Universiteit Leiden

Master Computer Science

Cross-provider Serverless Trigger Benchmarking

Name:	Dervis Onur Gurbuz
Student ID:	s3440524
Date:	23/08/2024
Specialisation:	Advanced Computing and Systems
1st supervisor:	Kristian Rietveld
2nd supervisor:	Rob van Nieuwpoort
External :	Timo Heijne

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Serverless Computing is a cutting-edge concept in cloud computing that greatly simplifies application development. A prominent issue in serverless computing is the length of cold starts, leading to unusable applications and unexpected latencies due to inactive functions. Since the introduction of Amazon's Lambda in 2014, the adoption of serverless platforms has surged, requiring performance comparisons across various providers, run times, and infrastructures.

Serverless performance has been well-researched in literature, however, there has been a gap in addressing cold start durations across various serverless trigger types by applying different invocation scenarios. This particular component has often been a significant deficiency in current studies, at most excluded or disregarded in the reporting of delay metrics. We seek to address this cold start gap by investigating cold start times in different run times and cloud providers.

Our research primarily focuses on cold start benchmarking to evaluate the performance of serverless applications hosted on AWS and Google Cloud Platforms. By analyzing important performance metrics like latency, we aim to gain insight into the duration it takes to process requests and send responses in different trigger types. To achieve this, we have implemented a cross-provider serverless benchmarker. Our comprehensive analysis revealed that AWS consistently provides lower latencies than GCP, with synchronous HTTP triggers outperforming asynchronous ones across all experiments. Also, we have developed an infrastructure composed of a chain of functions to illustrate how trigger latencies accumulate and impact overall response performance. We anticipate that the insights from this study will empower developers to make well-informed decisions regarding serverless computing.

Acknowledgements

Since the beginning of my academic adventure, I have had a deep fascination with cloud computing and its capacity to develop modern software. This interest was significantly motivated by the cloud computing course taught by Kristian Rietveld, whose inspiring teaching profoundly impacted me to pursue my thesis under his supervision. Foremost, I extend my sincere thanks to Kristian for his consistent support, guidance, and valuable perspectives throughout my research journey. I would also like to thank my supervisor at Accenture and all of my colleagues there who helped me either by word or deed for their guidance, supervision, and help.

Additionally, I am grateful to my friends and family for their support during these challenging times. Their support and insightful critique have been vital in improving my ideas and elevating the quality of this research.

Contents

1 Introduction

Serverless computing has garnered considerable attention due to its capacity to abstract infrastructure administration and enable developers to concentrate on coding. Within this domain, Function as a Service (FaaS) enables the deployment of individual functions that automatically scale according to demand. These functions can be activated in response to specific events or triggers, embodying the event-driven nature of serverless computing. Given this background, the research community has shown significant interest in benchmarking FaaS platforms. The motivation behind this interest stands for the fact that each cloud provider provides its Function as a Service (FaaS) architecture, frequently with distinct components, such as specific sorts of triggers. Triggers are events that initiate the running of a function, such as HTTP requests, modifications to a database, or updates to a messaging queue. In the public cloud sector, there are two primary types of function triggers: synchronous triggers and event-driven (asynchronous) triggers. These triggers have different performance characteristics and are used for certain purposes. In synchronous triggers, the execution waits for the completion of the previous function call before initiating a new one. On the other hand, event-driven triggers, built asynchronously, allow functions to call in response to specific events without the need to wait for prior calls to complete. The distinction between synchronous and asynchronous triggers allows for flexible design and deployment of cloud applications to accommodate varying performance requirements.

Another hot topic in the public cloud industry is the cold start and warm start latency. Cold start latency occurs when a function calls for the first time or after a period of inactivity. This means that the cloud provider has to initialize a new container, which causes an unexpected latency. On the other hand, we expect warm start in idle stage functions to significantly speed up response times when we call a function while an idle instance remains accessible. Understanding the differences between warm start and cold start latency is essential for maximizing serverless application responsiveness and performance for latency-sensitive implementations. Contrary to prior studies that evaluated multiple trigger types, we pay special attention to the cold start delay. Based on our comprehensive literature analysis in Section 3, existing research has not investigated the cold start delays about various trigger types. We believe this is essential because even when a function is triggered infrequently, it can be a component of a latency-sensitive application or part of an extensive chained microservice or workflow.

Our research aims to fill this cold start gap by investigating cold start times in different run times and cloud providers. By systematically triggering functions in a sequence, as described in Section 4.2.3, we can measure the latency and performance differences between cold and warm starts.

This thesis has been accomplished via collaboration with Accenture[1], which is a multina-

[1] https://www.accenture.com/nl-en

tional professional technology consulting company that specializes in information technology (IT) services, and consulting to help businesses thrive in the digital age by leveraging its expertise in technology, strategy, and operations. In their workflow, they build many custom applications for their clients by leveraging the serverless architecture offered by a variety of cloud providers. Efficiently handling and improving the time it takes for an application to start from a cold state is essential for applications that require low latency such as geospatial technology[2], mobile/web applications, and financial applications. Because a slow response rate can directly influence user experience, create financial losses, or result in inaccurate location data. Within this particular context, 'latency sensitive' refers to applications that necessitate immediate and timely replies. A study by Arapacis et al. [ABC14] found that web search users do not typically notice delays of up to 500 milliseconds; however, latency exceeding 1000 ms is mostly perceptible. Seven years later, the same researchers explored mobile web searches [APP21], finding users show four times greater tolerance for delays ranging from 1139 to 1709 milliseconds compared to desktop settings. Scholtus et al. [SvD12] noted that in high-frequency technical trading, a delay as brief as 200 ms can significantly decrease profitability. Therefore, surpassing these maximum limits might result in reduced user satisfaction, higher bounce rates, and possible financial losses, especially in competitive settings.

We believe the evaluation of trigger types is critical because they serve as fundamental components shared among various serverless functions. This paper introduces a benchmarking solution to assess serverless trigger performances while deploying serverless cloud functions, apply workload scenarios while triggering functions, test cold starts, and evaluate cold start performance metrics. This method sets long-lasting schedules to activate cloud functions, record their traces, and retrieve, and analyze these traces to investigate trigger latency. To conduct a comprehensive evaluation, the research will examine several types of triggers, such as HTTP requests, queues, storage, and database triggers, to determine their effect on systems that require low latency. By conducting experiments on AWS and GCP, we have evaluated how the latest run time environments affect performance measures, including cold start times, warm start times, and overall efficiency.

In the initial phase of our research, we have successfully reproduced the latest serverless benchmarking research conducted by Joel Scheuner et al. called TriggerBench [JS22]. The study offers a strong foundation for examining the performance of serverless systems. However, their implementation was meant for warm start invocations where they create cloud resources, make experiments, and destroy these resources in a short period. Additionally, while reproducing Joel Scheuner et al.'s experiment, we found the broad cloud authorization required by Pulumi [pul24] software, this enables the program to generate or remove existing

[2]Geospatial technology refers to a set of technologies used to generate datasets, such as point clouds, which are used for mapping and analyzing the Earth. These technologies facilitate data collection to get insights into geographic areas and track changes or patterns in landscapes, cities, and societies.

resources, such as cloud applications, cloud functions, databases, buckets, and allocate roles for cloud operations, such as granting permission to a new cloud function to edit databases or modify content. The additional Accenture regulations and our relative design decisions have been detailed in Section 4.

In contrast to TriggerBench research results, we have also utilized several frameworks in Section 5.1.2 and dependencies to explore their impact on cold start durations. This entails using the official AWS Software Development Kit (SDK)[3] [awsb] and Google [gooa] JavaScript frameworks. These frameworks were selected due to their extensive use in cloud applications and their pivotal role in managing cloud operations. We aim to address the latency implications of these frameworks, particularly focusing on the delays that occurred during cold starts by the loading and deployment of dependencies within serverless function containers.

The main objective of this research is to thoroughly assess the efficiency of serverless infrastructures, notably AWS and Google Cloud Platform (GCP), under different conditions.

Following this research, the primary research questions are:

1. What is the impact of trigger types, frameworks, and cloud providers on the cold/warm start latency of serverless functions, and how does this affect their suitability for applications that require low latency?

2. What is the impact of cold start durations on the serverless trigger latency, and how does this vary across different cloud providers (such as AWS and GCP)?

