft Azure. Focus was also put on discussions justifying the comparison between two largely different providers and on the reproducibility of the study. The HTTP trigger performed the best for both providers, the Queue trigger second-best for AWS and third-best for Azure, and the Storage trigger third-best for AWS and second-best for Azure. In terms of providers, both performed relatively similarly in terms of mean delay but Microsoft Azure had significantly more extreme outliers compared to Amazon Web Services. In conclusion, the study performed in this thesis found that the choice of service and provider can greatly affect a system's performance and can, in extension, affect the usage of cloud services.

Keywords: Computer science, engineering, master thesis, serverless, cloud, FaaS, trigger, performance.

Acknowledgements

We want to thank our supervisor Joel for his hard work in keeping us on course throughout the thesis and for his experience in, and dedication to, the subject. His help has been invaluable to us. Finally, we want to thank the people in the Pulumi Slack channel who could give us help with technical details when no one else could.

Marcus Bertilsson, Oskar Grönqvist. Gothenburg, June 2021.

Contents

List of Figures

List of Tables

1

Introduction

Cloud computing has enabled the entire software industry to shift away from locally maintained servers to massive data centers provided by some of the largest companies known today. Infrastructure-as-a-Service (IaaS), the most basic cloud computing paradigm, paved the way for this shift by offering customers Virtual Machines (VMs) through a pay-as-you-go model. A more recent cloud computing paradigm, known as Serverless computing has emerged and started to be offered by major cloud providers through a subset called Function-as-a-Service (FaaS). Unlike IaaS, FaaS enables developers to focus solely on the actual code (i.e. functions) to be executed, where all operational concerns are handled by the provider. The execution of these functions can be triggered by different types of events (such as HTTP requests and changes in a database), and these events are referred to as triggers. When a trigger event happens, the cloud provider executes the requested functions and automatically scales the underlying virtual machines to accommodate the workload [5]. FaaS is used in a variety of different applications, for example, to transform images, handle notifications, or implement REST endpoints. Despite being able to serve a wide range of applications, FaaS is not a one-size-fits-all solution. Latency-sensitive applications are examples of problematic use cases for FaaS. This is also partly due to limitations such as cold start overheads, which occur when the requested function is inactive and there are no available containers to run it in [6].

Scheuner and Leitner [7] concluded, in their literature review on performance evaluations on serverless services, that existing research in this area is mostly limited to Amazon Web Services (AWS) with a heavy focus on simplified, often CPU-intensive benchmarks called micro-benchmarks. The authors suggested a need to focus more on realistic workloads with respect to other characteristics such as network or function trigger performance. Moreover, further research could also preferably study these aspects across platforms, to enable easier comparison in a relatively new and unexplored field.

1.1 Purpose

The purpose of this thesis is to research, design, and implement a benchmarking framework to allow for performing cross-platform benchmarks of trigger performance with respect to latency. By performing cross-platform benchmarks focusing on trig-

ger performance rather than single platform evaluations using micro-benchmarks for the CPU, this thesis aims to address some of the gaps left by previous research. However, since the performance of the tested services can vary greatly depending on actions taken by the providers, a strong focus is put on reproducibility through the eight principles developed by Papadopoulos et al. [8], which will be explained further in Section 3.2. This is done to ensure that future researchers can replicate the experiments conducted in this thesis and thus promote future work in this field.

The ultimate goal of this thesis is to provide a better understanding of FaaS services and their performance and in turn encourage comparison and selection of services and providers, which could influence future design and architecture decisions. More specifically, our study will provide a measurement tool, a data set, and performance evaluations.

1.2 Research Questions

While serving similar purposes, there is a heterogeneous nature to cloud services due to their separate development by different providers. This makes it difficult to develop a benchmark to work as a one-size-fits-all solution, as some services might perform differently in different areas of use. While designing a benchmark it is important to take the heterogeneous nature of the services into account in order to make it fair and thus minimize bias. FaaS platforms also abstract away much of the operational logic from the user which limits research about how internal and external factors impact performance [9]. In addition to the above, a large portion of the total delay will be from delays outside of the service provider's jurisdiction, i.e. network delays due to the distance between end-users and the service provider's server halls. The extent to which these delays can be ignored depends on the quality of the service provider's internal logging tools.

To achieve the purpose behind this thesis we define three research questions we want to answer.

RQ1: How can function trigger latency be measured across different serverless platforms?

This question touches upon the problem of the heterogeneous nature of the services as a result of their separate development. It is unlikely that a one-size-fits-all solution will be found and this raises the question of how to make each approach comparable to the other.

RQ2: How does the choice of trigger type affect trigger latency?

This question is motivated by the gaps left in previous research, where there has been a heavy focus on CPU and memory benchmarks and not enough focus on other aspects. All major cloud providers support a list of different trigger types, each with its own characteristics that may impact performance.

RQ3: How does the choice of provider affect trigger latency?

This question is similar to RQ2 but is motivated by another gap in previous research, namely the lack of research into performance differences between providers. The same type of service provided by different providers is often only similar in their purpose and technical details can, therefore, make their performance vary considerably.

1.3 Limitations and delimitations

Although it would be preferable to use realistic workloads, this study is using workloads smaller than in micro-benchmarks and instead focuses on other aspects. By taking the practical steps discussed by Scheuner and Leitner [7] into account, as well as using Infrastructure-as-Code (IaC) to define and report our experiments, we address the challenges related to reproducibility and ensure that our study is reproducible under similar conditions.

Despite there being several cloud providers, this thesis focuses on cross-platform comparisons between Amazon Web Services and Microsoft Azure. This is partly due to the limited time frame, but also because Amazon and Azure are the two single largest providers in terms of market share. As of the second quarter of 2020, Amazon and Microsoft had 33 and 18 percent of the market share respectively, together making up a majority of 51 percent of the total market share [3]. In terms of trigger types, this thesis studies three of them, namely HTTP, object storage, and queue triggers. The selection of triggers is motivated by their popularity and usage in real cloud applications [10, Fig. 3].

2

Background

This chapter presents the necessary background needed to understand the thesis and introduces the reader to the most important topics. These topics include cloud computing, the transition to serverless computing, important paradigms such as Function-as-a-Service (FaaS), Infrastructure-as-Code (IaC), and distributed tracing.

2.1 Cloud Computing

Cloud computing became increasingly popular during the 2010s and Castro et al. argue it has now reached a stage where it is widely considered to be a paradigm shift in software engineering [5]. More specifically, it is a paradigm shift away from personal computing and locally maintained servers to distributed workloads "on the cloud", i.e., in centralized data centers. In some ways, the shift could be considered a return to centralized, rather than decentralized, computing which was the norm before personal computing replaced it in the 1990s [11].

Castro et al. [5] further describe the weaknesses of traditional cloud computing, and Infrastructure-as-a-Service (IaaS) in particular, that subsequently gave rise to the paradigm of serverless computing. This transition evolved from necessity as micro-service software architecture became increasingly popular in business. The authors also state that the principles of serverless computing make it "closer to [the] original expectations for cloud computing to be treated like as [*sic*] a utility" [5], meaning that serverless computing in many ways represent what cloud computing was meant to be. The weaknesses of cloud computing and the principles of serverless computing mentioned will be described in further detail later in this section.

When cloud computing was first introduced, it had many advantages over the existing client-server model. It required little upfront cost for businesses compared to the financial investment of setting up their own servers and it offered a pay-as-you-go financial model meaning that the client only pays for the resources consumed or execution time used [1]. But perhaps most importantly, it offered relatively simple ways to scale the hardware dedicated to certain tasks depending on the current workload. From the fundamental idea of cloud computing grew three different models with three different levels of abstraction, each service suitable for a different purpose. From the lowest to the highest level of abstraction, these three services are commonly known as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS),

and Software-as-a-Service (SaaS). Figure 2.1 shows a visual representation of IaaS, PaaS, and SaaS based on a figure by Bhardwaj et al. The infrastructure forms the foundation of both PaaS and SaaS, and the platform forms part of the foundation of SaaS. The position of each service in the diagram shows its level of abstraction.

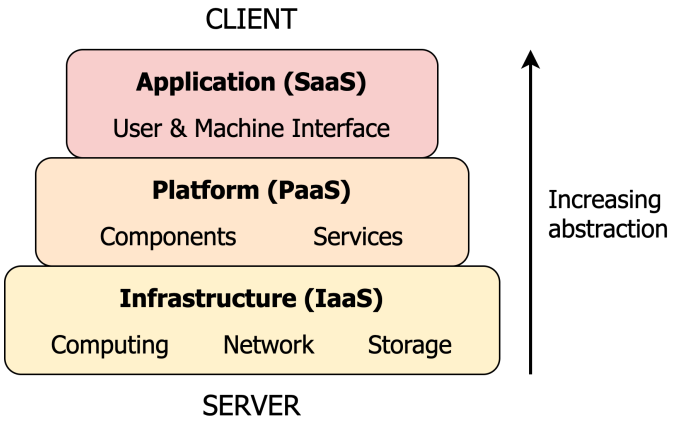


Figure 2.1: A visual representation of the relationship between the cloud computing services IaaS, PaaS, and SaaS. Based on Bhardwaj et al. [1, Fig. 2]

Infrastructure-as-a-Service is perhaps the most common service of the three and also offers the lowest level of abstraction. The idea of IaaS is that a host offers a hardware configuration and its appropriate software, e.g. servers, storage, network, and operating systems. By choosing this service, both the platform and the software have to be developed by the client, and only the physical hardware is provided by the host [1].

Platform-as-a-Service builds upon the configuration that IaaS offers plus a few certain additional software services to build a platform on which other software and applications can be run. The additional software added by PaaS could include software that further integrates the hardware offered by IaaS to build certain default functions that can be used on the platform [1].

Software-as-a-Service builds upon what PaaS offers but adds the software that the client would otherwise develop themselves. This makes the system less configurable and customizable, but instead requires little to no programming and is as close to Commercial-off-the-Shelf as cloud computing gets. The software offered by SaaS can be highly customizable, however, even if the system is not, but SaaS packages are often made for relatively specific purposes [1].

2.2 Serverless Computing

In many ways, serverless computing is the logical next step for cloud computing as an increasing number of businesses have started using microservices in application architectures [5]. Serverless computing offers improvements to the pay-as-you-go model, such as decreasing the time to start and stop a service, and further increases

the scalability of resources available to a service, commonly known as elasticity. Castro et al. [5] argue that these changes evolved naturally as microservice architectures have further requirements to be run efficiently. Serverless computing also further abstracts its use, often allowing users to utilize its services without any knowledge of programming, extending its potential client reach.

The most important service in serverless computing is Function-as-a-Service (FaaS). The usage of FaaS is based around a function that is responsible for some microservice, which in turn has been triggered by some event. There are often a large number of possible events that trigger a function, for example, HTTP requests, file uploads to storage, or events from time intervals based on a schedule. The most important benefit of using FaaS over other alternatives is that no server management is needed. All applications can utilize a high degree of elasticity, meaning that the resources consumed can be scaled from zero to any degree, and the client is only charged for the resources or execution time used.

2.3 FaaS Triggers

The invocation of a function in a FaaS application is always triggered by a certain event and these events are simply referred to as triggers [5]. Most cloud providers include multiple trigger types to fit customers' needs and some of these trigger types are highlighted in Table 2.1. The process of triggering a function is conceptually very simple. It begins with a trigger event being registered. In response to this event, the cloud provider will automatically allocate the right amount of computing resources to accommodate the workload and then execute the function. In serverless applications, triggers control the flow of the system and are responsible for connecting components to each other. Therefore, triggers pose a vital role where the choice of trigger type can affect large parts of the application.

There are many types of triggers but this thesis studies three of them, HTTP triggers, object storage triggers, and queue triggers. A trigger's type is based on the type of event that will trigger it. An HTTP request will trigger an HTTP trigger, a file upload or update will trigger an object storage trigger, and queuing a message will trigger a queue trigger. Aside from triggers reacting to different types of events, they also differ in how they transfer data and communicate. For AWS there are three distinct categories of triggers based on how they communicate; synchronous triggers, asynchronous triggers, and poll-based triggers. Looking at Table 2.1, API Gateway is an HTTP trigger but also a synchronous trigger, S3 is an object storage trigger but also an asynchronous trigger and SQS is a queue trigger but also a poll-based trigger [12]. The triggers of the other providers follow a similar categorization.

	HTTP	**Object Storage**	**Queue**
Amazon Web Services	API Gateway	S3	SQS
Microsoft Azure	HTTP Trigger	Blob Storage	Queue Storage
Google Cloud Platform	HTTP Trigger	Cloud Storage	Pub/Sub
Alibaba Cloud	HTTP Trigger & API Gateway	Object Storage Service	RabbitMQ

Table 2.1: Trigger comparison mapping for HTTP, object storage and queue triggers from the four largest providers by market share (as of 2020) [3].

2.4 Infrastructure-as-Code (IaC)

Infrastructure-as-Code (IaC) is the principle of managing cloud resources through code and configuration files instead of manually through various tools and provider consoles. This enables developers to version their code, test early versions of the infrastructure and increase development speed. Furthermore, an IaC is beneficial to the experiments performed in this thesis where similar workflows have to be deployed to multiple cloud providers.

An example of IaC in practice is Pulumi. Pulumi is an open-source tool that provides IaC using standard languages such as Python, JavaScript, Go, and .NET Core. The purpose of Pulumi is to create, deploy, and manage cloud infrastructures such as containers, serverless functions, networks, databases, or VMs for multiple cloud providers as illustrated in Figure 2.2. While Pulumi does not provide any additional cloud services other than those included by each provider, there are multiple benefits to using Pulumi. These benefits include developers being able to use the language, tools, and libraries they prefer while reducing boilerplate code and avoiding provider-specific YAML and DSL configurations. Another benefit of Pulumi is using the same workflow for all providers. Together with their concept of Policy as Code, organizations and teams can enforce security policies and standards without sacrificing the developers' full autonomy over the resources [13]. In the case of this thesis, Pulumi is used to simplify the development and deployment of cloud resources to Amazon Web Services and Microsoft Azure.

Figure 2.3 illustrates the structure and key components of a Pulumi project. A project is a folder containing all the source code together with some Pulumi configuration files. A program is the source code written in Python, JavaScript, Go, or .NET Core. A program is made up of the IaC code as well as the function code. The IaC code deploys and manages cloud infrastructure and the function code is the actual code that is deployed and run in the cloud. The cloud resources are simply

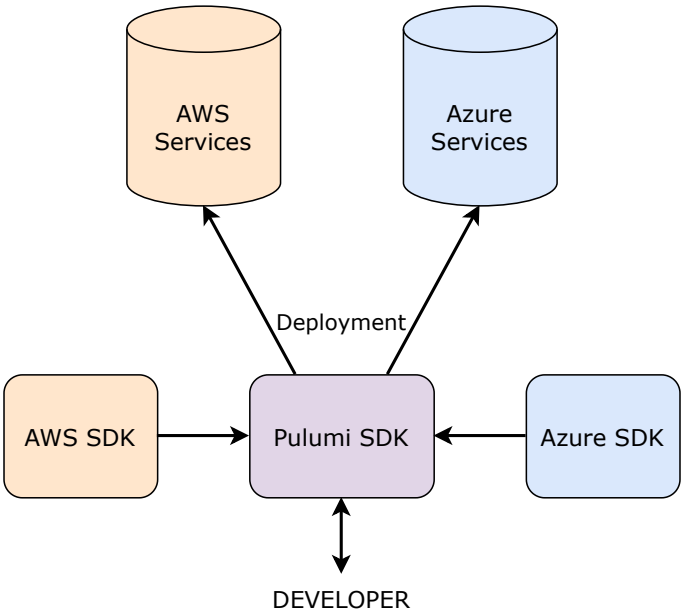


Figure 2.2: How Pulumi fits into the larger architecture and how it acts as an intermediary between the developer and the cloud services.

called resources in Pulumi terms and every resource have input and output values to handle dependencies between resources. For example, one resource could be a REST endpoint with an output of its URL. Another resource, such as an object storage trigger, could use the URL as an input and perform an HTTP request to that URL when triggered. Every Pulumi program runs as a separate and isolated instance called a stack. A stack could be used as a deployment environment (dev, qa, prod, etc) where every stack can have different configurations such as geographical region, secrets, and IDs depending on the cloud provider used. A stack can also have output and input variables to share resources with other stacks and projects.