3. What is the comparative cold start performance of serverless functions across the latest runtime environments (such as Node.js 20, Python 3.12, Java 21)?

4. What is the cold start effect involving long chains of sequential serverless trigger executions?

To create better solutions our study's goal is to find out if serverless architectures can meet the latency needs of these applications. This includes looking at how long tasks take to start and run in response to events. The goal is to determine if serverless functions can provide the necessary degree of responsiveness and performance without impacting the user experience or the effectiveness of the application in situations where low latency is crucial. We have also evaluated how different trigger types and run-time conditions affect performance measures, including cold start times, warm start times, and overall efficiency.

Additionally, we compare the performance of AWS and Google within the realm of Function-as-a-service (FaaS) infrastructures. By writing a trigger benchmarking software

[3]A software development kit (SDK) is a comprehensive suite of software development tools that may be installed together.

we were able to investigate the impact of cold start time, which refers to the time necessary to activate serverless functions, on the latency and responsiveness of applications deployed in a FaaS environment. To evaluate the latency factors in cold start durations we have implemented our experiments in different run times, compared two cloud providers, and proposed a chain of execution where multiple services run in a continuous sequence to show how the overall latencies add up on each other which makes a dramatic performance results.

Our experiments on cross-platform benchmarking and chain of functions in Section 5.5.2 revealed that AWS consistently provides efficiency in handling HTTP triggers, with minimal latency compared to GCP. However, both platforms face challenges with significant delays, ranging from 1500 ms to 2700 ms, when it comes to storage and database triggers. Additionally, we have observed that the presence of several delays in a chain of functions has a substantial impact on the overall efficiency of the system. The accumulation of these latencies might escalate, leading to sub-optimal system responsiveness.

This thesis is organized as follows: Chapter 2, focuses on the foundational concepts relevant to this research. Chapter 3, identifies the research gap by reviewing previous research related serverless triggers. Chapter 4, details the methodologies deployed. Chapter 5, describes the experimental setup and findings. Chapter 6, provides a summary and the key takeaways from our study. Chapter 7, describes the limiting factors during this study. Chapter 8, explains the areas for future research.

2 Background

The capabilities of serverless computing showed its potential as a crucial field of research and implementation, offering significant improvements in operational efficiency and development flexibility. However one of the significant problems in this architecture is the ”cold start” latency. This refers to unstable latencies that happen when the system allocates resources for newly activated or inactive functions to answer computational demand. To address this problem a related benchmarking implementation has to be implemented to observe GCP and AWS serverless performance metrics, and address the downsides which will guide possible architectural decisions.

This section examines the technical and practical elements of serverless computing, including its advantages, difficulties, and impact on future technological advancements.

2.1 Serverless Computing

In the rapidly evolving area of cloud computing, predominant models such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Serverless Computing have rationalized computation. IaaS provides users with the ability to access and utilize virtualized physical computing resources through the Internet. This allows users not to acquire and maintain physical hardware resources and enables them to rent virtual servers and other hardware products. This architecture offers the capacity to easily adapt and scale the management of virtual software resources. At a more abstract level, Platform as a Service (PaaS) offers developers a framework to create, test, and launch applications without the need to handle the underlying complicated hardware or software layers. This expedites software development cycles, which is particularly advantageous for many technology businesses. Furthermore, serverless computing goes even beyond PaaS by completely abstracting server administration. Serverless architecture automatically handles the allocation and provisioning of servers while the computation is adjusted automatically according to demand. It improves operational efficiency and cost-effectiveness, especially for applications that have varying workloads that do not require renting resources for long periods. On the other hand, PaaS provides more control over the deployment environment it has to be configured to scale automatically, and some demand forecasting is needed to scale properly.

Serverless computing allows customers to run apps that are triggered by different events and billed based on usage, without requiring to manage the operational details such as server provisioning, scaling, and maintenance. Erwin Van Eyk et al.’s study [EvEI18] indicates that serverless computing would not have been possible a decade ago because the necessary technologies were not available. These technologies include the differentiation between IaaS and PaaS as standardized by the National Institute of Standards and Technology (NIST),

fine-grained containerization like Docker, and a wide range of applications.

2.1.1 Serverless Functions

Function as a Service (FaaS), also known as cloud functions, is a distinct form of serverless computing in which individual functions are executed according to triggered events. Additionally, it is stateless, meaning that functions do not persist data between operations. Together with the events or invocations that provide the required data that must trigger these functions, this architecture guarantees horizontal scalability without requiring synchronization.

The event-driven architecture of serverless functions enables the creation of highly dynamic and reactive applications. Automatic scaling ensures that applications effectively handle varying demands. GCP and AWS's accurate billing services precisely calculate payments for users by considering the sensitive timing and utilization of resources. Additionally, the incorporation of integrated security elements and the existence of built-in logging and monitoring tools both empower execution management and data protection while optimizing speed.

Technically the serverless computing platforms such as AWS and Google Cloud, the serverless function and its dependencies are contained within an image. Cloud provider stores these images and when the event occurs they deploy the image in the servers. Some of the major event types have been discussed as trigger types in Section 2, such as HTTP, storage, queue, and database. The container and image approach guarantees that the container is equipped with all the essential components for the function to deploy properly in activation, thereby allowing for quick response and effective scalability. Additionally, the consolidation of the function and its dependencies into a single image simplifies administration. Alternatively, the serverless architecture allows for the deployment of AWS Lambda Functions via zip files. This process entails compressing the function into a zip file, comparing the hash of the existing files with the previously uploaded one, and only changes that have been applied. Also, ending the process in deployment failures and storing the older versions. This ensures efficient updates and facilitates the management of serverless functions.

When the function is triggered the relative zip folder has been used to initialize the AWS Lambda function. The initialization phase of the AWS Lambda function is shown in Figure 1, and during this process, three essential activities are executed: initiating all extensions (Extension Init), setting up the run time environment (Runtime Init), and executing the static code of the function (Function Init). After a while of inactivity, the Lambda functions shut down.

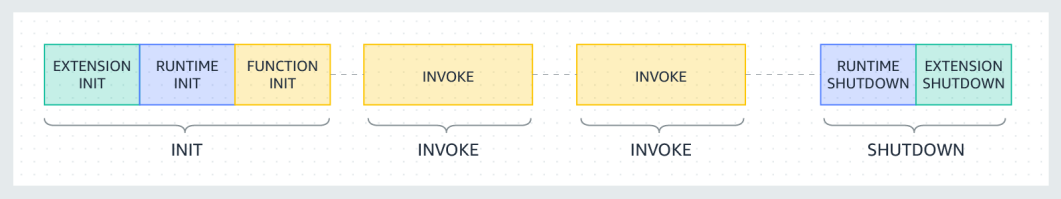

Figure 1: AWS Overview Successful Invokes [aws22]

Figure 2 illustrates the five major execution states of the cloud function. In the first phase, the deployment/creation function for the first time or after it has been updated, and the cloud function moves to a second state; pending. During this state, the AWS Lambda prepares the functions environment, which includes loading the code into a new execution context and controlling and setting up required resources in the new configuration. Then it switches to an active state if the deployment is successful or turns to a failed state when some major error occurs. In an active state, the function is ready to respond to the client, and if no further requests come in, the function turns into an inactive state. When an innovation occurs and there is no need to recreate these inactive functions, the system shuts them down. During periods of low demand, the system creates fewer instances to handle the volume of concurrent requests. The simplicity, dynamic scaling, and granular billing[4] of FaaS empowers its rapid adoption in enterprise and scientific applications, making it the first preferred choice for leveraging cloud elasticity and monitoring development and operations life cycles.