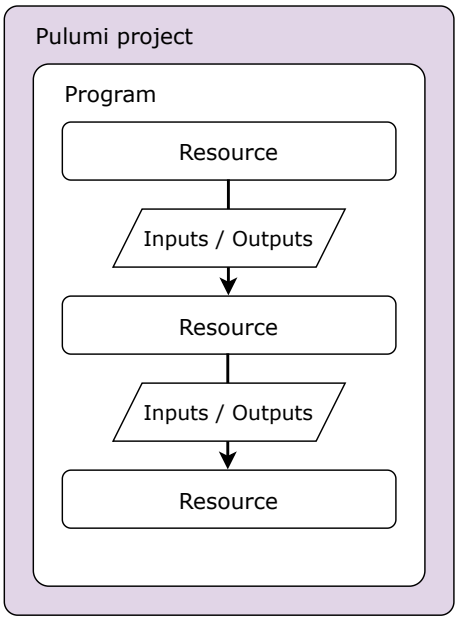


Figure 2.3: The general structure of Pulumi projects. Based on a figure provided by Pulumi [2].

2.5 Distributed Tracing

Historically, tracing application performance was done in monolithic architectures where the system was made out of tightly coupled sub-parts. This made tracing fairly easy as these types of applications tend to run on a specific device in a specific location. Tracing a request and its path through the system could be as simple as analyzing the stack trace in case of an error. With the introduction of cloud computing and serverless, applications have moved toward a microservice architecture where the system is divided into small, independent services that can be developed and deployed in isolation [14]. Despite the benefits of this approach it made tracing more difficult, as tracing the path of a request had to be made across services residing in different locations, running on different hardware and software.

Distributed tracing is the principle of tracing application performance in microservice or cloud architectures. As a request makes its way through the distributed system, it generates traces that can be correlated to other traces through a shared ID, or a parent-child relationship. A trace consists of one or multiple segments which in turn contains information such as service name, operation name, timestamps, and other metadata. As the traces are collected and correlated they can finally be analyzed together. This enables end-to-end visibility of the system in terms of its dependencies, performance, and errors, and thus facilitates debugging. Figure 2.4 shows the timeline of a sample system designed in AWS with distributed tracing enabled through AWS X-Ray. A FaaS function (AWS Lambda) is triggered by an HTTP request from the client to a REST API (AWS API gateway).

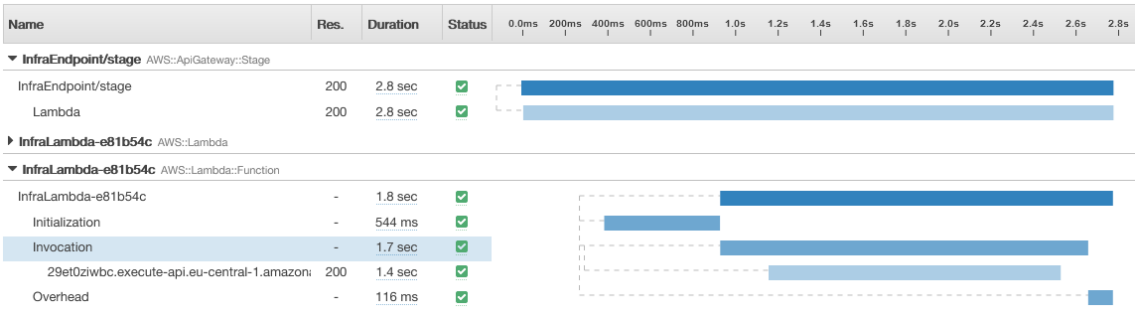


Figure 2.4: A distributed tracing example in AWS using AWS Lambda and AWS API Gateway.

3

Related Work

This chapter presents previous research in Function-as-a-Service performance evaluations related to our goals and research questions. Key differences compared to our method are highlighted and a framework for reproducibility is introduced. The chapter concludes with a discussion about how users perceive delays in software applications.

3.1 FaaS Performance Evaluation

As the adoption rate of FaaS continues to grow, it is important to establish a good understanding of its performance to better understand its use cases and limitations. Previous work by Pelle et al. [15] focuses on latency-sensitive applications where the authors use a drone control application to evaluate the performance of FaaS. The purpose of the study was to gain a better understanding of the performance of FaaS for latency-sensitive applications to enable such applications to better benefit from recent cloud technologies. The authors define a benchmark methodology to study execution time, invocation time, data store access, and data store throughput with varying hardware configurations across multiple services from AWS. The authors found that the selection of which services to use is an important step in designing cloud infrastructure. In the case of AWS, there are several similar services but with drastically different performance characteristics. They further noted that the platform itself introduces a latency affecting execution and invocation time. More specifically, the overall delay in one of their experiments comprised of 47% execution of drone control code, 22% network delay, and 31% platform delay. These findings are not unexpected due to the ephemeral nature of FaaS. In one of their tests, Pelle et al. measure invocation time and how it depends on the payload size. They define invocation time as the time between the service call from the invoker function and the start of the receiver function. This measure of invocation time is similar to how we define trigger delay in Section 5.5. In this thesis, however, we intend to contrast the study by Pelle et al. by focusing on how the type of trigger affects trigger performance rather than payload size. We also extend their scope to include multiple providers, while focusing on a smaller set of services. In comparison to the study by Pelle et al., our study is entirely based on traces from AWS X-Ray and Azure Application Insights. Pelle et al. do utilize distributed tracing but also rely on logging timestamps in every function as well as CloudWatch logs. A benefit to our

tracing-based approach is the possibility to handle concurrent function invocations with out-of-order execution. We provide a detailed overview of our tracing-based approach in Chapter 5.

3.2 Reproducible Experimentation

The software industry in general, and cloud computing in particular, move at a fast pace with its increasing adoption. Studies on cloud services can quickly become obsolete and remain relevant only for short periods due to major changes in services and platforms. To enable future research to compare against previous work, reproducibility must be taken into account when designing, running, and reporting experiments of cloud services.

A framework that can be used to evaluate the reproducibility of experiments within cloud computing was developed by Papadopoulos et al. [8] and consists of eight different principles. When the framework is used to evaluate an experiment, it should give a conceptualization of its overall reproducibility. The eight reproducibility principles, as described by Papadopoulos et al., are:

P1. Repeated experiments: The experiment should be repeated an appropriate number of times with the same configuration and the results should be quantified.

P2. Workload and configuration coverage: The experiment should cover a space of different possible configurations, possibly through randomization.

P3. Experimental setup description: The setup of hardware and software should be described at an appropriate level of detail.

P4. Open access artifact: A (at least) representative subset of the developed software should be made publicly available.

P5. Probabilistic results description of the measured performance: Report a characterization of the empirical distribution of the measured performance.

P6. Statistical evaluation: Provide a statistical evaluation of the significance of the obtained results.

P7. Measurement units: For all reported quantities, report the corresponding unit of measurement.

P8. Cost: The modeled cost of running the experiments should be included.

In the literature review on Function-as-a-Service performance evaluations by Scheuner and Leitner [7], the authors found a severe lack of reproducibility when applying the framework by Papadopoulos et al., to both academic and grey literature. The authors found that only one of the eight reproducibility principles (P7. Measurement units) was fully incorporated by a majority of the papers analyzed. In total, four out of the eight reproducibility principles were fully or partially incorporated

by a majority of the papers in both academic and grey literature.

The methodological principles developed by Papadopoulos et al. are, according to the authors themselves, the first attempt at establishing a minimal set of requirements on the method on which cloud computing experiments are performed. An already established system was chosen over the option of developing a new set of principles, as the latter would likely worsen the conformity within cloud service research.

3.3 User Studies on Response Times

As the purpose of this thesis is to benchmark customer-facing cloud services from AWS and Azure, how users and customers perceive delays is very relevant to the evaluations of trigger latency. There have been several user studies in the past studying response times and latency perception. Google has defined a performance model they call RAIL from its four parts; Response, Animation, Idle and Load [4], based on industry-recognized research by Nielsen [16]. Latency affects the user experience in several ways and every user perceives latency differently depending on what device is used as well as network conditions. A user on an old mobile phone with 3G network access is generally more latency tolerant compared to a user on a high-end desktop computer with a fast fiber internet connection. Google's RAIL model defines 5 latency ranges based on how users perceive latency, as seen in Table 3.1. The RAIL model can serve as a starting point for performance evaluations in cloud applications.

Range	**User perception**
0 to 16 ms	Optimal for animations (60 fps or 16 ms per frame)
0 to 100 ms	Users perceive this range as immediate
100 to 1000 ms	Users perceive this range as natural and acceptable in web applications or when changing view
1000 ms or more	Users lose focus on the current task
10000 ms or more	Users feel frustrated and are likely to not continue using the application

Table 3.1: User perception of latency in real applications [4].

In another study, Jupiter Research [17] studied users' reactions to latency in online shopping applications. They found that 33 % of customers who were dissatisfied with the experience stated that the cause for the dissatisfaction was the site being too slow. They also found that more than a third of these customers abandoned the site mainly for these reasons. For an online shopping brand, this can have profound effects on sales and brand reputation. To minimize these effects the authors recommended keeping site rendering time within 4 seconds.

When it comes to cloud computing, thousands of customers today rely on services provided by major cloud providers like AWS and Azure. In turn, those customers of-

ten have their own customers that rely on the services provided. If the end customers are experiencing high latency in the applications and services they use, it has the potential to cause even greater economic loss and customer dissatisfaction through a domino-like effect. Kohavi et al. [18] refer to how experiments by Amazon showed that for every additional 100 ms in latency, sales decreased by 1%. The authors also refer to another experiment by Google which showed that when the display of search results was delayed by 500 ms, revenues reduced by 20%. These studies highlight the effects latency can have on cloud services and suggest that latency should be minimized whenever possible. Based on previous research, we suggest cloud services should strive toward keeping latency under 1000 ms but ideally under 100 ms.

4

Research Method

This thesis involved designing, developing, and performing an experimental study to compare the function trigger delay of three corresponding cloud services from two different service providers. More specifically, an engineering research approach, based on empirical standards [19], was taken to propose a benchmark design that was later evaluated using controlled experiments. Figure 4.1 lists the steps taken in this study. First, the design of the trigger benchmarks was developed and implemented using Pulumi. This resulted in a set of benchmarks for the two providers, with one benchmark for each trigger type. The set of benchmarks together answer the first research question in Chapter 5. The next step was the execution of the set of benchmarks, which resulted in trace data being collected. Details about the experiment execution is discussed in Chapter 6. The data from the collected traces were then processed and analyzed, leading to the results presented in Section 6.3, answering research questions 2 and 3. A discussion about the results is finally made in Chapter 7.

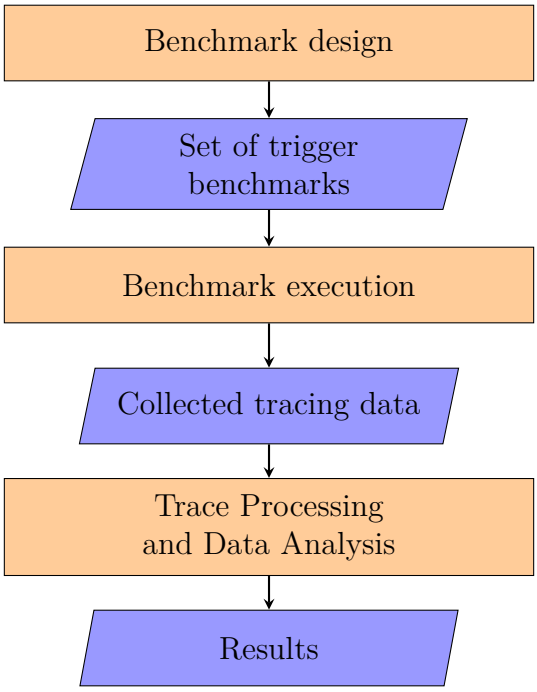


Figure 4.1: A flowchart of the high-level steps in the empirical study.

5

Trigger Benchmark

When designing the structure of the benchmark, two important requirements needed to be satisfied. Firstly, everything should be automated upon running a script, and secondly, the structure should allow for an arbitrary number of experiments to be invoked, i.e. the program should not exit after executing just one event. By automating the benchmark tests, a higher level of reproducibility is ensured as manual steps are eliminated.

Our goal was to design two benchmarks, one for each provider, that are as similar as possible to each other to be able to make fair comparisons. Due to the heterogeneous nature of the providers' services, a perfect comparison is not feasible. Although there are some differences in the two benchmarks, the structure of both benchmarks is conceptually the same, consisting of three fundamental components. Those being, (1) a component to handle shared resources such as configurations, (2) an infrastructure component to serve as an entry point for each benchmark, and (3) a trigger component to handle the actual trigger and workload. All three components are separate Pulumi projects to allow for individual deployment and sharing of resources through Pulumi stack references. These components are discussed in further detail in the sections below. To promote reproducibility, all code is written in the same language, in this case, Node.js JavaScript.

Figure 5.1 shows a high-level model of the architecture of the benchmark. Invoking a benchmark starts with issuing an HTTP request to an API Gateway, either manually or automatically. The API Gateway is connected to a FaaS function that triggers upon receiving the HTTP request. This function acts as the invoker function of the benchmark and is part of the infrastructure component discussed in Section 5.2. The API Gateway of the invoker function was omitted from the figure due to the focus not being on what happens before the invoker function executes. Depending on which benchmark to run, the invoker function will either issue an HTTP GET request to another API Gateway, upload a file to an object storage or send a queue message to a queue. These events will in turn trigger a second FaaS function acting as the receiver function, which runs the workload. The receiver function and associated external services are discussed in Sections 5.3.1, 5.3.2 and 5.3.3.

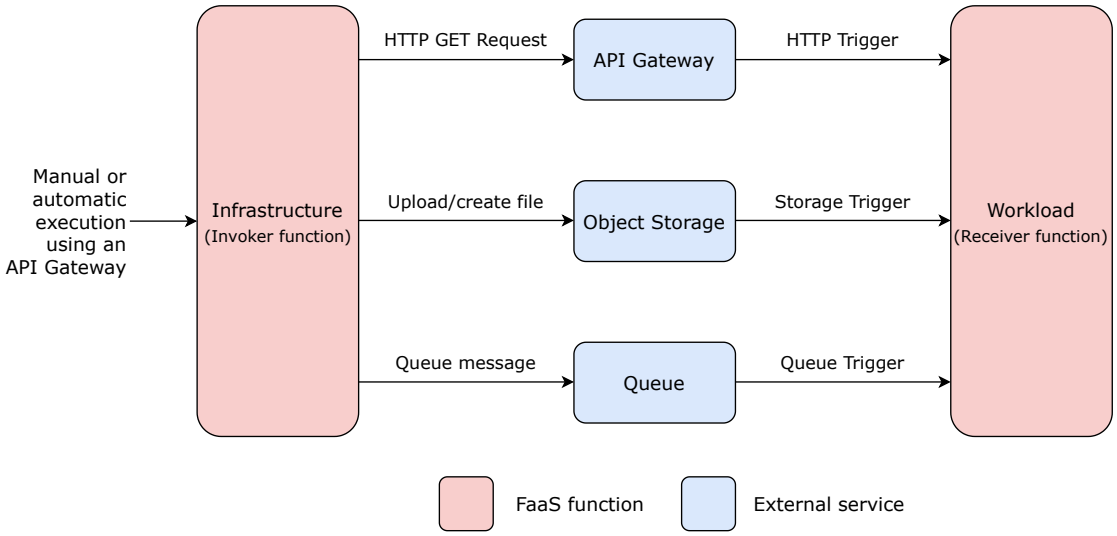


Figure 5.1: High-level benchmark architecture/structure.

5.1 Deployment

Before deploying any resources from the infrastructure or trigger components to any of the two providers, resources shared between the different components are created and deployed. These resources need and should only be created once as they serve as a common configuration between the different triggers within a provider's domain. Resources created within this component also provide authorization to certain users, applications, and services to execute specific tasks in the infrastructure and trigger components. Such tasks requiring explicit permissions include, for example, the creation of new files in one of the object storage services.

The shared resources for AWS include IAM roles and policies to manage resource access permissions. The IAM role was given full access to all services to simplify development. In the case of Microsoft Azure, the shared resources include a Resource Group to manage resources and an Application Insights configuration to be used for distributed tracing.

Figure 5.2 shows the order in which each of the three components previously listed (shared resources, infrastructure, and trigger) must be deployed. The trigger component is deployed before the infrastructure component is, despite the execution being in the opposite order. This is because the output of the infrastructure deployment requires references to the trigger resources to be added as parameters at the end of the infrastructure component's endpoint URL. What trigger resource references are needed depends on the type of trigger that is being deployed.