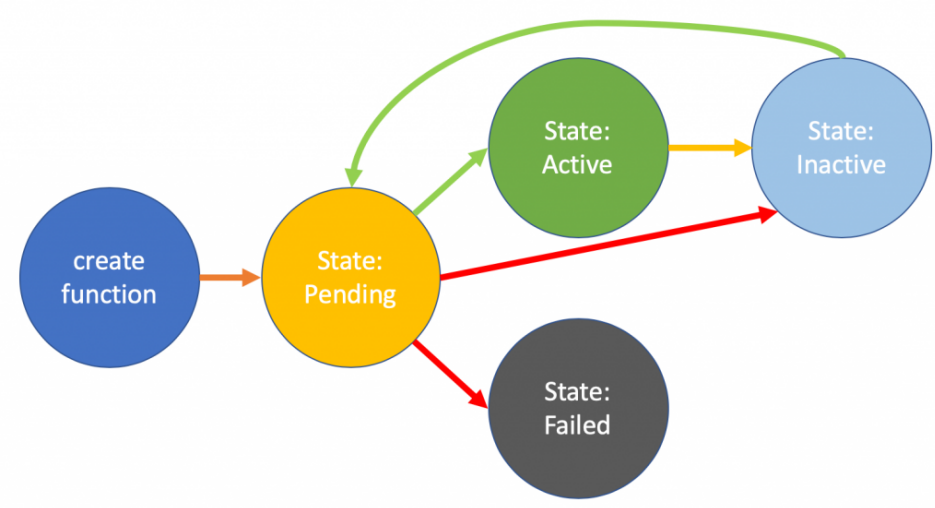

Figure 2: AWS Create Function Cycle [aws19]

[4]Granular billing specifically refers to a billing model that charges users based on the precise quantity of resources they utilize, such as the number of milliseconds of compute time and megabytes of memory consumption.

2.2 Cold Start Durations in FaaS Infrastructure

Cold start durations in FaaS infrastructure describe the latency that occurs when a cloud function is called for the first time or after a period of inactivity. The cloud provider must allocate resources, set up the run time environment, and load the required dependencies before the function can execute, causing extra execution time. FaaS causes an extra cold start duration however, it is still really rational due to its pay-as-you-go pricing system, charging users only for their actual service usage, not for periods of inactivity.

Multiple factors contribute to the delay experienced during a cold start. Initially, the function requires the initialization of a container, which entails the allocation of a fresh container or virtual machine instance to serve as the host for the function. Subsequently, it is necessary to access the function package storage and proceed to duplicate the function image into the container. This step involves getting the code package for the function from storage and transferring it to the container. The computer loads the image function into its memory, unpacks it, and prepares it for execution. Finally, the function executes within the prepared container. Comprehending and controlling these procedures is essential for maximizing the efficiency of cold start durations. This process was detailed in the paper by Parichehr Vahidinia et al. [VFA20]

Even some of the studies were made to mitigate cold start durations. For example, Xuanzhe Liu et al. [LWC+23] outlines how the cold start latency in FaaS applications can be optimized by eliminating irrelevant functions. These unnecessary functions are defined as those for the algorithm that has been not utilizing these codes or dependencies. Therefore, they can be removed without generating any run time failures in the FaaS application. Together with this FaasLight, they were significantly reducing the code loading latency (up to 78.95 percent, 28.78 percent on average), so that decreasing the cold-start latency.

In the case of compiled languages such as Java, various reasons contribute to longer cold start times, even when the code has already been compiled and packed. Upon deployment of a Java function, the cloud provider allocates resources like CPU and memory and establishes a new container or virtual machine instance. The Java Virtual Machine (JVM) needs to follow some initialization stages, which involve loading the JVM itself, initializing heap memory, and setting up the run time environment. The setup procedure is generally more intricate and demanding in terms of resources when compared to interpreted languages.

In addition, the Java Virtual Machine (JVM) must load and verify every class, a process that can be time-consuming, particularly for applications with numerous classes or substantial utilization of reflection. Java applications frequently use frameworks like Spring, which perform thorough dependency injection and configuration at startup, contributing to the overall initialization time. On the other hand, the Just In Time (JIT) compilation method enhances the efficiency of the bytecode at run time, but it also causes extra startup delay during the

initialization phase.

On the other hand, Python as an interpreted language typically has shorter cold start times because its setup operations are simpler. The Python run time has the advantage of faster startup time due to the lack of a separate virtual machine and reduced complexity in loading dependencies. Nevertheless, interpreted languages may exhibit alternative performance compromises during run time, such as comparatively slower execution times than compiled languages.

Different cloud providers implement diverse cloud function infrastructures and policies that influence cold start durations. Each provider determines the duration of idle time before terminating a function, depending on their resource management practices. For instance, AWS Lambda, Google Cloud Functions, and Azure Functions employ different strategies for allocating and managing resources, which might influence the frequency and length of cold starts. Developers must comprehend these distinctions to optimize their applications for particular cloud infrastructures.

These delays can have a substantial influence on the performance and overall satisfaction of the user. Developers can employ different approaches to minimize the delay in starting a function due to a cold start. These approaches involve optimizing the function's code, minimizing dependencies, and utilizing provided concurrency to keep functions ready to handle requests immediately. To ensure rapid response times for their applications, developers can tackle cold start difficulties and customize solutions for individual cloud providers, thus accommodating different workloads and conditions.

2.3 Trigger Types

The goal of this thesis is to evaluate the performance and latency of various serverless function triggers across different platforms. In this section, we described different trigger types that are commonly found in cloud platforms.

Hypertext Transfer Protocol (HTTP): As a synchronous triggering example HTTP is commonly utilized due to its simplicity and direct interaction with web services. Functions can be called through web requests, which makes them well-suited for real-time applications like APIs, and microservices. In this trigger type function executes and promptly returns a response upon receiving a request.

Storage: Storage triggers are crucial for managing events associated with file storage systems, such as file uploads, modifications, or deletions. They facilitate the automation of processes such as image processing, data backup, and content management. As an option for event-based triggering, which exhibits asynchronous behavior. Asynchronous computing refers to the execution of activities that can occur independently of the main program flow, enabling other processes to proceed without waiting for the job to finish. This flexibility

enables the efficient management of storage events.

Database: Database triggers play a vital role in preserving data integrity and automating activities in reaction to database modifications. They can initiate functions when records are created, modified, or removed. This trigger type is capable of supporting asynchronous activities. In addition, our database preferences were No SQL databases which are DynamoDB in AWS and Firestore in Google cloud platforms.

Queue: Queue triggers are essential for constructing decoupled and scalable architectures. They react to Queues that are placed in a Queue queue or topic, facilitating asynchronous communication between various components of an application. This trigger type is particularly crucial for managing high volume and decentralized systems, guaranteeing dependable Queue delivery and processing without the need for fast response.

Although HTTP triggers are simple and appropriate for direct interactions, storage, database, and Queue triggers offer the necessary mechanisms to manage background tasks, data synchronization, and inter-service communication. These mechanisms are crucial for constructing resilient and efficient cloud-based systems.

2.4 TriggerBench Research

Figure 3 represents the basic architecture of TriggerBench [JS22]. TriggerBench was used to compare the performance of AWS and Microsoft Azure in an optimal (warm start) scenario. The benchmark instrument's two main components serve as the primary means of conducting benchmarking experiments: benchmark orchestrator and cloud provider. The benchmark orchestrator is responsible for the creation and termination of function code and deployment scripts. It creates workloads according to the experiment specifications, and it collects and parses trace data from the cloud provider after execution. On the other hand, the cloud providers' side, whether AWS or Microsoft Azure, consists of cloud functions that run the experimental code to provide trigger latency data, which is then stored and analyzed by the benchmark orchestrator.

The general process begins with step 1 where the benchmarking orchestrator deploys cloud functions. To make the experiments reproducible an Infrastructure as a Code[5] software Pulumi [pul24] has been used. Then in step 2, according to the workload profile, the functions with different trigger types were invoked with the k6 [gra24] load generator. The invoker cloud function receives the HTTP requests and then triggers the receiver function using an external cloud service (trigger type). During the experiment, the cloud provider saves a significant amount of trace and execution data. In stages 3 and 4, the benchmark orchestrator retrieves these logs to parse the relevant execution data and analyze metrics.

[5]The ability to supply and support your computer infrastructure using code rather than manual procedures and configurations is known as infrastructure as code.