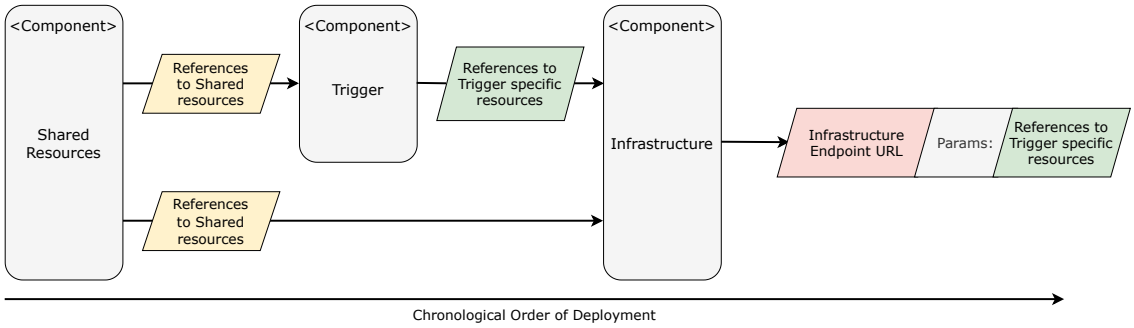


Figure 5.2: The deployment process of each benchmark.

5.2 Infrastructure

The infrastructure component acts as the entry point of each benchmark. It allows triggering of the benchmark flow for a specific provider and trigger type based on user-defined, trigger-specific URL parameters. The infrastructure component is divided into two parts, one for each provider as in the case of the shared component. It deploys a REST API through an HTTP trigger with a handler function that gets triggered by incoming HTTP requests. This enables us to either manually or automatically trigger a benchmark with no further configuration. The handler function is responsible for triggering the workload functions by either issuing an HTTP request in the case of the HTTP trigger, uploading a file to an object storage in the case of the Storage trigger, or uploading a message to a queue in the case of the Queue trigger.

5.3 Trigger Types

In this section, the different trigger types used in this thesis will be discussed. We present how they are implemented and discuss how they relate to the benchmark as a whole. The mapping of corresponding services from each provider is based on a list provided by Microsoft [20], except for the Queue trigger where two alternatives are possible. In this case, the mapping is based on the similarity between services.

5.3.1 HTTP Trigger

The HTTP trigger is the trigger that requires the least amount of setup and configuration out of the triggers being benchmarked in this thesis, and the AWS and Microsoft Azure implementations are structurally very similar. For AWS, an AWS API Gateway was used, with an attached Lambda function as callback. For Azure, an Azure Functions HTTP trigger was used, along with an attached Azure Function as callback. In both cases, the infrastructure component performs an HTTP request to the specified HTTP trigger endpoint which triggers the exposed Azure or Lambda function. As no extra payload is needed, the infrastructure component issues simple HTTP GET requests together with some special headers for tracing, as will be described further in Sections 5.5.1 and 5.5.2. The URL to each HTTP

trigger is created when deploying its Pulumi project. The URLs are then imported to the infrastructure component using Pulumi stack references.

5.3.2 Storage Trigger

To implement a storage trigger in AWS, Amazon Simple Storage Service (S3) was used. S3 is AWS's implementation of an object storage and can be used to store and retrieve large amounts of data. As all data is part of an S3 bucket, the deployment process of the Storage trigger begins with deploying a S3 bucket with an attached Lambda function that triggers on the *onObjectCreated* callback. For the storage and trigger functionality itself, no other resources are needed. To trigger the AWS version of the Storage trigger, the infrastructure component simply uploads a new file to the specified S3 bucket. This will trigger the *onObjectCreated* callback which in turn will trigger the attached Lambda function to execute its workload.

Azure Blob Storage is Azure's implementation of an object storage service. Similar to S3 buckets, data in Azure Blob Storage is uploaded to Azure Storage Containers to manage and group data, similar to a folder in a traditional file system. The Storage Container is connected to a Storage Account to handle data access permissions. In the Azure version of the Storage trigger the traditional Azure Blob Storage trigger was not used and instead used an Event Grid trigger. The reason for this is that the Event Grid trigger offers better performance in terms of blob updates per second as well as avoiding the up to 10-minute processing delay present in Blob Storage triggers. The time difference between Blob Storage trigger and Event Grid trigger is so great because Event Grid is event-driven, meaning that an event triggers an action to be taken, while Blob Storage trigger polls the storage every few milliseconds to every few minutes depending on recent activity. This means that the Event Grid trigger is more similar to how an AWS S3 bucket trigger functions, in that both utilize events, making a comparison between these two services fairer. The Event Grid trigger is set to trigger a callback in the form of an Azure function on the *onGridBlobCreated* event. To trigger the Azure version of the Storage trigger, the infrastructure component uploads a new file to the specified Storage Container. This will trigger the *onGridBlobCreated* callback and the attached Azure function to execute its workload.

5.3.3 Queue Trigger

To implement a queue trigger in AWS, Amazon Simple Queue Service (SQS) was used. The deployment process begins with deploying an SQS queue with an *onEvent* callback in the form of a Lambda function. Similar to the HTTP and Storage triggers, the Queue trigger is triggered from the infrastructure component by sending a new message to the specified queue. The message then triggers the *onEvent* callback and the attached Lambda function.

In the Azure implementation, Azure Queue Storage was used to deploy a Storage Queue connected to a Storage Account that manages message access permissions. In addition to Azure Queue Storage, Azure also provides Azure Service Bus Queue

as an alternative. The reason why Azure Queue Storage was favored and ended up being used over the alternative is that the former is more similar to AWS SQS (with default configuration) than the latter is. Both AWS SQS (with default configuration) and Azure Queue Storage use best-effort ordering with at-least-once delivery. To the deployed Storage Queue, an Azure Function was attached as a callback to *onEvent*. As in the AWS SQS Queue trigger, the infrastructure triggers the Azure Queue trigger by sending a new message to the specified queue. This will trigger the *onEvent* callback and the attached Azure Function.

5.4 Workload

When performing micro-benchmarks, which has been a common research topic in the past, the workload plays an important role as it is the performance of this component of the system that is measured. In this thesis, however, the performance of the workload is ignored and its only purpose is to serve as an endpoint for the measured trigger. In other words, the measurement stops as soon as the workload starts executing. The workloads are run as simple serverless functions and use AWS Lambda or Azure Functions for their respective cases. More specifically, the workloads are run in the receiver component.

The purpose of this thesis was not to measure the execution performance of cloud service providers and, therefore, a minimal workload in terms of code length was desired to have as little effect on start-up times as possible. In his experiments, Shilkov [21] found a correlation between greater package size, i.e. a larger application with more dependencies, and longer cold-start times for AWS. A minimal workload would, therefore, theoretically interfere less with the overall delay as well as potentially minimizing the cost of resource allocation and execution time for providers with pay-as-you-go financial models, such as AWS and Microsoft Azure. The source code of the workload used in this thesis by all trigger types can be found in Figure 5.3.

```
1 const factorial = (n) => {
2   let res = 1;
3
4   for (let i = 2; i <= n; i += 1) {
5     res *= i;
6   }
7   return res;
8 };
```

Figure 5.3: The minimal workload that all trigger types execute.

5.5 Distributed Tracing

To achieve the goal of measuring trigger latency, we want to obtain end-to-end visibility of the system by using distributed tracing to instrument every component. More specifically, we want to be able to track a request as it makes its way from the invoker function, to the external services, and to the receiver function as depicted in Figure 5.1. Both AWS and Microsoft Azure offer relatively advanced and configurable tracing services that work for both monolithic and distributed systems. As mentioned earlier, tracing a distributed system across several micro-services is inherently more difficult than tracing a non-distributed system as the services themselves most often are not related to each other (except in geographical location). Using the benchmark designed and discussed in this chapter, all traces will have the same high-level structure and contain the same components.

The benchmarks have two types of invocations, synchronous and asynchronous invocations. In synchronous invocations, as illustrated in Figure 5.4, the benchmarks start by executing the invoker function and a timestamp *T1* is recorded. The invoker function then calls a synchronous dependency by issuing an HTTP request. Once the dependency responds to the call made by the invoker function, the request is made at timestamp *T3*. The request then triggers the execution of the receiver function at timestamp *T4*. In addition to logging timestamps, provider-specific trace identifiers are also passed from the invoker function to the request and finally to the receiver function to correlate the traces. In asynchronous invocations, as illustrated in Figure 5.5, the benchmarks also start by executing the invoker function at timestamp *T1*. The invoker function then calls an asynchronous dependency using an HTTP request. The request is eventually made at timestamp *T3* and finally, the request triggers the execution of the receiver function at timestamp *T4*. Like with synchronous invocations, provider-specific trace identifiers are passed on from the invoker function to the request, to the receiver function to enable trace correlation. The time between *T3* and *T4* is, for both synchronous and asynchronous invocations, the time the system waits for the next function execution to happen, thus making this delay the trigger latency measured in this thesis.

This is, however, not the only way to measure the trigger latency. In AWS X-Ray the resulting trace is highly detailed and contains several different timestamps other than those used in this thesis. Unfortunately, Azure Application Insights is not as detailed and the method described above is the only one that results in timestamps that could be extracted for both AWS and Microsoft Azure.

The benchmark methodology of this thesis relied on distributed tracing services provided by AWS and Azure. During the process of this thesis, several limitations in those services regarding distributed tracing were discovered. In the sections below we discuss how distributed tracing was used for AWS and Azure as well as how limitations in the AWS X-Ray and Azure Application Insights implementations were handled.

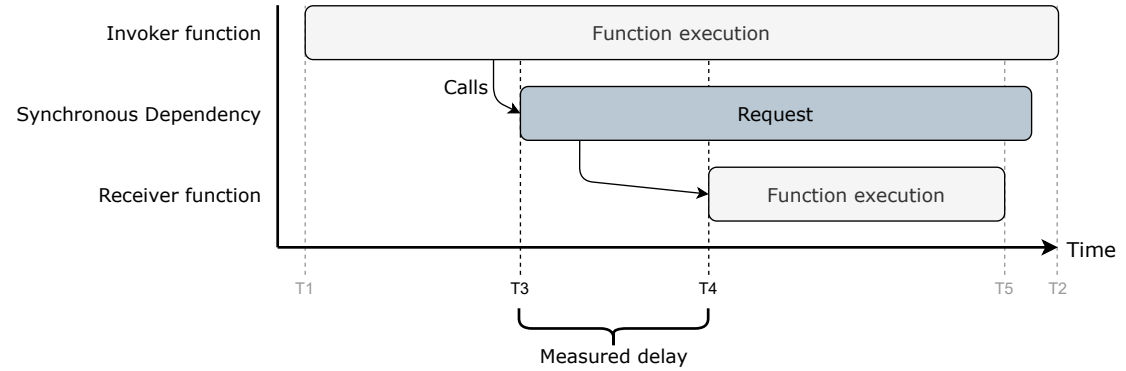


Figure 5.4: A representation of a trace for a synchronous trigger, i.e. the HTTP trigger.

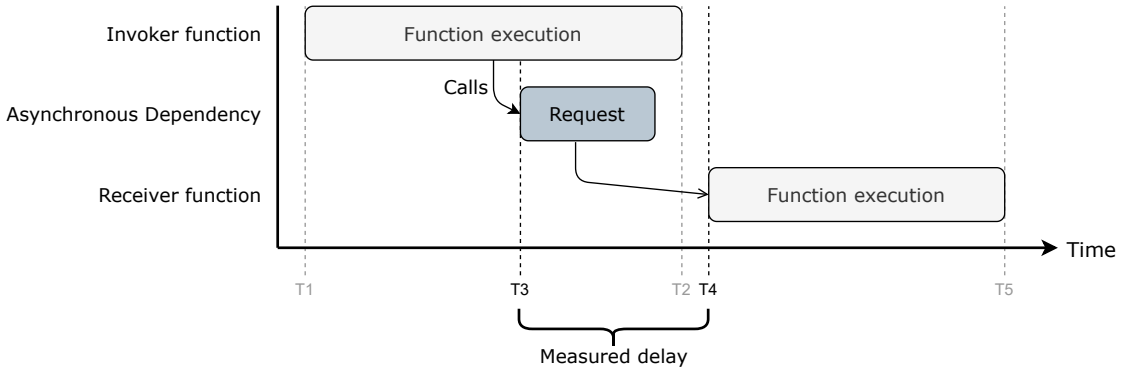


Figure 5.5: A representation of a trace for an asynchronous trigger, i.e. the Storage and Queue triggers.

5.5.1 Amazon Web Services X-Ray

Distributed tracing for AWS relies on AWS X-Ray and is enabled per resource to obtain end-to-end visibility of the system. Exactly how to enable X-Ray tracing of a certain resource depends on its type. For Lambda functions, a *TracingConfig* set to active is enough to enable tracing of that Lambda function. For API Gateways, X-Ray is enabled through *stageArgs* by setting the X-Ray tracing flag to true. This results in X-Ray logging incoming requests to the specific component. By repeating the processes for other components, the system as a whole can be instrumented. After invoking a number of benchmarks, the X-Ray traces are downloaded using the AWS SDK and processed by a Python script.

Although X-Ray provides a good overview of a system, there are some limitations to the services provided by AWS. It was found that in the case of the Queue trigger benchmark, where a Lambda function (invoker) uploads a message to a queue (trigger) that triggers another Lambda function (receiver), trace correlation was not achieved. Invoking the Queue trigger benchmarks lead to disconnected traces where X-Ray did not correlate the invoker function with the receiver function. This was later found to be due to X-Ray not supporting instrumentation of AWS SQS out-of-the-box. As the two traces had no shared trace-ID and no clear relationship

between each other, automating the collection of trace data would be impossible due to out-of-order executions. At the point of this thesis, there was no official solution to this problem. As trace correlation is a crucial part of the experiment, a manual workaround had to be implemented. AWS SQS messages support *MessageSystem-Attributes* with a *AWSTraceHeader* as meta-data. Our solution to the disconnected traces was then to send the correct X-Ray trace-ID as an *AWSTraceHeader* and read it in the receiver function. After reading it, an annotation is added to the current X-Ray trace segment with the value of the *AWSTraceHeader*. This enables manual correlation of the disconnected traces through a Python script.

5.5.2 Microsoft Azure Application Insights

In the Azure implementation of the benchmarks, the Azure Application Insights service is used to trace the system. An Application Insights configuration is created as a shared component and is subsequently used by all other components to group traces and create a single, correlated trace. Using the Azure Application Insights REST API [22], traces related to a specific Azure Application can be fetched, which can then be filtered and analyzed using a Python script.

While the AWS X-Ray traces are relatively detailed showing timestamps for both services and their sub-components, Azure Application Insights records much fewer timestamps but includes function requests, function execution, and dependency calls. The limited extent of the Azure traces narrows the configuration space, making Azure Application Insights the deciding factor in what timestamps to measure trigger latency between. Looking at Figure 5.4 and 5.5, *T1* to *T5* are the only timestamps available in Application Insights while X-Ray records additional timestamps. Because of this, the trigger latency had to be defined as the time between two of those five timestamps.

Similar to the case of AWS X-Ray, it was found that Azure Application Insights have some limitations in tracing some Azure services. In the Queue and Storage trigger benchmarks, Azure Application Insights was unable to forward tracing headers between components which lead to disconnected traces. In the Queue trigger benchmark, there is an Azure Function (invoker) that uploads a message to a queue (trigger) and triggers a second Azure Function (receiver). Similarly, in the Storage trigger benchmark, there is an Azure Function (invoker) that uploads a blob to a container (trigger) which triggers a second Azure Function (receiver). Running the benchmarks of those triggers leads to disconnected traces with no reliable way of connecting them. At the point of this thesis, there was no official solution to these problems. But considering that everything was fully implemented and the only issue being the correlation itself, a simple workaround could be developed. Since the Storage trigger relies on a Blob being uploaded to a Storage Container, meta-data can be passed together with the uploaded blob. In the event that triggers the receiver function, the included meta-data can be read directly by the receiver function. By sending a message with the correct operation-ID as meta-data, a custom trace can be logged in the receiver function where the trace message consists of the correct and current (but incorrect) operation-ID. This enables manual correlation between

the invoker and receiver functions through a Python script. As the same problem was present in the Queue trigger benchmark, a similar solution was implemented. However, since queue messages in Azure Storage Queue do not support sending meta-data, the correct operation-IDs were instead included in the actual message text. As the operation-IDs are short, this should not affect the package size, and thus, should not affect the overall performance. By reading the correct operation-ID in the receiver function and manually logging it as a custom trace together with the current (but incorrect) operation-ID, the invoker and receiver functions can be manually correlated in a Python script.