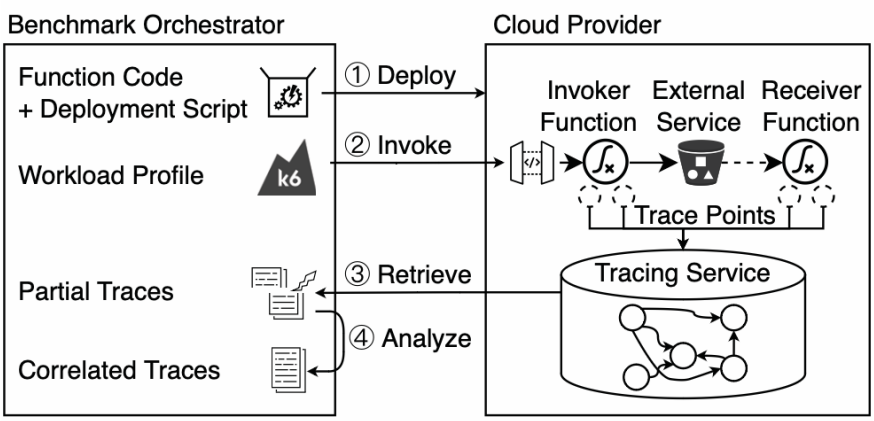

Figure 3: TriggerBench Architecture [JS22]

3 Related Work

In the Related Work section of this thesis, we extensively examine the existing literature on serverless computing. Our analysis specifically concentrates on crucial factors including Cloud Provider, Trigger Type, Programming Language, and the Cold Start and Warm Start situations. The objective of our study is to evaluate the impact of these elements on the architectural capabilities and performance characteristics of serverless operations. This literature search highlights the current deficiencies in serverless computing and shows how we strategically placed our research to successfully solve these gaps.

Table 1 presents an overview of recent literature on FaaS platform benchmarking. As mentioned by Grambow et al. [GPB+21], frameworks only evaluate single aspects of FaaS platforms; a more holistic, application-driven approach is still missing. That is why, within the scope of this study, we focus on benchmarking (execution time, latency, and responsiveness) of serverless functions under different workloads. Also, Table 1 shows that the large majority of studies have focused on the AWS platform and HTTP triggers. Several studies have included experiments on both cold start and warm start; however, these only investigated HTTP triggers. Despite all these research efforts, we notice a critical gap concerning benchmarking the latency of trigger types beyond the standard HTTP trigger. Only two papers, Joel Scheuner [JS22] and Nikhila Somu et al. [NSK20], considered multiple trigger types, but only surveyed the warm start latency by assuming optimal use cases.

Additionally, a thorough examination of serverless computing performance studies reveals that the majority of the research, 53 percent, looks at both cold and warm starts, with 17 percent of the studies focusing exclusively on cold starts and 30 percent on warm starts. The studies that take into account the cold and warm invocation have largely centered on benchmarking HTTP or AWS Step Functions. Motivated by this gap, we aim to assess cold and warm invocation across a broader range of triggers such as HTTP, queue, storage, and database triggers. Our research will highlight the need to understand the latency impact of different triggers on serverless applications, which significantly contributes to the optimization and improvement of serverless computing environments.

We contend that the growing popularity of FaaS will lead to the development of more complex applications and the adoption of more trigger types. It is therefore imperative to have a comprehensive overview of the performance trade-offs of different trigger types on warm start as well as cold start at different cloud providers.

Most of the up-to-date studies around serverless computing are pursuing performance analysis and implementing tests to observe factors contributing to latency. For example, Martins [MAC20] et al. study mainly analyzes the effect of memory allocation, CPU bound cases performance, payload size, and the use of programming language. Martins et al. successfully analyzed these factors for three main providers (AWS, Google, and Azure); however, the

available research did not investigate the impact of serverless triggers on latency, which is the most commonly used factor in every execution.

Referring to scholarly literature examining triggers in serverless computing, Scheuner et al.[JS22] conducted research demonstrating performance benchmarks on serverless function triggers. However, optimized workloads, which omit the cold start duration, constrained their experiments. In addition, Samu et al. [NSK20] argue the present research does not thoroughly investigate the latency effects of serverless chaining and the choice of serverless function triggers. They have tested various trigger variants, but warm starts primarily dominated the shared latency metrics.

In addition to previous benchmarking research, Dmitrii Ustiugov et al. [UAG21] demonstrate a good implementation where they investigate both warm and cold starts in the tail latency; however, in this study, only HTTP requests were considered triggers. Further research is required to examine the effects of serverless triggers on cold start duration and latency.

Aakash Khochare [KKKS23] et al.'s "XFaaS" study, which experimented on AWS Step Functions, emphasizes that the initial delay in serverless tasks is directly connected to the size of the deployment stored on disk. It was discovered that using a smaller package, like the Resize function set at 100 kB, results in a cold start delay of approximately 500 ms. On the other hand, bigger packages such as the ResNet inference function, which is 120 MB in size, encounter considerably longer delays, approximately 2000 ms. This discovery highlights the significance of package size on the duration it takes for initialization in serverless settings which is directly proportional to cold start latency.

Some of the studies that focusing HTTP triggers have also conducted comparisons between various programming languages. For example, Horacio Martins et al. [MAC20] experimented on AWS Lambda functions running in Node.js, Python, Go, and Java. As a result, Java showed consistent performance, with a warm start latency of approximately 55 ms. Among them, Node.js and Java shows the most consistent and stable outcomes. On the other hand, they have performed experiments on IBM Cloud Functions and experienced notable performance problems when using Ruby, with a delay of 790 ms. Similarly, Swift also exhibited subpar and inconsistent performance.

This approach aligns with the findings of Wang et al. [WLZ+18], where they also provided evidence about the cold start latency by discussing how the amount of memory allocated to functions and the choice of programming language has a substantial impact on the delay experienced during AWS cloud function cold starts. According to their study in July–Dec 2017, Python 2.7 has the lowest median cold start latency, ranging from 167 to 171 milliseconds. In contrast, Java functions have much greater latency, ranging from 824 to 974 milliseconds. Furthermore, they discovered that the cold start delay generally decreases as the memory of the function increases.

Related Work	Cloud Provider	Trigger Type	Programming Language	Cold Start or Warm Start
TriggerBench: A Performance Benchmark for Serverless Function Triggers [JS22], 2022	AWS, Azure	AWS: HTTP, Queue, Storage Extra triggers in Azure: Timer, Database, Stream	Python	Warm start
PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications [NSK20], 2020	AWS, Google	HTTP, Storage, Queue	Python3	Warm start
Benchmarking Serverless Computing Platforms [MAC20], 2020	AWS, Azure, Google, IBM	HTTP	Node.js, Go, Java, Python	Warm start
CrossFit: Fine-grained Benchmarking of Serverless Application Performance across Cloud Providers [SDSL22], 2022	AWS, Azure	HTTP then bucket triggering	.NET Core 3.1 and C#	Warm start
Master Thesis: Benchmarking of Serverless Application Performance across Cloud Providers [Den22], 2022	AWS, Azure	HTTP then bucket triggering	.NET Core 3.1 and C#	Warm start
BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms [GPB+21], 2021	AWS, Azure, Google	HTTP	JavaScript	Warm start
Cold Start Influencing Factors in Function as a Service [JMW18], 2018	AWS, Azure	HTTP	Java and JavaScript	Cold start
Cold Start in Serverless Computing: Current Trends and Mitigation Strategies [VFA20], 2020	AWS	HTTP	Python 3.6 and Node.js	Cold start
WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing [SA20], 2020	AWS	HTTP	Python	Cold start
Let's Trace It: Fine-Grained Serverless Benchmarking using Synchronous and Asynchronous Orchestrated Applications[SET+22], 2022	AWS	HTTP	Golang, Java, .NET	Cold and warm start
FaaSLight: General Application-Level Cold-Start Latency Optimization for Function-as-a-Service in Serverless Computing [LWC+23], 2023	AWS	HTTP	Python	Cold and warm start
SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing [CKB+20], 2021	Azure, Google	HTTP	Python, Node.js	Cold and warm start
Analyzing Tail Latency in Serverless Clouds with STeLLAR [UAG21], 2021	AWS, Azure, Google	HTTP	Golang, Python	Cold and warm start
Peeking Behind the Curtains of Serverless Platforms [WLZ+18], 2018	AWS, Azure, Google	HTTP	Python 2.7, Nodejs 4.3.2, Java 8	Cold and warm start
Benchmarking elasticity of FaaS platforms as a foundation for objective-driven design of serverless applications [KWB+20], 2020	AWS, Azure, Google, IBM	HTTP	Python	Cold and warm start
FaaSDOM: A Benchmark Suite for Serverless Computing [MFKS20], 2020	AWS, Azure, Google, IBM	HTTP	Node.js 10, Go 1.11 , Python 3.7	Cold and warm start
XFaaS: Cross-platform Orchestration of FaaS Workflows on Hybrid Clouds [KKKS23], 2023	AWS, Azure	AWS Step Functions: SQS or EventBridge	Python	Cold and warm start
Cross-provider Serverless Trigger Benchmarking, 2024	AWS, Google	HTTP, Queue, Storage, Database	Node.js 20, Python 3.12, Java 21	Cold and warm start

Table 1: Literature Review

4 Methodology

In this chapter, we will explain the design and implementation of our thesis study. The design chapter identifies the experimental requirements and the Implementation chapter explains the infrastructure and components to accomplish our benchmarking experiments.