6

Trigger Experiment

This chapter presents the details of the controlled experiment used to evaluate the benchmark design proposed in Chapter 5. The chapter begins with discussing the methodology, namely, the execution and trace processing, and concludes with a section on the main findings with respect to our research questions.

6.1 Execution

How a cloud service is managed and run is a highly complex system, meaning many variables determine its performance. The performance can, therefore, vary significantly depending on the server location chosen, the day of the week, and the time of day the tests are performed. In this thesis, the goal was to compensate for two of these independent variables, specifically the day of the week and the time of day, by running a set of invocations throughout the day over multiple days. The server locations chosen were not changed throughout the benchmark execution, however, choosing instead to use servers in similar geographical locations for both AWS and Azure. More specifically, this location was chosen to be central Europe, where both providers happen to have a server hall in Frankfurt, Germany (the identifiers used are *eu-central-1* for AWS and *GermanyWestCentral* for Microsoft Azure.) Each set consisted of 1000 sequential invocations run three times per day (9:00, 15:00, and 21:00 CEST) between the 14th and 17th of May 2021, resulting in a total of 12000 invocations for each of the six benchmarks.

The sheer number of tests that had to be run at roughly the same time required automating the invocations using the *serverless-benchmarker* tool envisioned by E. van Eyk et al. [9]. Using hooks, *serverless-benchmarker* could be used to efficiently deploy, invoke, and destroy the resources specified in the benchmarks. For the AWS benchmarks, *serverless-benchmarker* was also used to extract the traces recorded by X-Ray through the AWS SDK. In the case of the Azure benchmarks, its REST API had to be used manually as *serverless-benchmarker* provided no support for extracting Azure Application Insights traces [22]. Under the hood, *serverless-benchmarker* provides automated load generation through the open-source loading testing tool *k6* [23]. This lets us sequentially invoke the large number of benchmarks required by our experiment.

6.2 Trace Processing

When traces from AWS and Azure are fetched, the next step is to process the traces. The processing of trace data is handled by Python scripts where the trace data is represented as JSON objects. The processing of trace data starts with grouping traces by *Trace-ID* for AWS or *Operation-ID* for Azure. A group, in this case, represents all trace data for one invocation of a certain benchmark. After grouping the traces, the trigger latency is calculated based on the available timestamps as discussed in Section 5.5. All delays are recorded in milliseconds and exported as CSV files which in turn can be analyzed and visualized by a separate Python script.

6.3 Experiment Results

This section presents the results from the six benchmarks performed in this thesis. Figures 6.1 and 6.3 show the results as frequency distribution plots for all results from the AWS and Azure benchmarks respectively. Figures 6.2 and 6.4 show the same results as cumulative distribution functions (CDF), also for AWS and Azure respectively. The CDF of a random variable X, in this case the delay, evaluated at x, is the probability that X will take a value less than or equal to x. As an example, looking at Figure 6.2, the probability that the AWS Storage trigger has a delay less than or equal to 1500 ms is about 0.6 or 60%. Table 6.1 shows a summary of general statistics for each of the six benchmarks, including sample size, mean, median, standard deviation, relative standard deviation, and the 75th and 99th percentiles of each distribution. Relative standard deviation is the standard deviation expressed as a percentage of the mean.

	AWS HTTP	**AWS Storage**	**AWS Queue**	**Azure HTTP**	**Azure Storage**	**Azure Queue**
Sample size	11972 (-28)	11715 (-285)	11826 (-174)	11852 (-148)	11932 (-68)	11168 (-832)
Mean	40.8 ms	1309.4 ms	121.0 ms	58.2 ms	303.1 ms	749.6 ms
Median	33 ms	1298 ms	93 ms	30 ms	298 ms	125 ms
Standard deviation	48.7 ms	346.8 ms	158.7 ms	270.2 ms	150.9 ms	2002.8 ms
Relative standard deviation	119.4%	26.5%	131.2%	464.4%	49.8%	267.2%
75th percentile (P_{75})	45 ms	1556 ms	112 ms	38 ms	412 ms	163 ms
99th percentile (P_{99})	162 ms	2114 ms	656 ms	153 ms	645 ms	9838 ms

Table 6.1: Summary of statistics for all benchmarks.

When extracting the traces only complete traces, i.e. invocations that executed both the infrastructure (invoker function) and the trigger workload (receiver function),

were recorded and used in the analysis. This means that for all six benchmarks the sample sizes are smaller than the expected 12000 (1000 invocations × 3 times each day × 4 days). Other than some traces being incomplete, other traces were not recorded (due to sampling) by AWS X-Ray and Azure Application Insights to reduce traffic and cost. These traces were not displayed at all in X-Ray and only partially displayed in Azure Application Insights. Due to the above reasons, 2.1% of the total expected sample size was lost.

Despite the samples being taken at different times, all twelve samples have been combined into one for each of the six benchmarks. This is because the results, overall, showed little variance when performed over different times and days and no correlation between time and day and performance was found. The little variance that did occur was likely caused by random differences in performance. The justification for this will be discussed further in Section 7.4.2.

6.3.1 Amazon Web Services

For all three AWS benchmark results, the median is somewhat lower than its mean, meaning the distribution is positively skewed resulting in a long tail on its right side. This is, however, common when measuring delays. The HTTP benchmark had the lowest mean of 40.8 ms, significantly lower than the mean of the Storage benchmark (1309.4 ms) and less than half the mean of the Queue benchmark (121 ms). Despite having the lowest mean, the HTTP benchmark had the second-highest relative standard deviation of 119.4%. The results from the Storage trigger show a very high mean of 1309.4 ms and standard deviation of 346.8 ms. The median (1298 ms) of this distribution is, however, almost equal to the mean and it takes the shape of an almost perfect normal distribution. The low relative standard deviation (26.5%) of the same distribution shows that the spread is relatively small if the relatively high mean is taken into consideration. The Queue trigger had the highest relative standard deviation of 131.2% but a mean (121 ms) fairly close the the HTTP trigger. The distributions of both the HTTP trigger and Queue trigger have clear peaks, indicating less spread compared to the Storage trigger.

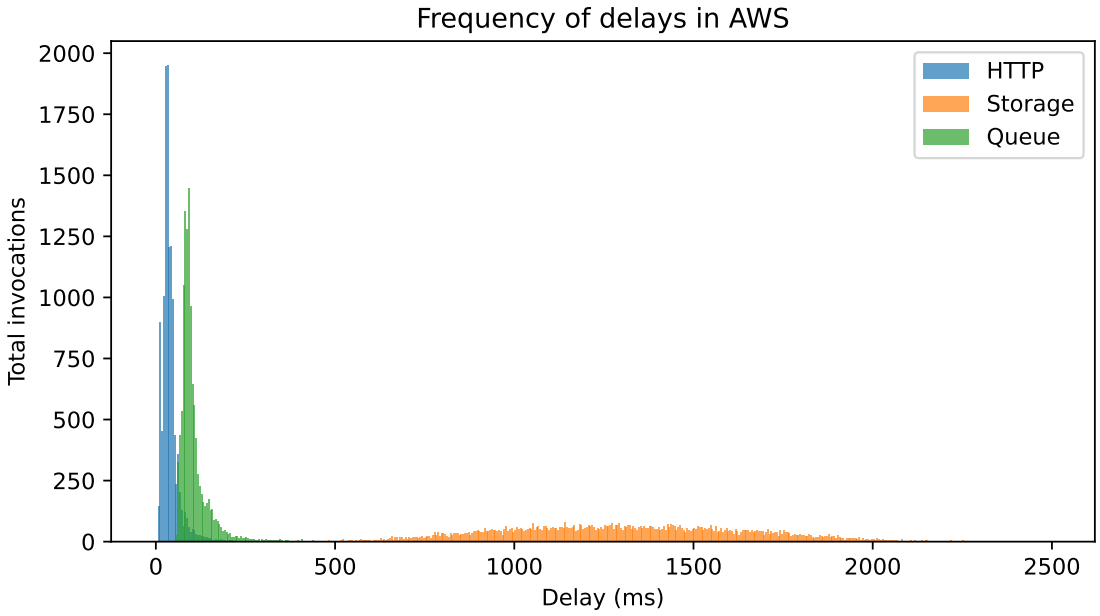


Figure 6.1: Frequency of delays in the AWS trigger benchmarks.

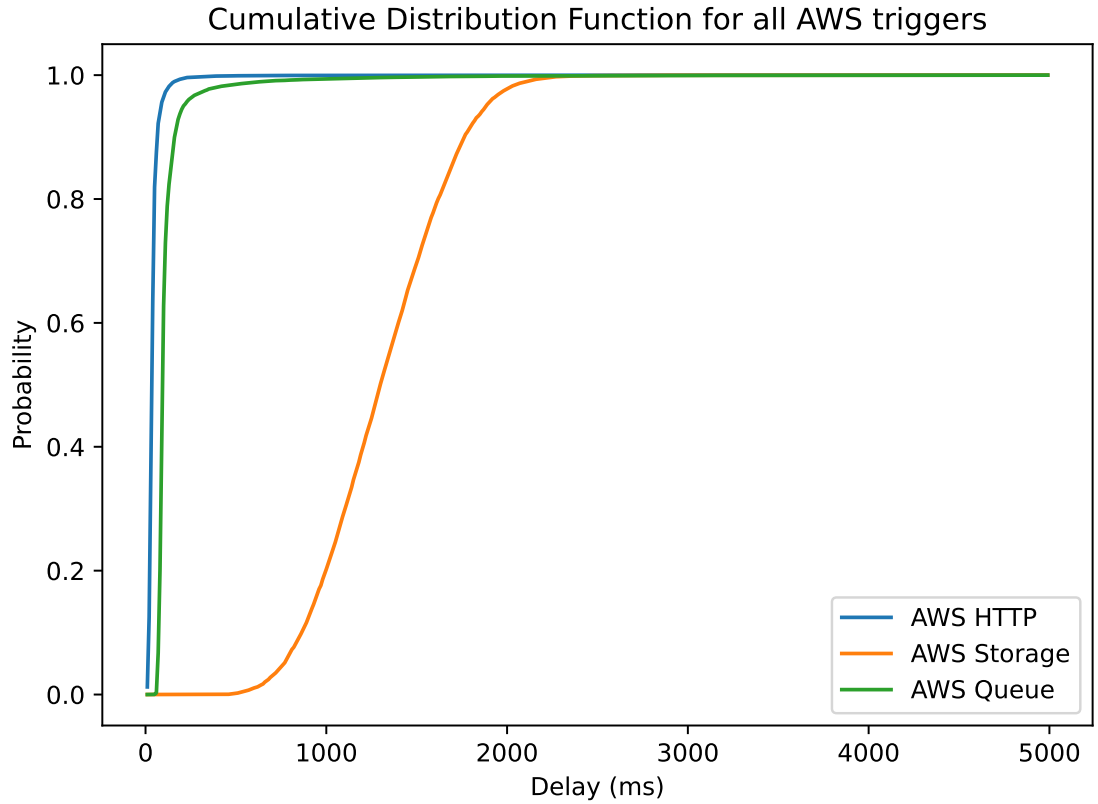


Figure 6.2: Cumulative distribution functions for the AWS trigger benchmarks.

6.3.2 Microsoft Azure

In general, the results from the HTTP benchmark have little spread but its mean (58.2 ms) is almost double its median (30 ms), and both the standard deviation (270.2 ms) and relative standard deviation (464.4%) are relatively high. This was caused by some extreme outliers reaching over 3000 ms in delay. As can be seen in Figure 6.3, the distribution of the HTTP trigger in Azure has a long tail and is tightly clustered around its mean. The presence of extreme outliers is even more noticeable looking at the results from the Queue benchmark. In this case, the mean (749.6 ms) is almost six times the median (125 ms), and the standard deviation (2002.8 ms) is once again high. In the case of the Queue trigger, some extreme outliers reach as much as 12000 ms in delay. The percentile-values in Table 6.1 and the cumulative distribution function of the Queue trigger in Figure 6.4 clearly display this phenomenon. The results from the Storage benchmark has relatively little spread, with a standard deviation of 150.9 ms, but consists of four clear peaks. The mean of the Storage trigger benchmark (303.1 ms) is lower than the Queue trigger benchmark (749.6 ms) but still significantly higher than the mean of the HTTP trigger benchmark (58.2 ms). Interestingly, like in the case of AWS, the Azure Storage trigger benchmark showed a much smaller relative standard deviation (49.8%) compared to the other triggers.

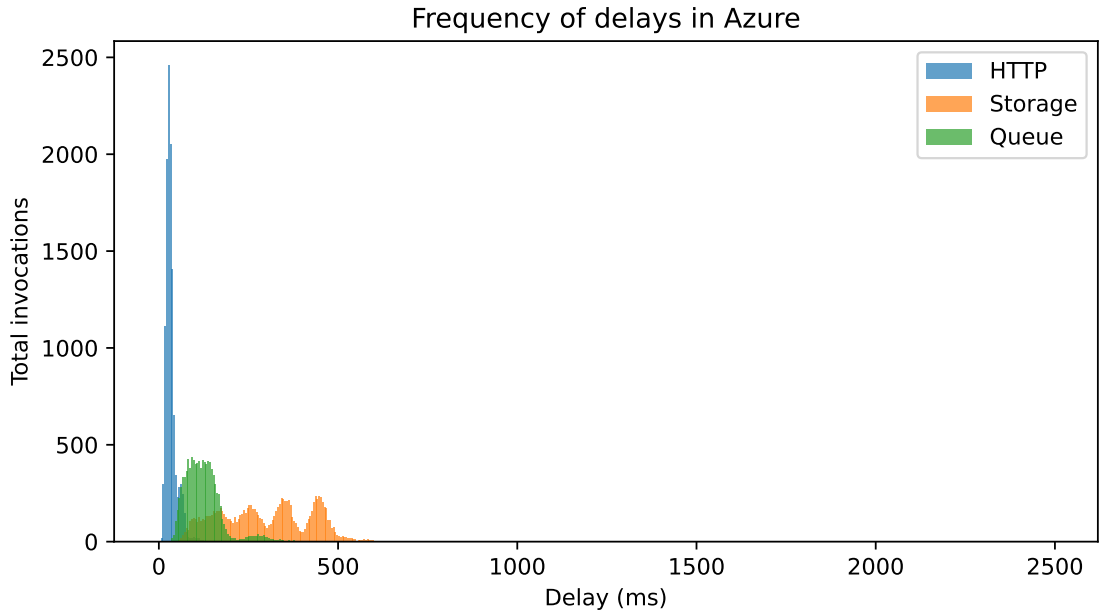


Figure 6.3: Frequency of delays in the Azure trigger benchmarks.

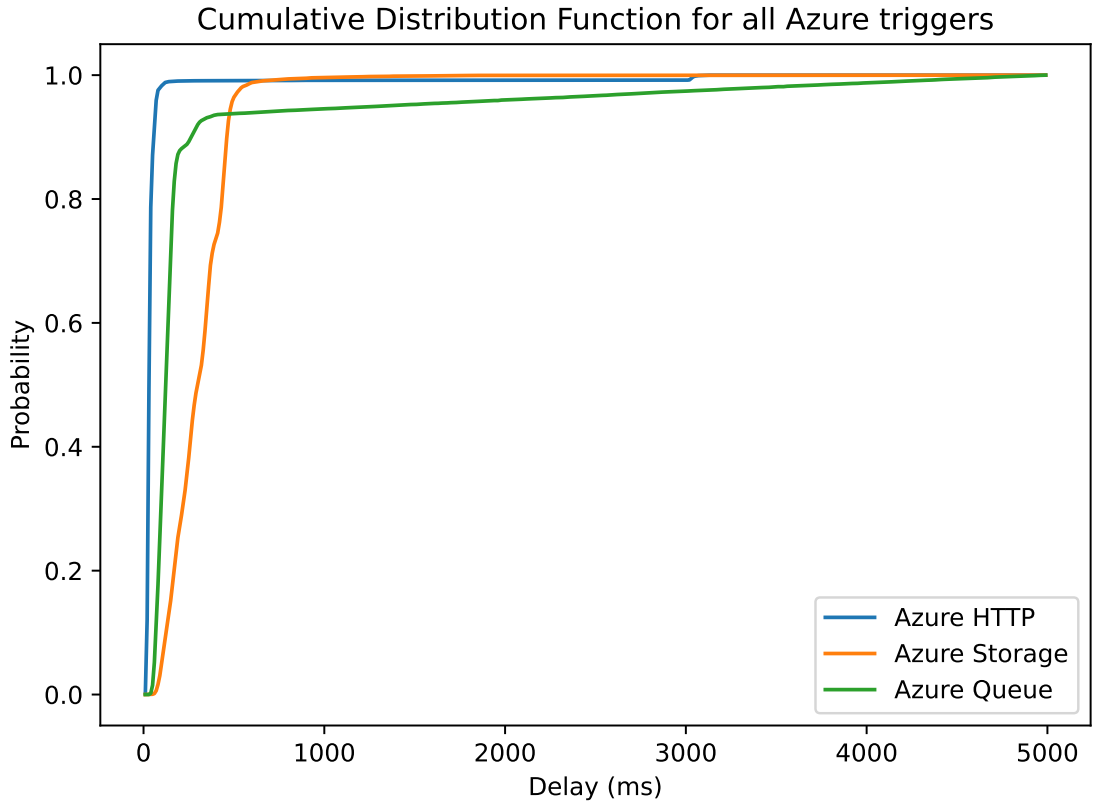


Figure 6.4: Cumulative distribution functions for the Azure trigger benchmarks.