4.1 Design

During the development stage, TriggerBench's use of Pulumi proved to be a limiting factor, because Accenture enforces tight internal security policies and compliance standards to regulate the utilization of external software products and their access to cloud environments. The required authorization methods for Pulumi, such as Service Account Keys for Google Cloud Platform (GCP) and Access Keys or IAM Roles for Amazon Web Services (AWS), presented considerable difficulties. Pulumi was requiring substantial permissions and possibly wide-ranging access for managing cloud services, which contradicts Accenture's standards regarding limiting security risks and guaranteeing data privacy. We determined that utilizing Pulumi in our cloud operations architecture would be unfeasible without compromising their rigorous security and compliance requirements. TriggerBench [JS22] was primarily suitable for measuring warm starts however, to mitigate these issues and be able to measure cold start durations in long schedules, we developed our benchmarking implementation while adding a new cloud provider, Google Cloud Platform (GCP).

Furthermore, we decided against using a workload generator like k6 [gra24] or Locust [loc] because they were just suitable for testing warm starts in the TriggerBench [JS22] research. Instead, we were compelled to quantify the number of cold start invocations, which prompted us to embrace a sequential triggering mechanism as described in Section 4.2.3. This methodology enables us to quantify both warm and cold invocations within a single scheduled execution over a long period, thereby offering a thorough assessment of function performance across various conditions.

Considering the potential security breaches and significant risks associated with granting extensive permissions in software was putting Accenture projects in jeopardy. This practice is also generally not permitted in most corporate settings. Consequently, we developed our implementation to address these concerns.

The three main experimental criteria that we expect to meet to evaluate the efficiency of serverless functions under different circumstances are:

Latency Measurement: The necessity to accurately measure the time it takes for functions to start and respond when they are activated using various trigger types, including both warm starts and cold starts. With this measurement, we can identify trigger timings for low latency scenarios and we decided to conduct experiments on AWS and GCP, in line with the

resources provided by Accenture.

Runtime Variability: We want to be able to test serverless functions written in the latest run times by integrating a variety of programming languages frequently employed in cloud environments, such as Node.js 20, Python 3.12, and Java 21. Then we can evaluate the impact of language selection on function performance.

Framework Utilization: We expect our experiments to assess the performance impact of package, dependency, and framework utilization by utilizing the SDK offered by GCP and AWS.

Additionally, we believe that although this Infrastructure as a Code component improves the ability to replicate experiments and helps to easily deploy architecture initially, it was just applicable while measuring warm start experiments. Controversially, in our written benchmarking orchestrator, we do not need to create resources and terminate cloud functions as soon as possible like in TriggerBench [JS22] warm start experiments; instead, our research focuses on conducting tests over extended periods which was a requirement for measuring cold start latency and with this approach we were able to adapt framework utilization to observe the latency effect of larger deployment sizes.

4.1.1 Cross-provider Serverless Trigger Benchmarking

In contrast to the research conducted by Joel Scheuner et al. [JS22] in their study cited as TriggerBench, our experimental configuration has been specifically designed to evaluate benchmarking more realistically. This entails using and assessing the performance impact of the official AWS and Google Cloud Platform JavaScript frameworks, testing this benchmarking methodology in variant trigger types, and adding the examination of database triggering. To accomplish our goal all of the triggering designs have been written from scratch and relative cloud frameworks described in Section 5.1.2 have been used to have a more realistic approach. We warrant the use of these widely accepted frameworks because they embody standard procedures for building cloud services, thereby enhancing the relevance and applicability of serverless functions. Our approach entitles the influence of dependency, frameworks in serverless cold start duration which is critical for adding a new perspective to the benchmark while providing valid analysis on dependency usage decisions.

In more detail, TriggerBench [JS22] was not using aws-sdk [awsb] and @google-cloud/logging [goob] frameworks in their trigger functions. We have been applying this framework to compare the impact of installing and running these frameworks during the deployment of cloud functions expected to cause longer cold start durations. We also validated our output with TriggerBench research by conducting run-time HTTP experiments without these frameworks, testing the differences between cold and warm starts.

In conclusion, we have written our benchmarking infrastructure by incorporating Google

Cloud platforms, allowing for a more comprehensive investigation of cold start duration in Function-as-a-service (FaaS) environments. After implementing and deploying the cloud functions architecture, we deployed the infrastructure module to directly accommodate receiver function invocations. This modular approach helps us to address a variety of test scenarios, including multiple trigger types, run times, and cold/warm invocation types. This improvement not only expands the range of our experimental possibilities but also establishes the foundation for a more thorough comprehension of cloud function performance. Our methodology adds a novel approach to evaluating serverless performance by: First, accessing the serverless triggers in cold starts. Secondly, we emphasize the substantial influence of package management on cold start durations. By examining how different dependency and library configurations affect startup times, we offer insights into optimizing these elements to reduce latency. Our evaluation covers not just cold starts, but also run time performance, including the impact of different cloud providers, trigger kinds, and frameworks. With this holistic approach, we aim to enhance the understanding of serverless architecture's dynamics, resulting in more efficient and responsive serverless apps.

4.2 Implementation

To measure cold/warm start durations in research we have developed three main components:

Firstly, we have successfully deployed and implemented receiver serverless functions detailed in Section 4.2.1 with a variety of trigger types in AWS and GCP.

Secondly, we have established a benchmarking orchestrator that creates workloads and collects benchmarking traces from AWS CloudWatch[6] logs or Google Cloud Logging API [goob]. The orchestrator initiates multiple HTTP requests containing query parameters like trigger type and run time, which consequently activate warm and cold invocations of cloud functions. Following each scheduled task we gather and store the NDJSON [7] logs locally.

Thirdly, we developed the Infra module, which receives HTTP requests from the benchmarking orchestrator and triggers receiver cloud functions according to predefined query parameters. We verified that all essential configurations were established and appropriate authorizations were granted to the Infra module, enabling flawless triggering of receiver functions and writing logs via Google Logging API or AWS CloudWatch.

By strategically aligning our infrastructure across AWS and GCP, we can thoroughly and precisely evaluate the capabilities of both cloud providers. This also enables an easier solution to run experiments at the same time. Adopting this dual-provider approach, improved our benchmarking scope, and yielded valuable insights from comparing a wider range of data. The implementation of the algorithm, is available for review and reuse at repository[8].

4.2.1 Cloud Functions

To conduct our experiments, multiple receiver cloud functions have been developed by utilizing different trigger technologies. The separately written JavaScript (Nodejs 20) receiver functions that are specifically customized for cloud provider requirements are shown in Table 2 and they have been used in Experiments 2 and 4. However, while implementing Experiment 4 the Infra module has not been used, and the cloud functions have been redesigned to consecutively trigger each other to represent a chain of function triggering as described in Section 4.2.5. While deploying AWS receiver functions the serverless framework [ser14] has helped adjust authentications and have faster deployment cycles. As the serverless framework does not support GCP the authentication and deployment have been done through Google SDK comments.

Furthermore, as presented in Table 3 the Experiment 3 cloud functions have been written in different programming languages. Specifically, we utilized two prevalent cloud functions

[6] Amazon CloudWatch is a service that oversees applications, reacts to fluctuations in performance, enhances resource use, and offers valuable information on operational well-being.

[7] Newline-delimited JSON (NDJSON) is a data format that uses lines as separators to specify the structure of JSON data.

[8] https://github.com/dervisonurgurbuz/cross-provider-serverless-trigger-benchmarker.git

Trigger Types	Supported Cloud Provider and Technologies	Programming Language
HTTP	AWS and Google API Gateway	JavaScript
Queue	AWS (SNS) and Google (Pub/Sub)	JavaScript
Storage	AWS (S3 Bucket) and Google (Bucket)	JavaScript
Database	AWS (DynamoDB) and Google (Firebase [Fir24])	JavaScript

Table 2: Cloud Functions in Experiment 2 and 4

within the Java run time. For AWS we have applied a serverless framework and Jackson et al.'s example [awsa] which occupies 2.3 MB on disk. Additionally, for GCP, Google has provided official documentation and optimal software [goo23] requiring 25 KB of disk space.

Trigger Types	Supported Cloud Provider	Programming Language
HTTP	AWS And Google	JavaScript
HTTP	AWS And Google	Python
HTTP	AWS And Google	Java

Table 3: Cloud Functions in Experiment 3

4.2.2 Architecture

Figure 4 illustrates the main infrastructure used to benchmark cloud function triggers. The process begins with a benchmarking orchestrator. Depending on the setup for Experiment 2 or 3, multiple unique HTTP requests have been sent to the Infra module. Each request initiates a different Infra process, and according to query parameters specifying trigger type (HTTP, Message, Storage, or Database) or run time (Python, Java, or JavaScript), the relevant receiver function is activated. For instance, in Experiment 2, when the trigger type is storage the Infra module uploads a file in a specific bucket and as soon as a storage event occurs the storage trigger has been activated with a cold/warm invocation latency. Another example from Experiment 3, might be a run-time parameter set to Python, which would send an HTTP request to a receiver function written in that programming language. While triggering receiver functions, critical parameters such as traceId and invocation type are also transmitted to track each execution. Serverless Benchmarking Architecture has been developed separately for GCP and AWS, as the authentication, logging service, and the used frameworks are different.

Cloud provider command line interfaces have deployed the functions to build this interconnected infrastructure. For AWS deployment, Serverless framework [ser14] [ser] has been used to efficiently deploy FaaS services with relevant authorization. On the other hand, Google lacks this framework, prompting us to employ Google SDK [gooa] and libraries have been used in the deployment stage. Additionally, while writing the GCP Infra module we have utilized Firebase Admin SDK [Fir24] for secure connection and edit data to trigger

database events.

Unlike the TriggerBench[JS22] infrastructure, which used AWS X-Ray [aws24] for tracing logs, our work utilizes AWS CloudWatch, corresponding to the Google Logging API [goob], for our logging needs. With this approach, we were able to include the performance impact of frequently used cloud provider logging frameworks. This implementation also requires us to write our parser which has been detailed in Section 4.2.7. Additionally, as their implementation has been more focused on warm start they have written a benchmark orchestrator to run in cloud servers. However, our implementation is expected to take longer periods, making it more practical to use a local machine to schedule these in long experiments. In our architecture, Execution Logs have been created as objects and written into cloud provider log services. According to a cloud provider, AWS Cloud Watch or Google Logging API has been used to record each execution step per invocation, enabling the cloud provider to save logs within unique log groups for further installation, processing, and data analysis.

As presented in Figure 4, the benchmarking experiment starts with a request from a benchmark orchestrator. In stage 1, the request is sent to the Infra cloud function. Then, in stage 2, the Infra function invokes the next cloud function according to the trigger type. Stage 2 involves activating the function trigger type and finalizing the computation. During this process, stages 3.1 and 3.2 have been active for collecting all of the cloud function execution logs to write into AWS/Google logging services. In stage 3.1, Cloud Function 1 records execution logs, which include information about trace ID and the trigger invoke time. The invoke time refers to the Epoch[9] time when the Infra module creates an event to activate Cloud Function 2, which we define as the receiver function. As soon as the receiver function enters to active stage and begins compiling, the trigger start time is logged with the same trace ID in stage 3.2. Then finally in stage 4, the benchmarking orchestrator gathers this execution data to parse and further analyze the performance metrics after completing the executions and finalizing the computations. After parsing the invocation time and trigger start time for each trace, these values are subtracted to analyze trigger latency.

[9] An Epoch time is a specific point in time that serves as a reference for determining a computer's clock and timestamp values.

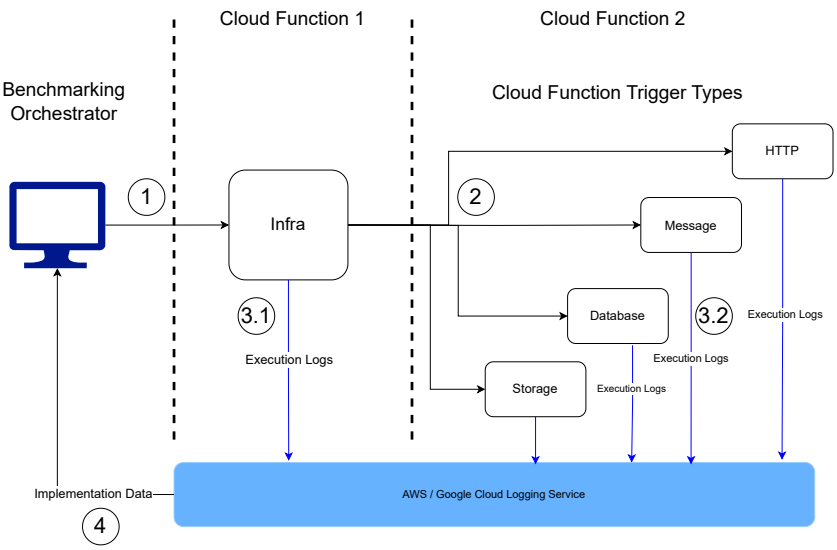

Figure 4: Serverless Benchmarking Architecture

Both Google and AWS treat each FaaS invocation individually, which makes it challenging to relate the trace flow of a single chain of trigger actions. For this reason, we have assigned a unique trace ID to FaaS functions from the outset, and use this trace ID at each logging step to identify the relevant execution and capture cloud function trigger execution traces. In contrast, while reproducing the TriggerBench [JS22] research, we adopted their approach; the trace ID has been created by an open-source load testing tool k6 [gra24] and adopted to their invoker function.

4.2.3 Sequential Triggering

To be able to measure cold start latency without the need to destroy cloud functions, we have decided to apply a sequential triggering design in Figure 5. In this scheduled implementation the benchmarking orchestrator creates workloads within a time interval between invocations, which has been inspired by Manner et al. [JMW18], who used 30-minute intervals between invocations to trigger cloud functions in the cold state and compared it with the warm start. During Experiments 2, 3, and 4, the Infra module was triggered at 30-minute intervals, with two invocations in each interval. The Infra module activates the corresponding receiver function, where the first invocation is a cold start and the second invocation is a warm start occurring after 2 seconds. The cold start occurs during the first invocation, and happens when no function is idle state and the code needs to run. This is due to the cloud provider's setup in FaaS infrastructure, which destroys functions not in use. Executing the function after a long period requires loading its scripts into the container and deploying the related dependencies, typically leading to a longer initialization duration known as the cold start duration. With a 30-minute interval, we make sure that sufficient time has passed for the

cloud provider to destroy the receiver function, allowing us to observe cold start durations while recreating the cloud function.

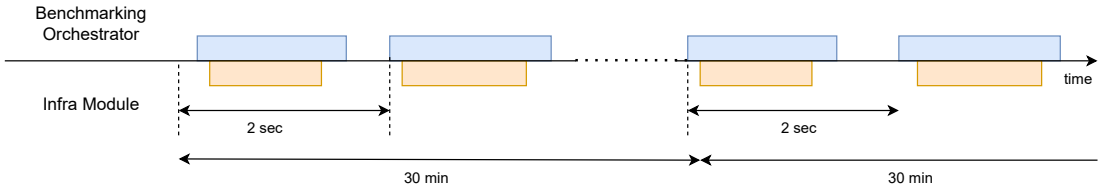

Figure 5: Sequential Triggering [JMW18]

The benchmarking orchestrator uses the node-schedule [nod23] library to set up a scheduled job that executes ever 30 minutes. In every job, the sendRequest function has been called four times specifically targeting each one of the serverless triggers; HTTP, database, storage, and queue. For each trigger type, a request is initially sent to the Infra module, labeled as a cold invocation. Then the algorithm waits 2 seconds before invoking these trigger types again, simulating warm starts. Additionally, all of these requests have been sent to the Infra module of the corresponding cloud provider (AWS/GCP). Also, benchmarking orchestrators were written separately to enable simultaneous experimentation. Finally, the job waits an additional 3 minutes to finish operations before executing, the readLogs function which installs, saves, and processes the Execution Logs on the local device. The outcomes, as well as any observed mistakes while reading the log, are recorded on the console. The purpose of this organized timing and request sequence is to methodically assess and evaluate the effects of cold starts and warm invocations on performance in cloud settings.