7

Discussion

Based on our research questions, this chapter reflects upon and discusses the results from Chapter 6. Threats to the validity of the results and conclusions are also presented. Finally, the reproducibility of this thesis is discussed in terms of the eight reproducibility principles introduced in Chapter 3.

7.1 RQ1: Measuring latency across providers

As previously discussed, it is difficult to find a one-size-fits-all benchmark solution for services that have been developed separately despite being very similar in purpose. To make the benchmarks as fair and comparable to each other as possible, the differences between the benchmarks for different providers have been kept to a minimum, avoiding any provider-specific configurations or components that might alter the results. The Infrastructure-as-Code (IaC) service Pulumi has helped in this regard by allowing the same workflow to be used for both providers. Despite these precautions and the benefits of Pulumi, some differences that will affect the benchmarks are bound to arise, however.

The success of designing a fair benchmark is highly dependent on the underlying architecture of the triggers and related services. Starting with the HTTP trigger, the benchmark structures of the HTTP trigger for both AWS and Azure are quite similar. In both cases, the FaaS function is exposed through a simple HTTP endpoint making a comparison between the providers fairer. Although the internal architectures of the HTTP endpoints likely vary between providers, we argue that the benchmarks for the HTTP trigger are as similar as it gets when taking the factors we have control over into account.

Continuing with the Storage trigger, both the AWS and Azure implementations of it are event-based, as discussed in Section 5.3.2. The AWS Storage trigger benchmark uses an *onObjectCreated* callback handler on the specified bucket to trigger the lambda function. The Azure implementation of the Storage trigger benchmark uses an *onGridBlobCreated* callback handler of an event grid to trigger the Azure function. On a surface level, the benchmark structures for both implementations of the Storage trigger seem fairly similar. However, the internal architecture behind each trigger is largely unknown. In terms of benchmark design, we argue that our Storage trigger implementations are as similar as they get considering the factors

we have control over. We also expect the providers' internal architectures of the Storage triggers to differ more between providers than the HTTP triggers likely do.

The Queue trigger is perhaps the most interesting trigger out of the three from a technical perspective. Both the AWS and Azure implementations are poll based making the polling-algorithm the most important difference between the two. The benchmarks for the Queue trigger both use the default settings out of the ones that can affect polling. As discussed in Section 5.3.3, both implementations use best-effort ordering with at-least-once delivery. Azure Storage Queue uses a random exponential back-off algorithm for polling to avoid polling idle queues. The maximum wait time is configurable through the *maxPollingInterval* property and in our case, it is set to the default value of 1 second. AWS SQS uses short polling where a subset of SQS servers, based on weighted random distribution, is queried. This might cause a poll request to not return all messages but, according to the AWS documentation, if the number of messages is less than 1000, a subsequent request will return the remaining messages. In terms of benchmark design, we argue that the benchmarks for the Queue triggers are designed to be as fair as possible.

7.2 RQ2: How the trigger type affects latency

As seen in Figure 6.3 and mentioned in Section 6.3.2, the distribution of the HTTP trigger in Azure has a long tail where the measured delays are tightly clustered around the mean. The Azure Storage and Queue triggers display other patterns with the Storage trigger distribution showing a wave-like pattern where the frequency is higher for some delays. The peaks also seem to occur almost exactly every 100 ms which is an interesting result considering that the *EventGrid* resource used for triggering the Storage workload is event-based. This could suggest that the number of events is throttled which could give the resulting pattern. In addition to displaying different patterns, the high relative standard deviations of the HTTP and Queue triggers indicate the presence of extreme outliers as can be seen in Figure 6.4.

Based on the measured results from all benchmarks, HTTP triggers perform significantly better than Storage and Queue triggers. A majority of all benchmark invocations of an HTTP trigger fall within the 0 to 100 ms category of Google's RAIL model [4], where users perceive delays as immediate. According to research by Kohavi et al. [18] delays of this magnitude would also increase revenue by minimizing customer dissatisfaction due to higher delays. The HTTP triggers are also the most consistent in terms of delay, as visualized in Figures 6.2 and 6.4.

Storage triggers were shown to have higher means, 1309.4 ms and 303.1 ms for AWS and Azure respectively. 303.1 ms falls within the 100 to 1000 ms category in Google's RAIL model where users perceive delays as natural in web applications. However, 1309.4 ms is perceived differently by users. In the category of 1000 ms or more in delay, users start to lose focus on the current task. Delays of this magnitude would also severely impact revenue by several percent. Similar to the HTTP trigger, the Azure Storage trigger was shown to be fairly consistent in the measured delay while the AWS version was surprisingly inconsistent. Figure 6.2 and 6.4 show the

cumulative distribution functions of the Storage triggers, where the Azure Storage trigger is fairly close to being as consistent as the HTTP trigger while the AWS Storage trigger is much less consistent compared to the HTTP and Queue triggers.

The Queue triggers also performed worse compared to the HTTP triggers with mean delays of 121.0 ms and 749.6 ms for AWS and Azure respectively. Both of these means fall within the acceptable range of Google's RAIL model but, compared to the HTTP trigger delays, they would decrease sales significantly. As visualized in Figure 6.2, the Queue trigger was shown to be the second most consistent in the measured delay for AWS and the least consistent trigger for Azure.

The results from this study are not only beneficial to the research topic, but also to industries that utilize cloud services. Most importantly it shows that the trigger chosen to be used in a system can greatly affect its performance and, according to these results, an HTTP trigger should be preferred over others when the choice is possible. This research can also help the providers recognize strengths and weaknesses in their services so appropriate actions can be taken to improve their quality.

7.3 RQ3: How the provider affects latency

Looking at Table 6.1, AWS generally performs better in terms of mean and relative standard deviation compared to Azure. An exception is the Storage trigger which performed better in the Azure version with a much smaller mean and standard deviation. The results also show that AWS performance is generally more consistent across invocations compared to Azure. Both the Azure HTTP and Azure Queue triggers had a significant number of extreme outliers, with high relative standard deviations. This is especially noticeable in the difference between the 75th and 99th percentiles of performance in the Azure Queue benchmark as can be seen in Table 6.1. 75% of invocations had a delay of 163 ms or less but 99% had not finished until after 9838 ms. Extreme outliers in delay can negatively affect user satisfaction if the application normally performs better. A large delay is likely to more noticeable if the user is accustomed to faster load times. This also negatively affects latency-sensitive applications.

The results from this study are especially beneficial to developers and software architects having to decide between what providers to use for a system. Aside from performance being important in many applications, cost is another important factor in deciding what provider to use. Although this thesis does not provide any detailed cost analysis, it is possible that the cost differences between providers are significant, especially in large applications. Depending on what features and services that are necessary for the system being developed, what performance requirements are present, and budget, the preferred choice of provider can vary. For HTTP and queue triggers Amazon Web Services is the recommended choice as it is faster than Microsoft Azure on average. For storage triggers, Azure is the recommended option as the average trigger delay is shorter. But for all three triggers Azure is more likely to have performance outliers and can be considered less predictable. Cloud developers and architects should, therefore, consider the trade-off between speed

and predictable performance.

7.4 Threats to Validity

In this section, threats to the validity of the results are discussed. The discussion is guided by the popular categories of construct validity, internal validity, and external validity [24]. Construct validity is the degree to which the experiments performed answer the research questions proposed. Internal validity is the degree to which a causal relationship can be found between the dependent and independent variables. External validity is the degree to which the results of this study are generalizable to other independent variables or the subject as a whole.

7.4.1 Construct Validity

An important point in the construct validity of this thesis is to what degree Amazon Web Services and Microsoft Azure are comparable. To make a comparison on performance, some aspects of the two services must be similar. But as previously discussed, due to their separate development, two services from different providers that have been compared in this thesis are not necessarily similar in structure, making a simple one-to-one comparison difficult. However, one thing the two services compared in this thesis have in common is the purpose they serve to users. Whether a user decides to use a service from Amazon Web Services or Microsoft Azure, they expect it to serve the purpose they have for it. This makes comparisons across providers comparable in terms of purpose, while not necessarily in terms of structure.

The potential for issues in the construct validity of the benchmark has been mitigated with Pulumi and the modular approach used. The modular approach means that both the infrastructure (invoker function) and the workload (receiver function) use the same code across trigger types for the same provider. Since that makes the triggering component the only component that varies between tests, the likelihood of other components affecting the results is lower.

7.4.2 Internal Validity

In terms of internal validity, we want to answer whether or not the results correlate to the type of trigger and provider rather than some other factor. As the results were gathered over multiple days and at different times each day and still showed very little variance, it suggests that time and day likely has little effect on trigger performance in terms of delay. However, comparing the results between trigger types and providers showed clear differences in performance. Multiple other factors could potentially affect the results, including the geographical location of the cloud resources, dynamic resource limits, and multi-tenancy in the underlying VMs. These factors are either difficult to measure or were left as future work to investigate how they affect trigger performance.

When presenting the results in Chapter 6 all twelve samples from each benchmark were combined into one sample, thus accounting for differences in time and day but not being differentiated between. The justification for this was that the different samples visually showed little variation. Two common tests used to measure the variance between samples are Kruskal-Wallis and Kolmogorov-Smirnov tests. The Kruskal-Wallis test can be used to prove whether two data sets have been sampled from the same population and the Kolmogorov-Smirnov test is used to calculate the goodness of fit between two or more samples. However, due to high test scores neither test could prove that two samples from the same benchmark taken at different times and/or days were sampled from the same population, with the exception of the Azure Storage trigger over different times. This is likely due to minute differences in performance that are not relevant to this thesis in combination with large sample sizes, which tend to increase the test scores. The justification for combining the twelve samples is, therefore, not based on statistical evidence but rather based on the visual similarity between samples. Had the difference between samples been greater than what was measured this might have been problematic, but, since the aim of the study was not to compare differences over time and day, just to measure over it to take possible differences into account.

Due to the closed nature of both providers' internal tracing tools (Amazon X-Ray and Azure Application Insights), the results of this thesis relies entirely on timestamps recorded internally by the services used. This is not necessarily a problem with monolithic systems but in distributed systems using microservices, the clocks are not necessarily synchronized. Another potential effect on the timing of the internal clocks is that which comes from the observer effect [25]. Using tools to observe the internals of a system, which is necessary when tracing an invocation, increases the instrumentation overhead and could further delay events in the traces.

The difference in the amount of trace detail offered by AWS X-Ray and Azure Application Insights also poses a threat to validity as it severely limits the number of possible experiment configurations. This in combination with the lack of information regarding the details of existing traces makes it difficult for users to know what the system is doing "under the hood", and in extension, makes it difficult to analyze the traces on a lower level.

7.4.3 External Validity

As now established, the performance of triggers in two serverless computing providers varies between different types of services and between providers. From these results, we can assume that the same will be true when testing other services and providers. It is, however, difficult to relate the results from this thesis to other similar experiments using other services or providers. It is also difficult to draw any conclusion about other trigger types offered by Amazon Web Services and Microsoft Azure, as these are likely unrelated to the triggers tested in this thesis. The performance of other services and providers can easily be measured by extending the benchmark methodology presented in Chapter 5 to include more services and providers. This is made possible by the high level of automation and usage of IaC to ensure consistent

workflow between providers.

Some justifiable generalizations can be made with these results, however, that are related to the policies of each provider. The first such generalization is the rate at which traces are completed upon an invocation. This is related to the Quality of Service (QoS) and its ability to handle surges of invocations. In this case, none of the two providers performed particularly well, especially not if a delivery guarantee is required. Another generalization is the likelihood of having extreme outliers in the performance results. AWS generally had few outliers and Azure was far more likely to have extreme outliers. This can be seen in the standard deviations in Table 6.1 and in the cumulative distribution functions in Figure 6.2 and 6.4.

7.5 Reproducibility

As discussed in Chapter 3, effort has been put into ensuring that a similar study can be made in the future under the same conditions. Therefore, in this section, the reproducibility of this study will be discussed in terms of the eight reproducibility principles developed by Papadopoulos et al. [8]. First, a principle will be presented the same way as in Chapter 3, followed by a discussion related to the methodological approach.

P1. Repeated experiments: The experiment should be repeated an appropriate number of times with the same configuration and the results should be quantified.

When describing P1 in their article, Papadopoulos et al. state the importance of quantifying the significance of experiment repetitions but never go into detail of how it would be quantified. Quantifying the level of saturation in a sample is no trivial task, especially when the samples are not normally distributed as in our case. Moreover, this requirement is not mentioned again and the analysis of P1 is changed to instead focus on the presence of repeated experiments and long experiment runs. Basing the evaluation of P1 on this, instead of quantifying, the sample sizes can be considered sufficient because each benchmark was repeated twelve times (3 times per day $\times$ 4 days) and each repetition consisted of 1000 invocations, resulting in a total of 72000 invocations (1000 invocations $\times$ 12 repetitions $\times$ 6 benchmarks) with a loss of roughly 2.1%.

To further strengthen the statement above, one can still try to quantify the sample saturation by performing tests to find the confidence interval of the actual population mean compared to the sample mean. Performing both a Z-test and bootstrapping, the results show narrow confidence intervals around the sample means and both tests show similar results, as can be seen in Table 7.1. The bootstrapping method was based on the second method described by Efron and Tibshirani [26], where the number of generated samples was set to the number of samples in the original data sets. The tests indicate a high confidence that the sample means are close to the population means.

	AWS HTTP	AWS Storage	AWS Queue	Azure HTTP	Azure Storage	Azure Queue
Z-test 99% confidence interval (as % of mean)	±2.81%	±0.63%	±3.11%	±11.0%	±1.18%	±6.52%
Bootstrapping 1st & 99th percentile intervals (as % of mean)	-2.69% to 3.03%	-0.64% to 0.64%	-3.00% to 3.29%	-10.65% to 11.47%	-1.16% to 1.19%	-6.47% to 6.64%

Table 7.1: Confidence intervals of the true population mean for each of the six benchmarks expressed as a percentage of its mean.

The results from the Z-tests and bootstrapping are not meant to prove a sufficient level of saturation, but instead to give some indication of it. While some results from this test were quite high, likely due to extreme outliers, we can be confident that the sampled results accurately represent the actual population. Also, the large number of samples further increase the credibility of this. The number of invocations is, therefore, considered appropriate.

P2. Workload and configuration coverage: The experiment should cover a space of different possible configurations, possibly through randomization.

The configuration space of this thesis are in some ways limited due to service and provider-specific features. For example, when studying the performance of storage triggers there is only one service to try for every provider. Additionally, the choice of a specific service often implies the use of other related services, such as a storage event detection service that is specific to that storage. Still, some configurations related to the invocation are possible and could motivate further research, such as invocation patterns, polling rates, payload sizes, and to further extend measurements over more times and days. Overall, however, the aim of this thesis has not been to try every combination of services possible offered by a provider, but to make the configurations as similar as possible between providers and to make them truthful to real use-cases.

Some independent variables that were tested include the time of day and day of the week. These were included because the greatest load may be put on the provider during working days, and at specific times during those days.

P3. Experimental setup description: The setup of hardware and software should be described at an appropriate level of detail.

The experimental setup and how external variables and factors are handled have been discussed in detail in Chapter 5 and Section 6.1.

P4. Open access artifact: A (at least) representative subset of the developed software should be made publicly available.