4.2.4 Runtime

We have developed our Infra module to respond to various test scenarios, such as run-time specifications and trigger types. We have deployed new HTTP trigger functions using different run times: JavaScript, Python, and Java. To validate our experimentation with the TriggerBench JavaScript warm start invocations we have used neither AWS nor GCP frameworks in this experimental setup. We successfully triggered functions with different run times within a scheduler workload profile and saved logs to parse and plot graphs. The functions have been tested with sequential triggering described in the previous Section 4.2.3 where the first invocation is cold and the second in warm start. Then the responses of these HTTP requests were logged and the response data was filled with the triggerStartTime to calculate cold and warm start latencies.

4.2.5 Chain Of Cloud Functions Triggers

In this implementation, multiple cloud functions across four triggered types have been triggered like a chain of reaction. This experimental setup has been designed to illustrate the effect of trigger latency in a bigger architectural scope constructed with numerous microservices working together to maintain a big software application. As there is no need for dynamically triggering different types of receiver functions and the same types of triggers have been consecutively activating each other, we have authenticated and modified each trigger function to initiate the event that triggers the next cloud function.

Figure 6 represents the architecture of this experimental setup; all of the implementations have been deployed for this use case by two major cloud providers. At stage one, the server/local benchmarking orchestrator invokes HTTP by the workload profile. Then, in stage 2, the cloud function invokes a Queue trigger, which initializes a chain of triggers that follows with storage and database triggers. In stage 3, the AWS or Google Cloud Functions logging services save the infrastructure logs during this execution under a unique log group name. Next, we extracted the implementation data from the log groups for additional data parsing, processing, and plotting. Finally, the local device or server parses this execution data saved in logs.ndjson sharing the identical trace IDs.

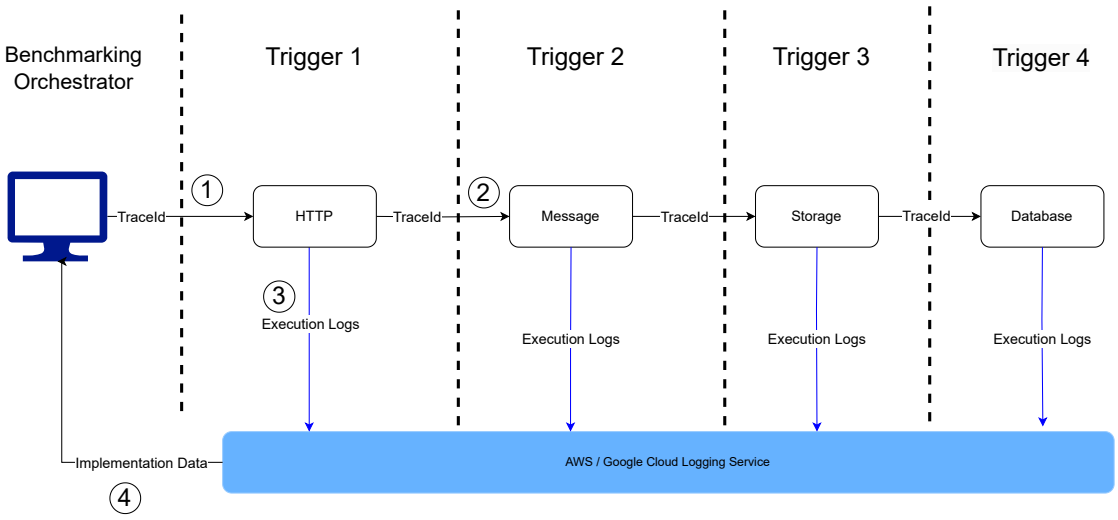

Figure 6: Chain of Functions Architecture

4.2.6 Trace Collector

During the experimental setup in Figure 4, after the execution the algorithm starts the trace collection phase, which then pulls relevant logs and saves them by appending logs.ndjson file. To efficiently parse these data the NDJSON format has been applied.

While collecting traces from AWS, like in Listing 2, the Cloud Watch service was ben-

eficial in storing logs within the same trace groups, enabling further querying and retrieval of relevant logs. The needed traces and invocation timings were pushed under the name of Execution Logs to AWS Cloud Watch. Additionally, GCP offers a logging service that allows for the retrieval of logs with API requests following the completion of executions which has been shown in Listing 3. In our implementation, both AWS and GCP generate execution logs independently, leading to the logging and receiving of information in different formats. This separation requires plotting algorithms and distinct parsers to process individual data formats as will be described in Section 4.2.7.

In contrast, in TriggerBench [JS22] implementation utilized k6 for workload generation and creating trace ID. These trace IDs were then adopted in AWS X-Ray [aws24] tracing headers, which allows the finding correlation of trace points from the same request between the invoking and receiving functions. Additionally, they used parent ID to identify the origin of the service call by connecting it to the trace point that initiated the call. As the k6 [gra24] open source load testing tool is suitable to measure warm starts but we aim to conduct studies that measure both warm and cold starts, we have developed our infrastructure to deliver these needs.

4.2.7 Parser

As part of our benchmarking architecture, we have written an essential Python script, where the algorithm parses data and creates critical benchmarking graphs. We extract important measures, such as TraceId, infraInvokeTime, triggerStartTime, invocation and run time, from NDJSON log files request.ndjson shown in Listing 1 and logs.ndjson shown in Listing 2. The trace ID serves as a unique identification that allows for the correlation of logs from the same invocation, facilitating a comprehensive inquiry. We meticulously examine this data and store it in dictionaries. Next, we calculate the differences in latency between the time that the infrastructure is invoked (infraInvokeTime) and the actual start of the trigger (triggerStartTime). The latencies are classified according to whether the invocation was 'cold' or 'warm', enabling us to evaluate the effectiveness of the serverless system.

After organizing the data into a pandas DataFrame, we proceed to compute the Empirical Cumulative Distribution Function[10] (ECDF). We have used Matplotlib[mat24] to generate our graphs and add vertical lines to emphasize the median latencies, which aids in the identification of typical performance bottlenecks. Furthermore, we utilize Seaborn[sea24] to create box plots that graphically compare latencies among various run times (Java, Python, JavaScript) and types of invocations. The charts, which are distinguished by unique designs and colors, are saved in PDF format for easy and thorough examination. By employing a

[10] An empirical distribution function in statistics refers to the distribution function that is linked to the empirical measure of a sample.

thorough technique, we can fully examine and visually represent the latency characteristics and performance indicators of cloud functions. This analysis investigates impact of several aspects, such as trigger type, run time, and invocation type, on trigger latencies.

The distinctions in data parsing between AWS and Google Cloud logs are determined by factors such as the structure and storage formats, as well as the certain logging services employed: AWS CloudWatch and Google Cloud Logging API. AWS logs, that have been accessed using CloudWatch, are structured with important metrics like 'TraceId' and 'triggerStartTime' which have been saved in NDJSON format. This streamlines the parsing process, enabling the direct correlation of these attributes to Python dictionaries without the need for considerable text manipulation. The organized structure of CloudWatch logs enables easy extraction and visualization of data. On the other hand, when using the Google Cloud Logging API, the retrieved data has been saved in NDJSON format, and important information is frequently included in the 'textPayload' of Google Cloud logs. This requires intricate manipulation of strings to extract data like 'traceId', 'start times', and 'trigger types'. This necessitates the script to analyze and divide text according to precise patterns, hence rendering the parsing process more complex. In addition, the Google Cloud strategy frequently requires managing several data sources and consolidating data based on 'traceId', which introduces challenges in guaranteeing data consistency and comprehensiveness before analysis.

During Experiments 2, 3, and 4 the request.ndjson data was directly appended by the benchmarking orchestrator with data associated with the invocation time and response of the HTTP requests. Initially to conduct Experiment 2, both logs.ndjson, containing cloud service logs, and request.ndjson were parsed. However, in our run time experiment, we noticed that analyzing synchronous functions could be efficiently conducted by scanning the single data file request.ndjson, as the cloud functions have been configured to return trigger start times. This configuration enabled us to easily extract performance metrics, such as the duration of invocations and the start times of triggers, directly from request.ndjson. Nevertheless, the logging procedure for asynchronous (event-based) functions is intrinsically more complex because of their non-blocking execution nature. In such instances, the initial invocation data and execution outcomes are logged independently within the same log group provided by the cloud service. These logs are retrieved when the execution is completed. This configuration requires the utilization of a shared 'TraceId' to establish a connection between the initiation and completion events in various log entries. As a result, specific information about the execution process is kept in a separate log file called logs.ndjson. This log file captures data once every execution is finalized.