The source code of our benchmarks and the data set can be found at

www.github.com/oskgro/faas-trigger-study [27].

P5. Probabilistic results description of the measured performance: Report a characterization of the empirical distribution of the measured performance.

All results have been presented in Section 6.3 both as distribution plots and cumulative distribution functions. More plots can be found in Appendix A.

P6. Statistical evaluation: Provide a statistical evaluation of the significance of the obtained results.

Statistical evaluations and the statistical significance is discussed in Sections 6.3 and 7.4. However, an extensive statistical analysis of the results is not very valuable as simple metrics such as mean, median and standard deviation, etc, are more relevant to real life scenarios and applications. The analysis done in this thesis should, therefore, be sufficient for those interested in the usage of FaaS.

P7. Measurement units: For all reported quantities, report the corresponding unit of measurement.

The corresponding units of measurement are presented in all figures and tables.

P8. Cost: The modeled cost of running the experiments should be included.

As the resource usage of the tests run in this thesis is relatively low, the free tier offered by both Amazon Web Services and Microsoft Azure should be sufficient to perform the benchmarks. In Microsoft Azure, the user is given an amount of credit to spend and the total cost is therefore displayed. In AWS, however, the total cost must be approximated. The AWS bill reached an approximate total of $0.29 and the Azure bill reached a total of $0.72.

8

Conclusion

The performance in which a service can trigger a function is greatly affected by its type and the cloud provider. For a software architect deciding between two or more similar services from different providers, the choice he or she makes can affect the overall performance of the system considerably. The results are also of interest to cloud providers where performance is an important aspect to differentiate from competitors. Users are sensitive to large delays and large delays could cost cloud providers several percent in lost revenue. Serverless applications are likely to see continued growth in usage across providers and as triggers serve an important purpose in them, we argue that due to the above reasons, trigger performance is a crucial area for improvement for continued adoption.

To answer the research questions defined in Section 1.2, a benchmark design was proposed and evaluated using controlled experiments. A total of 72000 invocations were performed to determine how the choice of service and provider affects the trigger delay. The answers to our research questions can be summarized as:

RQ1: How can function trigger latency be measured across different serverless platforms?

To design a fair benchmark the structure must be as simple as possible and any excess features will increase the likelihood of provider-specific effects on the results. There will always be differences that are caused by provider-specific proprietary features but as long as each benchmark is built for the same purpose and uses similar services (and possibly IaC tools like Pulumi) the comparisons can be made fair.

RQ2: How does the choice of trigger type affect trigger latency?

Out of the three different trigger types tested, the HTTP trigger performed the best for both providers. The Queue trigger performed second-best for AWS and third-best for Azure. The Storage trigger was the third-best trigger type for AWS and the second-best trigger type for Azure.

RQ3: How does the choice of provider affect trigger latency?

The two providers studied in this thesis, Amazon Web Services and Microsoft Azure, both performed similarly in terms of mean trigger delays. However, Microsoft Azure

was far more likely to have extreme outliers reaching delays of up to 10 times the mean, affecting the measured mean and standard deviation as well as lowering its consistency. In terms of trigger performance, AWS is the better option for HTTP and queue triggers, while Azure is the better choice for storage triggers. However, it is important that cloud developers and architects consider the trade-off between speed and predictable performance.

8.1 Future Work

Based on the limitations presented in Section 1.3, future work in researching serverless and FaaS could preferably study more trigger types and more providers in addition to those studied by this thesis. Future work could, for example, study Google Cloud Platform or trigger types such as SDK/CLI triggers. This would give a broader view of trigger performance and further facilitate architectural and design decisions by developers, system architects, and providers. As a base for future research, we also suggest repeating the experiments of this thesis as future updates and evolving infrastructure are likely to affect trigger performance.

As previously discussed, incomplete traces were excluded from the results of the experiments in this thesis. A total of 2.1% of all traces were lost due to being incomplete or due to sampling. Future work could study the correctness of traces, why, how, and how many traces end up incomplete or sampled, as well as how configurable settings can affect these results. The order in which invocations are received and handled by the triggered function was also ignored in this thesis but is highly relevant for some applications. A study on the degree to which different services and providers can accommodate a strict order of execution would, therefore, also be beneficial.

Similar to how Azure Event Grid was used as Storage trigger in favor of the more traditional Azure Blob Storage, it would be interesting to study the AWS equivalents AWS SNS and AWS Event Bridge to see if they add any extra latency when used as triggers. Instead of uploading an object to AWS S3 from an invoker Lambda function to trigger a receiver Lambda function, future experiments could upload an object to S3 from the invoker function, which would send an event to SNS or Event Bridge that finally triggers the receiver Lambda function.

In this thesis, independent variables such as the day of the week and the time of day were considered. Aside from studying temporal effects, future work could study other interesting variables that could affect trigger performance. One such variable could be invocation patterns; how invoking the benchmarks in certain patterns (constant invocation, in bursts, spiky, etc.) affects trigger performance. In our experiments we relied on the default settings for all triggers, another variable could, therefore, be trigger configuration to study how relevant configurable parameters regarding polling, batching, or other aspects affect trigger latency. As previously discussed, the server location was ignored in this thesis. It is possible that trigger latency is affected by the server location, especially when the invoker and receiver functions are located in different geographical regions.

Bibliography

[1] S. Bhardwaj, L. Jain, and S. Jain, "CLOUD COMPUTING: A STUDY OF INFRASTRUCTURE AS A SERVICE (IaaS)," *International Journal of Engineering and Information Technology*, vol. 2, no. 1, pp. 60–63, 2010.

[2] Pulumi, "Architecture & concepts." https://www.pulumi.com/docs/intro/concepts/. Accessed Mar 2021.

[3] Statista, "Global market share of cloud infrastructure services from 2017 to 2020, by vendor." https://www.statista.com/statistics/477277/cloud-infrastructure-services-market-share/. Accessed Jan 2021.

[4] Google, "Measure performance with the RAIL model," 2020. https://web.dev/rail/. Accessed May 2021.

[5] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Association for Computing Machinery*, vol. 62, p. 44–54, Nov. 2019.

[6] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A mixed-method empirical study of function-as-a-service software development in industrial practice," *Journal of Systems and Software*, vol. 149, pp. 340 – 359, 2019.

[7] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," *Journal of Systems and Software*, vol. 170, 2020.

[8] A. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. von Kistowski, A. Ali-Eldin, C. Abad, J. Amaral, P. Tůma, and A. Iosup, "Methodological principles for reproducible performance evaluation in cloud computing," *IEEE Transactions on Software Engineering*, vol. PP, Jul 2019.

[9] E. van Eyk, J. Scheuner, S. Eismann, C. L. Abad, and A. Iosup, "Beyond Microbenchmarks: The SPEC-RG Vision for a Comprehensive Serverless Benchmark," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, (New York, NY, USA), p. 26–31, Association for Computing Machinery, 2020.

[10] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the

wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 {USENIX} Annual Technical Conference*, {USENIX}{ATC} 20, pp. 205–218, 2020.

[11] EXOR, "Analysis of the paradigm shifts in software and business development," Nov 2019. https://www.exorint.com/en/blog/paradigm-shifts-in-software.

[12] G. Mao, "Understanding the different ways to invoke lambda functions," 2019. https://aws.amazon.com/blogs/architecture/understanding-the-different-ways-to-invoke-lambda-functions/. Accessed May 2021.

[13] Pulumi, "Why pulumi?." https://www.pulumi.com/why-pulumi/. Accessed Mar 2021.

[14] Microsoft Azure, "What is distributed tracing?," 2018. https://docs.microsoft.com/sv-se/azure/azure-monitor/app/distributed-tracing. Accessed Apr 2021.

[15] I. Pelle, J. Czentye, J. Dóka, and B. Sonkoly, "Towards latency sensitive cloud native applications: A performance study on aws," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 272–280, 2019.

[16] J. Nielsen, "Response times: The 3 important limits," 1993. https://www.nngroup.com/articles/response-times-3-important-limits/. Accessed May 2021.

[17] Jupiter Research, "Retail web site performance: Consumer reaction to a poor online shopping experience," 2006. https://www.akamai.com/us/en/multimedia/documents/report/akamai-site-abandonment-final-report.pdf. Accessed Apr 2021.

[18] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne, "Controlled experiments on the web: survey and practical guide," *Data Mining and Knowledge Discovery*, vol. 18, p. 140–181, 2009.

[19] EXOR, "Empirical standards," Nov 2019. https://www.exorint.com/en/blog/paradigm-shifts-in-software.

[20] Microsoft, "Aws to Azure services comparison," Nov 2020.

[21] M. Shilkov, "Cold Starts in AWS Lambda," 2021. https://mikhail.io/serverless/coldstarts/aws/. Read Apr 2021.

[22] Microsoft Azure, "Azure Application Insights REST API." https://dev.applicationinsights.io/. Visited May 2021.

[23] k6. https://k6.io/.

[24] H. K. Wright, M. Kim, and D. E. Perry, "Validity concerns in software engineering research," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, (New York, NY, USA), p. 411–414,

Association for Computing Machinery, 2010.

[25] A. Najafi, A. Tai, and M. Wei, "Systems research is running out of time," in *Workshop on Hot Topics in Operating Systems*, HotOS '20, Association for Computing Machinery, 2020.

[26] B. Efron and R. Tibshirani, "Bootstrap Methods for Standard Errors, Confidence Intervals, and Other Measures of Statistical Accuracy," *Statistical Science*, vol. 1, no. 1, pp. 54 – 75, 1986.

[27] M. Bertilsson and O. Grönqvist, "faas-trigger-study," *GitHub repository*, 2021. https://www.github.com/oskgro/faas-trigger-study.

A

Appendix 1

A.1 Individual Results

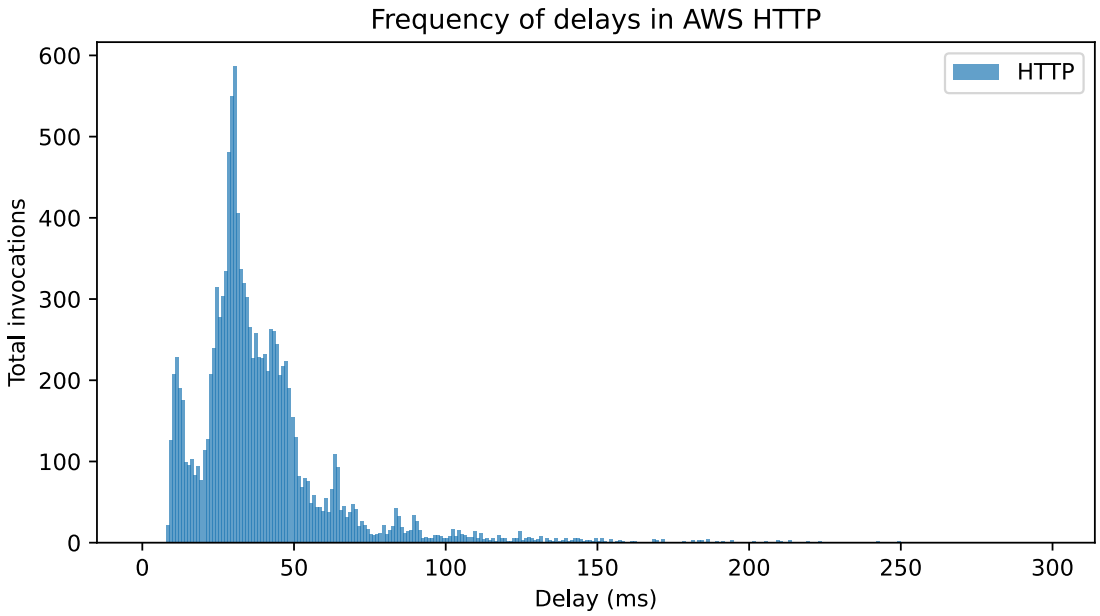


Figure A.1: Combined AWS HTTP benchmark results.

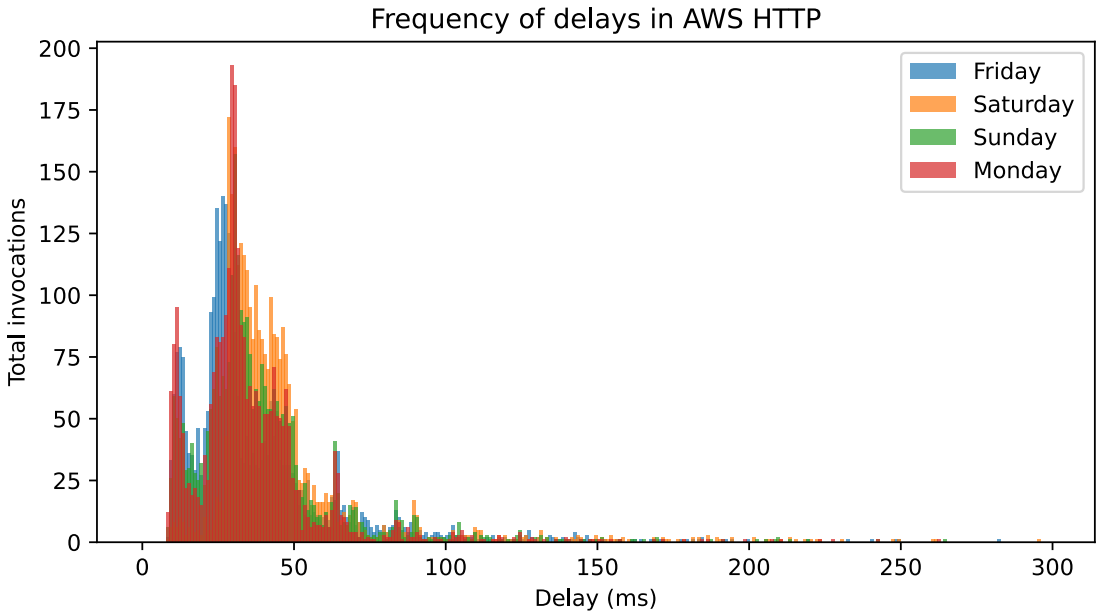


Figure A.2: AWS HTTP benchmark results for different days.

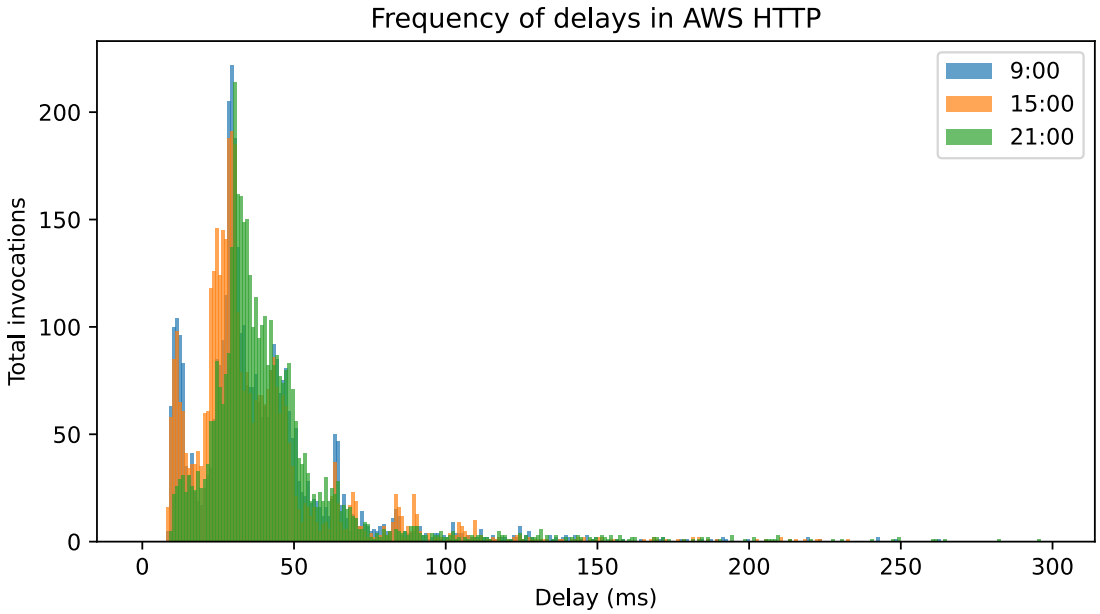


Figure A.3: AWS HTTP benchmark results for different times.

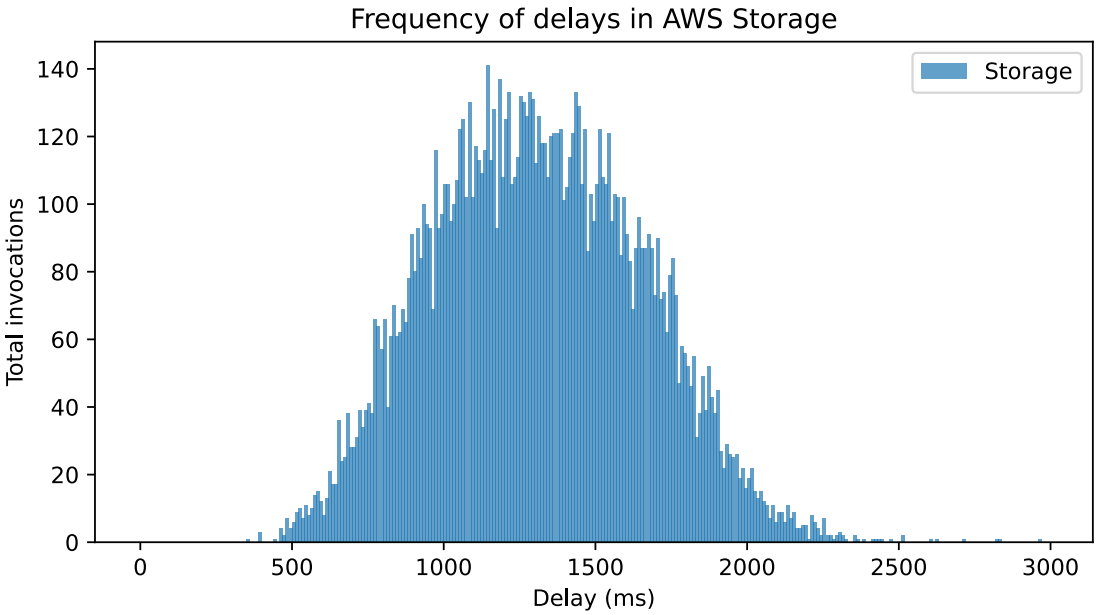


Figure A.4: Combined AWS Storage benchmark results.

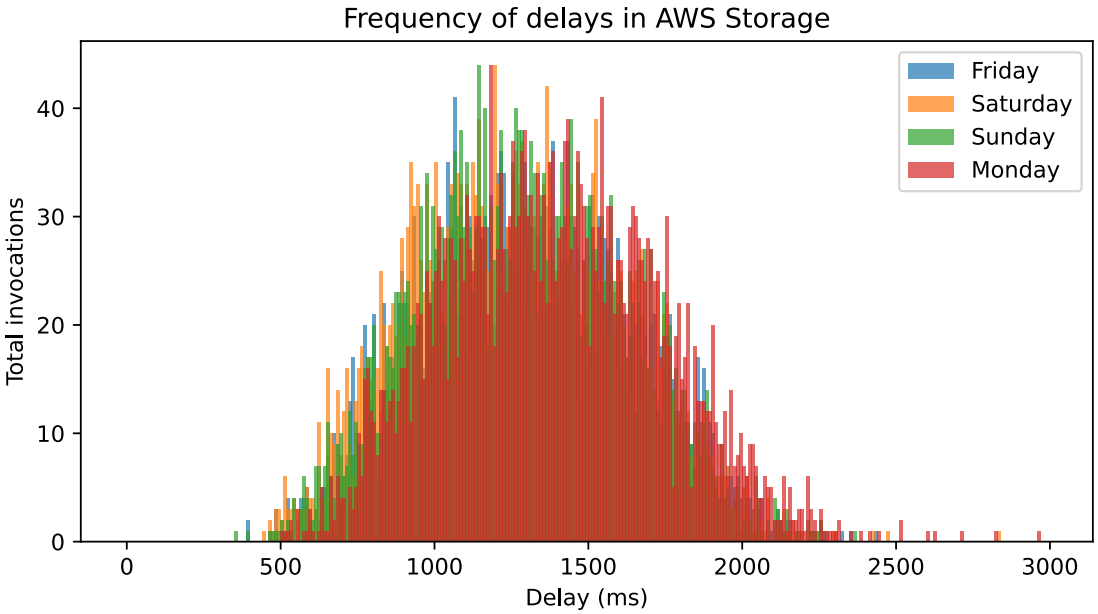


Figure A.5: AWS Storage benchmark results for different days.

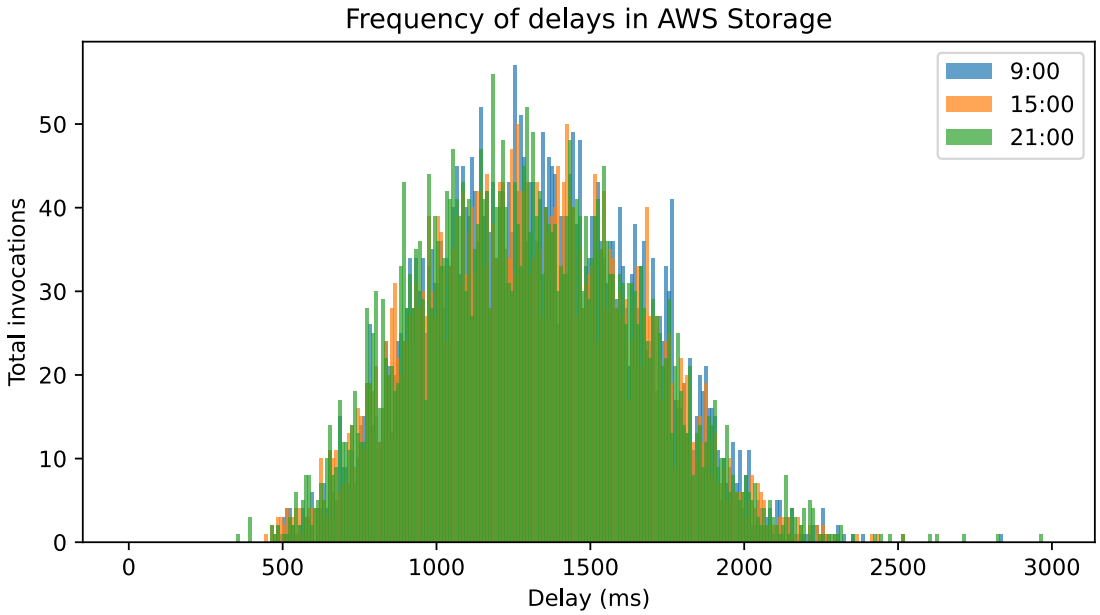


Figure A.6: AWS Storage benchmark results for different times.

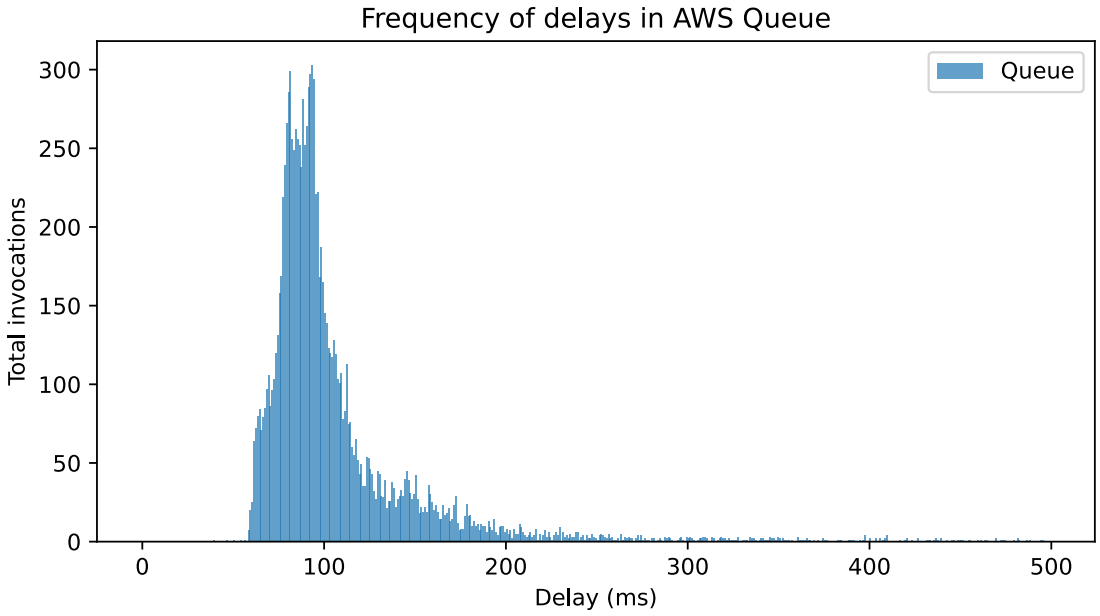


Figure A.7: Combined AWS Queue benchmark results.

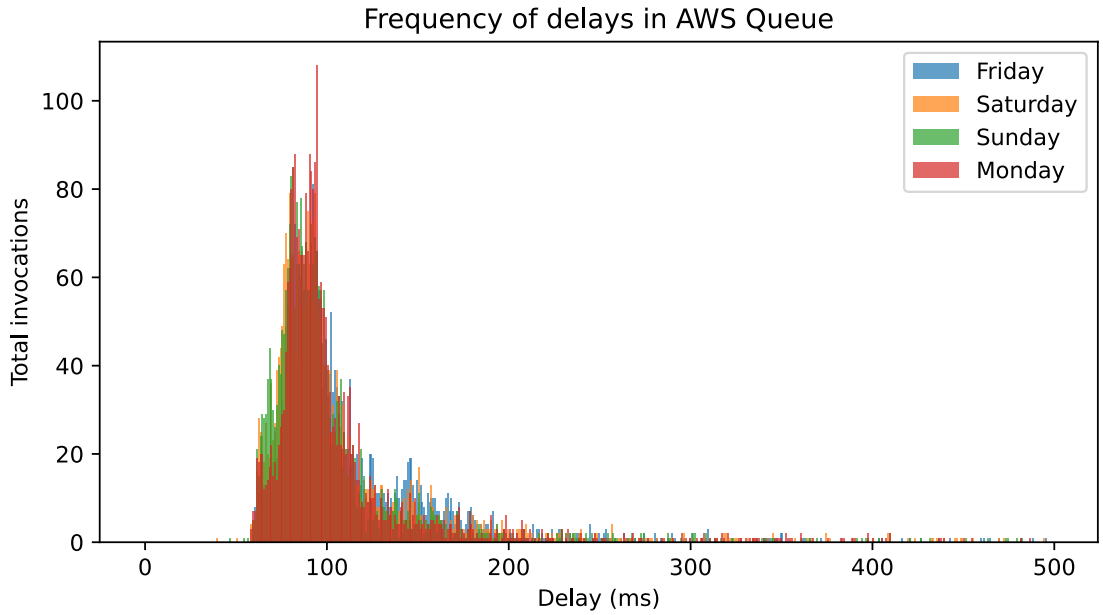


Figure A.8: AWS Queue benchmark results for different days.

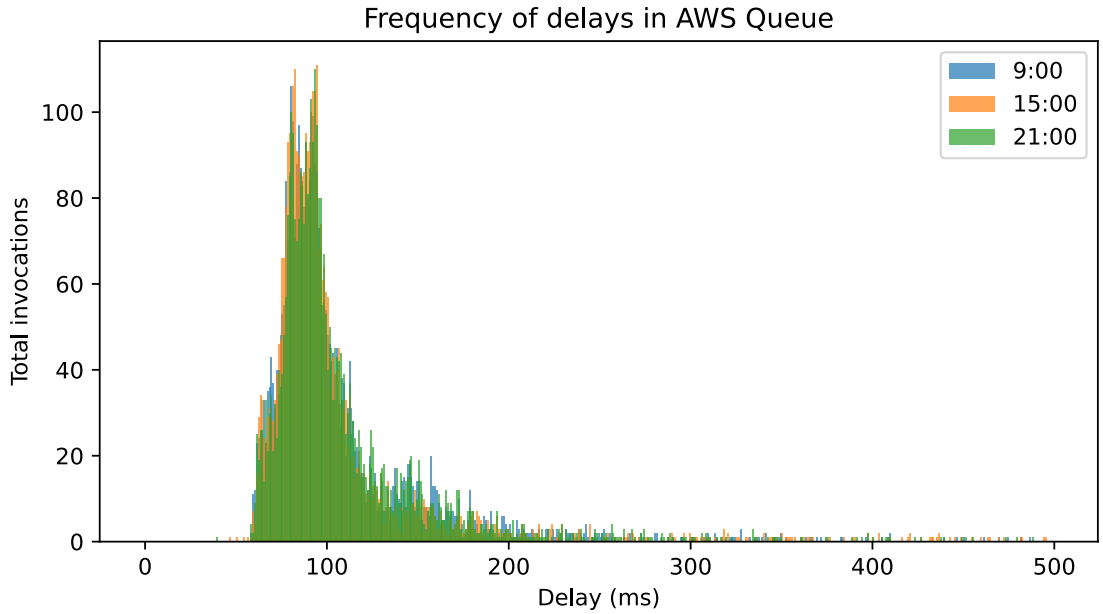


Figure A.9: AWS Queue benchmark results for different times.

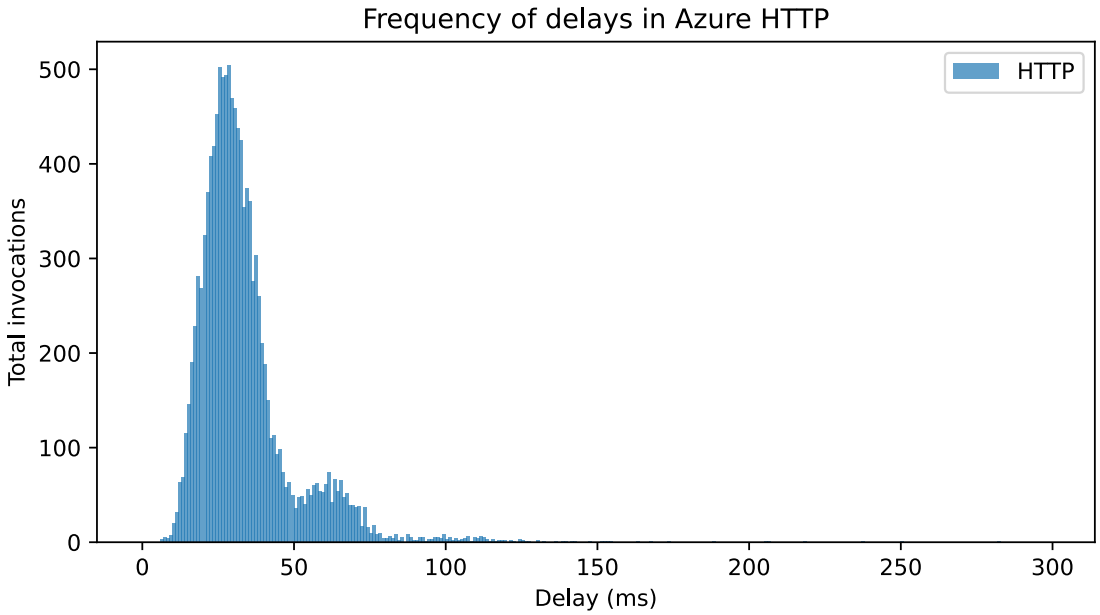


Figure A.10: Combined Azure HTTP benchmark results.

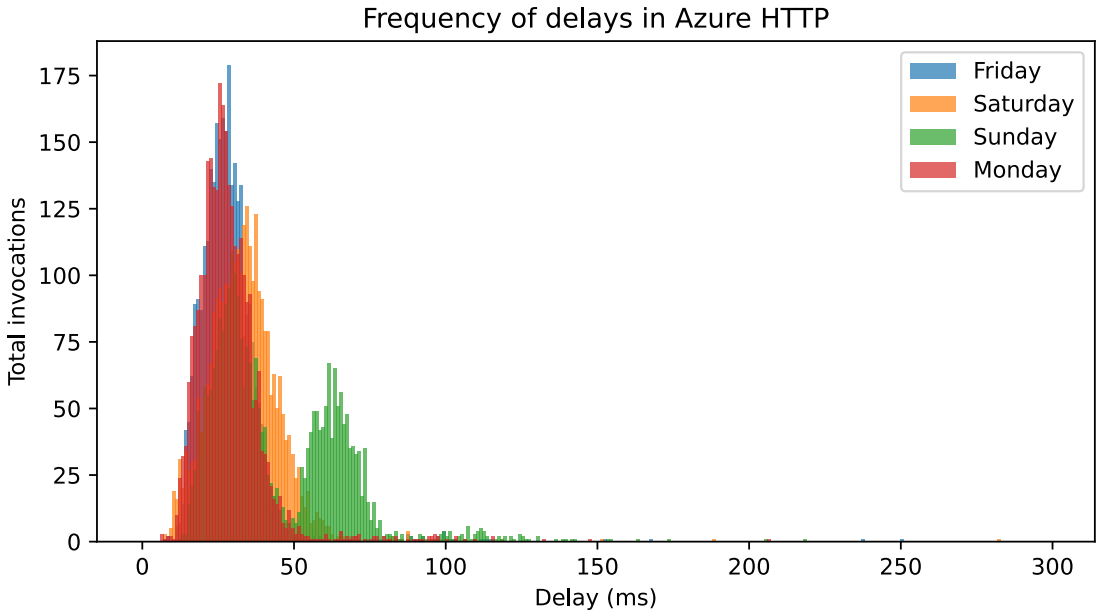


Figure A.11: Azure HTTP benchmark results for different days.

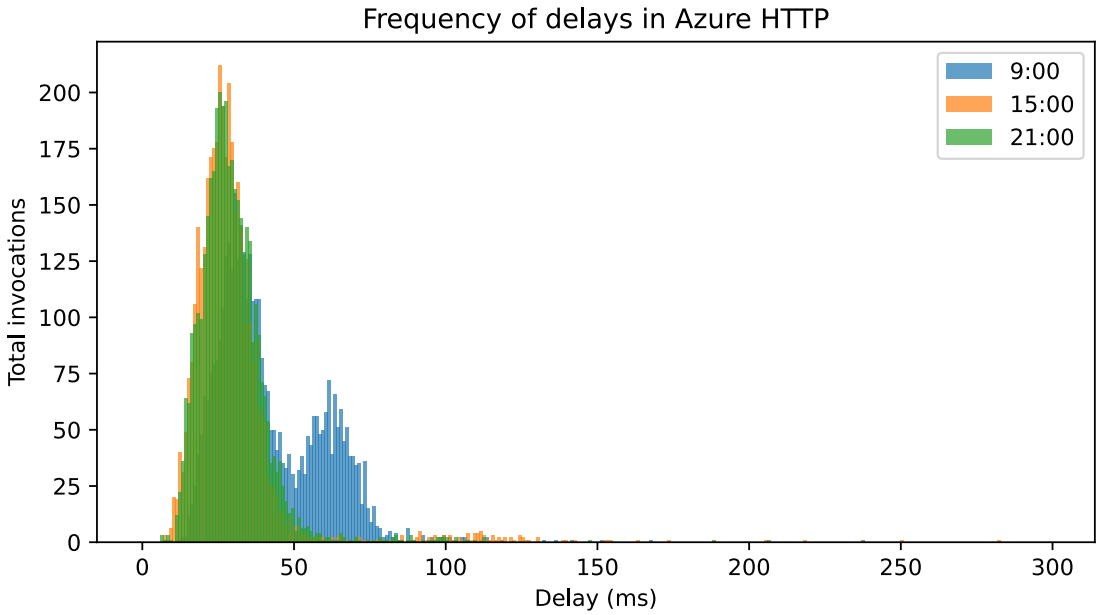


Figure A.12: Azure HTTP benchmark results for different times.

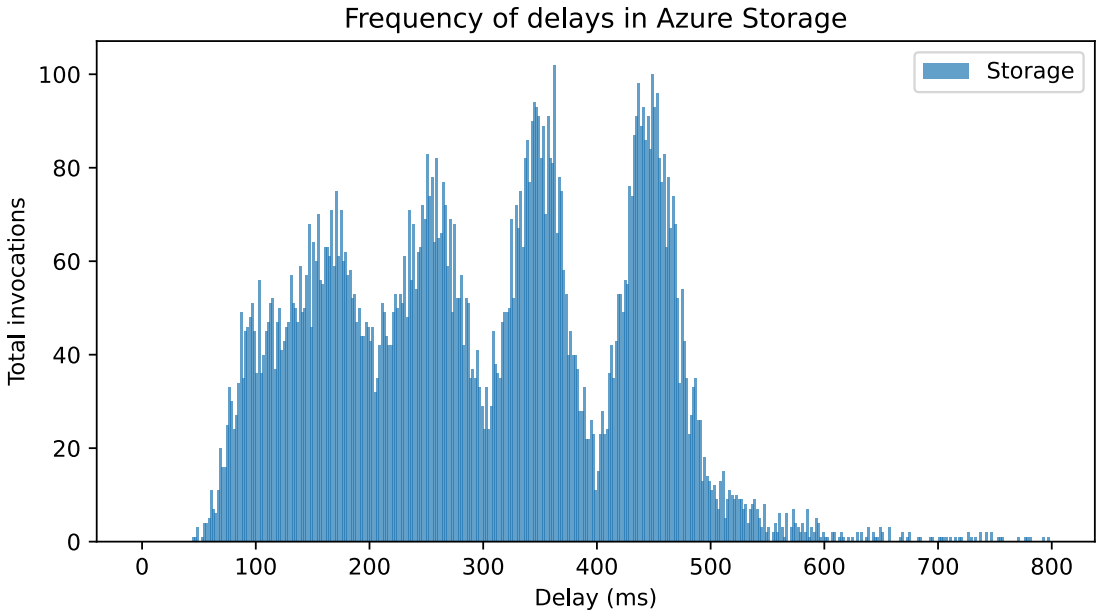


Figure A.13: Combined Azure Storage benchmark results.

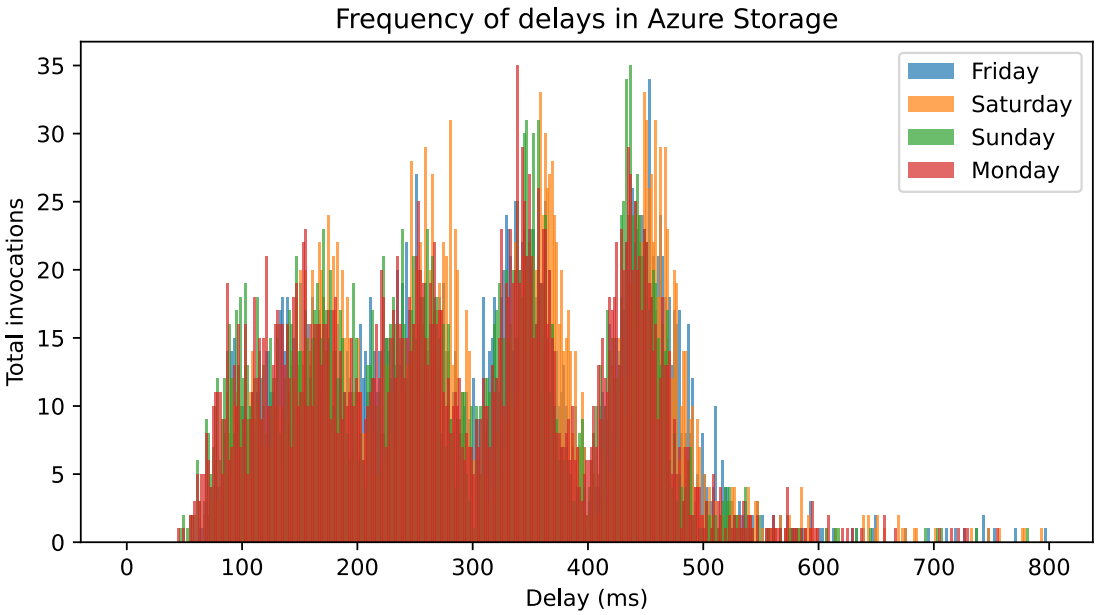


Figure A.14: Azure Storage benchmark results for different days.

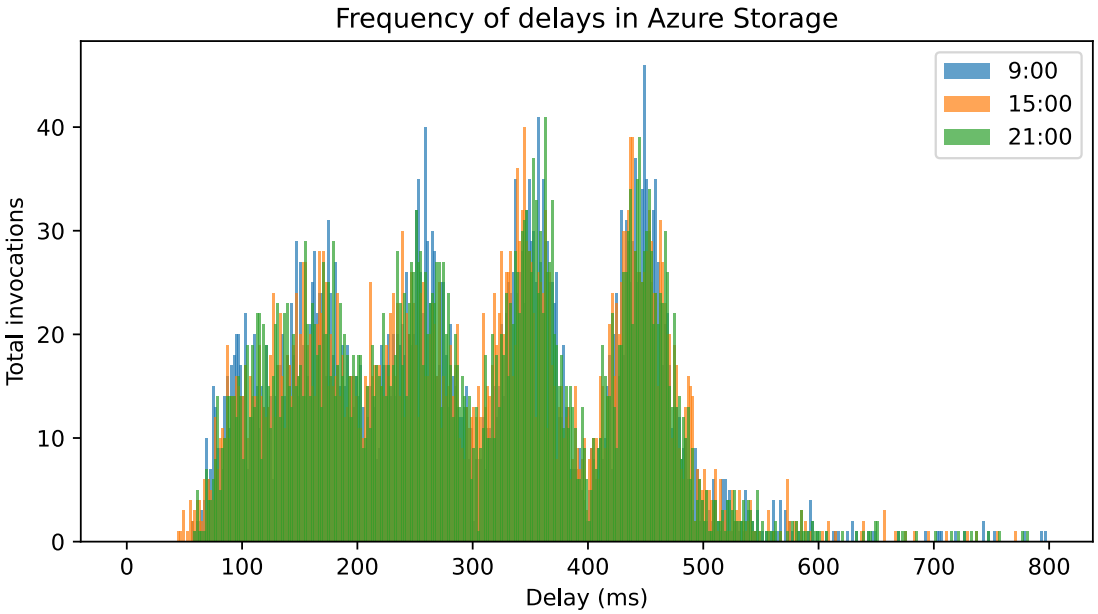


Figure A.15: Azure Storage benchmark results for different times.

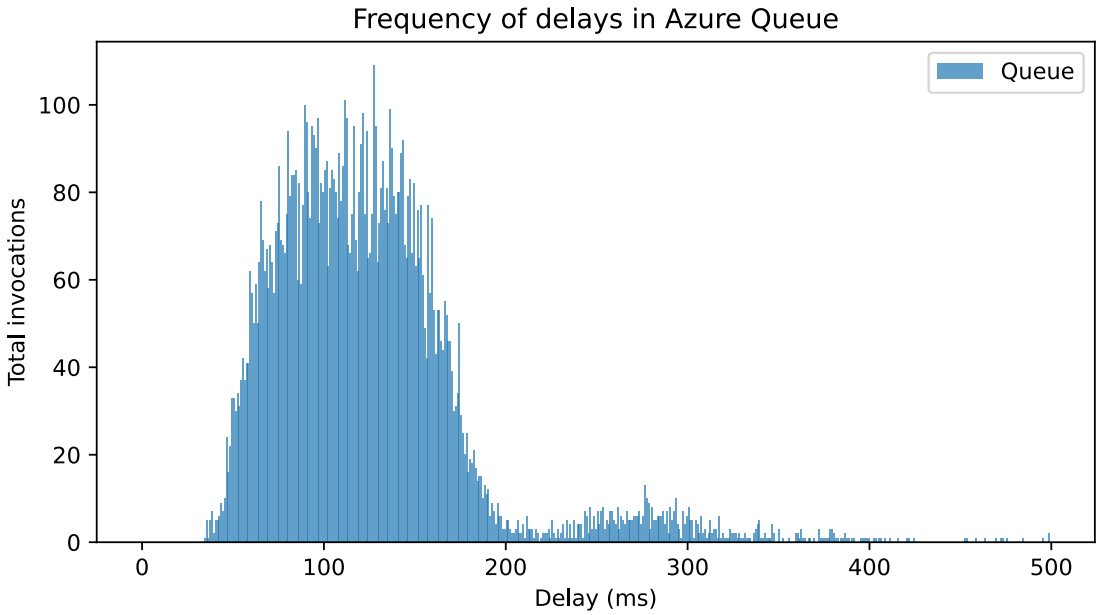


Figure A.16: Combined Azure Queue benchmark results.

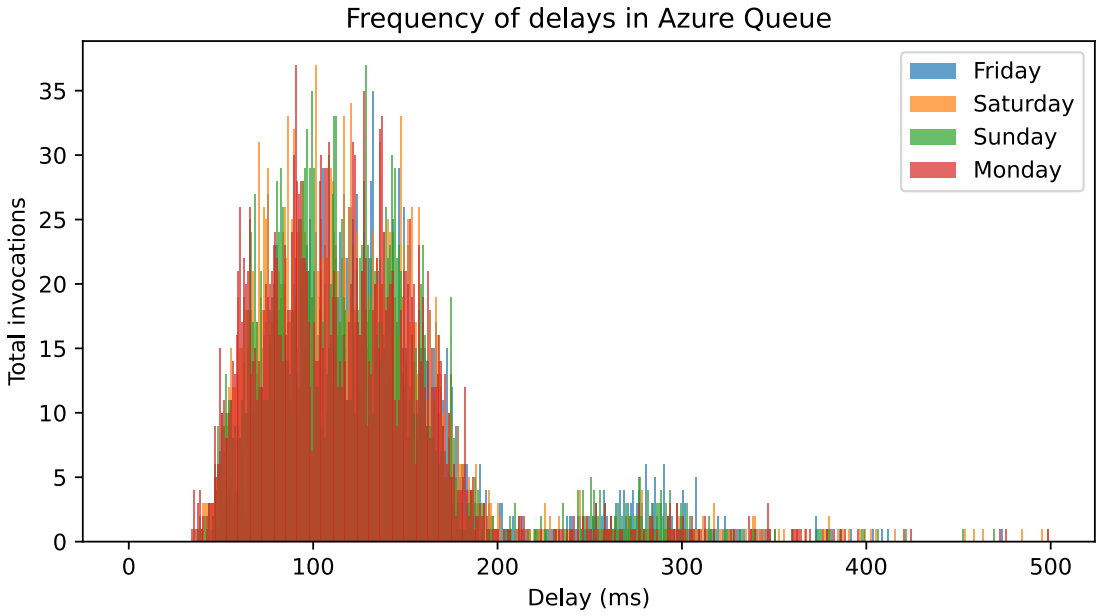


Figure A.17: Azure Queue benchmark results for different days.

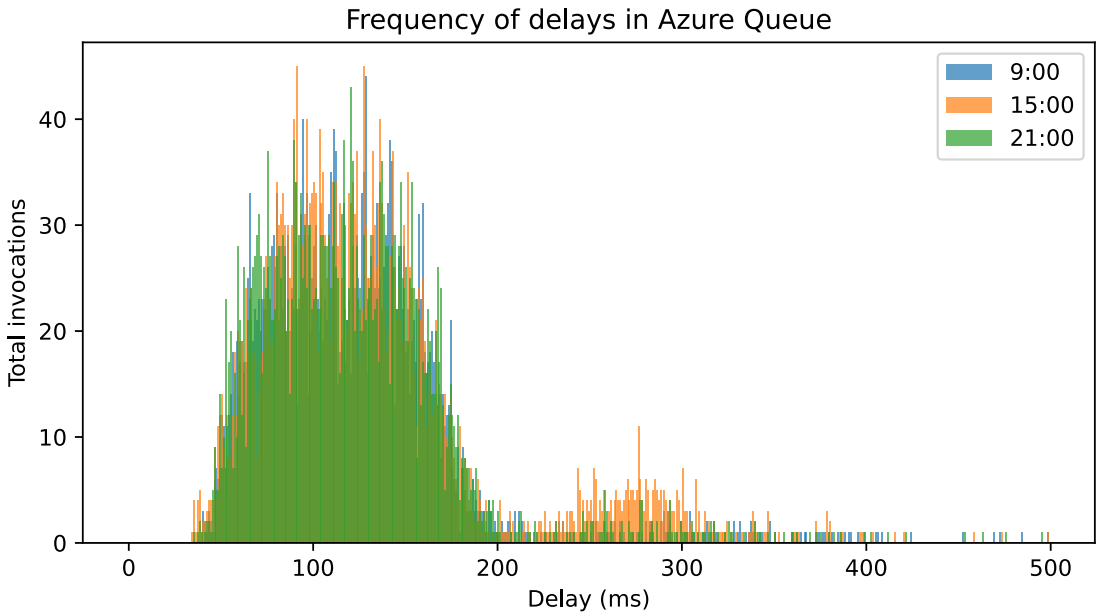


Figure A.18: Azure Queue benchmark results for different times.