As mentioned in the previous paragraph, in Experiment 2, two log files —request.ndjson and logs.ndjson— were parsed by our algorithm, which compares and combines the data from both files using the traceId. The objective of this comparison and merging process is to verify the precise execution inside a specific time frame, validating 'TraceId's in that

execution from ‘request log.ndjson‘ to categorize the type of invocation as either warm or cold. This approach guarantees that we obtain an accurate picture of the entire function life cycle, encompassing both the start and end metrics. This is vital to precisely evaluate the performance attributes and behavior of asynchronous serverless functions.

Example of request.ndjson Listing 1, presents the log produced from the benchmarking orchestrator with the response from the HTTP request sent to the Infra module. The ”infraInvokeTime” represents the time at which the cloud function has been triggered by Infra. Additionally, the invocation is the first invocation in that schedule which is why the invocation has been labeled as cold.


```
{"triggerType":"http","TraceId":"35d59156-b328-455e-a85f-
   c42113d5d575","invocation":"cold","requestTime
   ":1718782593974,"requestTimestring":"2024-06-19T07
   :36:33.974Z","responseTime":"2024-06-19T07:36:36.658Z","
   requestDelta":2684,"status":200,"data":{"Queue":"Go
   Serverless v4! Your function executed successfully!","
   traceIdHttp":"35d59156-b328-455e-a85f-c42113d5d575","
   currentTime":"2024-06-19T07:36:36.664Z","infraInvokeTime
   ":1718782595479}}
```


Listing 1: request.ndjson

Listing 2 shows the logs pulled form the AWS Cloud Watch. In this example, the event type is HTTP which means the trigger type. Also, we can recognize the ”TraceId” is the same. The parser uses these ”TraceId” to capture invocation traces from separate files and calculate latency in cold or warm invocations.


```
{"eventType":"HTTP","method":"GET","path":"/","sourceIp
   ":"35.181.5.178","userAgent":"axios/1.7.2","
   queryStringParameters":{"traceId":"35d59156-b328-455e-a85f-
   c42113d5d575","trigger":"http"},"triggerStartTime
   ":1718782596483,"TraceId":"35d59156-b328-455e-a85f-
   c42113d5d575"}
```


Listing 2: AWS Cloud First HTTP Log Example - logs.ndjson

The last listing 3 represents the logs received from Google Cloud Log Service API and the important information used in our processes has been highlighted.


```
{"timestamp":{"seconds":1721147404,"nanos":638000011},"
   labels":{},"insertId":".........5WPDIT6HOE7PnXYZQVjlMl
```

```
","httpRequest":null,"resource":{"labels":{"function_name
":"nodejs-http-function","project_id":"deep-beanbag
-395510","region":"europe-west3"},"type":"cloud_function
"},"severity":"INFO","logName":"projects/deep-beanbag
-395510/logs/my-trace-log","operation":null,"trace":"
projects/deep-beanbag-395510/traces/473c06de-abf7-49cc-86
df-e3b1a9801fa5","sourceLocation":null,"receiveTimestamp
":{"seconds":"1721147404","nanos":920791743},"spanId
":"","traceSampled":false,"split":null,"textPayload":"
CORSENABLEDFUNCTION, start: 1721147404495, end
:1721147404501, execution: 6, TraceId: 473c06de-abf7-49cc
-86df-e3b1a9801fa5","payload":"textPayload"}
```


Listing 3: Google Cloud Logging HTTP Log Example - logs.ndjson

We have assigned unique traceId to cloud functions and used these identifications during all trace logging to address this issue. These examples have been given to illustrate the scope of data parsing functionalities in our implementation. These fine-grained parsing tools have been constructed to work with the 4 trigger types: database, storage, HTTP, and queue. Finally, we have parsed and saved the execution metrics with the same traceId as data frames to calculate the trigger latency and implemented data analysis accordingly.

For the chain of functions, the received NDJSON logs have been used to extract timestamps and trace IDs, initiating the organization of the records. Then a dictionary has been created to systematically add data values based on trace IDs. Latency was calculated by subtracting the 'triggerStartTime' from the 'invokeTimes'. All of the acquired data was saved by appending a data frame, and the results were stored in a CSV file in table format.

5 Experiments

In this section, we explore various experiments designed to assess and contrast the performance of serverless computing across diverse platforms and circumstances. Our experiments are structured around specific setups and the architectures described in Section 4.2 to provide comprehensive insights into the performance of serverless functions. Node.js is our intended primary scripting language because of its popularity among cloud providers and its suitability for high throughput, low latency workloads. Also in experiments with different run time environments, Python and Java have been utilized to assess their performance in GCP and AWS.

5.1 Experimental Setup

We developed our experimental setup in the eu-west-3 data centers of both AWS and GCP platforms. In Experiment 1 we replicated the TriggerBench research [JS22] in AWS to have a broader understanding of serverless benchmarking and compare AWS performance with 2022 and May 2024 results.

5.1.1 Cloud Providers

The rest of our experiments 2,3 and 4 conduct a comparative analysis of two prominent cloud service providers, namely AWS and GCP. We have developed the required benchmarking orchestrators and cloud functions on AWS and Google Cloud Platform, expanding upon the current progress. Concurrently, we have created a benchmarking orchestrator, cloud functions, and infrastructure modules on both Amazon Web Services (AWS) and Google Cloud Platform (GCP) which has been conducted in experiments 2, 3, and 4. Additionally, our study has applied enhancements such as database triggering, the usage of a new cloud platform (GCP), and a scheduled dual invocation service. Given that previous research in TriggerBench focused primarily on benchmarking warm invocations, we found it necessary to redesign the entire structure on AWS to incorporate sequential triggering 4.2.3. This integration enables us to measure both cold and warm starts within a single schedule while facilitating a fair and valid comparison. Our implementation also leverages the fundamental cloud frameworks mentioned in section 5.1.2, which is why we have rewritten the architectural design to ensure consistency in both cloud platforms while using their SDK frameworks and asses the latency impact of these frameworks in cold start duration.

Furthermore, instead of reserving extra cloud resources for benchmarking orchestrators and deployment of the cloud functions we have used our local device to reduce cost and can use it in weekly long experimental cycles. Additionally, the benchmarking orchestrator created a workload according to the specifications from Section 4.2.3 while storing the

”Execution Logs” locally.

5.1.2 Frameworks

The frameworks have been added to our experimental setup to assess the impact of different frameworks and dependencies on serverless warm/cold invocations by executing serious controlled experiments in AWS and GCP. We expect larger initialization durations and longer cold start durations as the serverless infrastructure package size increases. Our implementation aimed to demonstrate the influence of these frameworks on the cold start durations of serverless functions.

To investigate the effect of various frameworks and SDKs listed in Table 4 on cold/warm start latencies, in Experiments 3 and 4 we used these frameworks, whereas we did not apply them in Experiments 1 and 2. This approach allowed us to assess the influence of these frequently used frameworks on cold/warm invocations.

	serverless framework [ser14]	**aws-sdk [awsb]**	**@google-cloud/functions-framework [gooa]**	**@google-cloud/logging [goob]**
Experiment 1	No	No	No	No
Experiment 2	Yes	Yes	Yes	Yes
Experiment 3	No	No	No	No
Experiment 4	Yes	Yes	Yes	Yes

Table 4: Framework Utilization Per Experiment

5.1.3 Programming Language

We have also added the programming language component in the Infra module to retrieve the run-time parameters via HTTP Query because we wanted the bench marker orchestrator to autonomously create workloads with a single HTTP Request. During the run time experiments, the HTTP triggers programmed in different languages were tested for both warm and cold invocations. This Infra module was created to invoke cloud functions written in JavaScript (Nodejs20), Python (Python3.12), and Java (Java 21).

5.2 Experiment 1: Reproducing TriggerBench Warm Start Research in AWS

To establish a strong foundation and gain insights into the latest AWS warm start performance metrics in Node.js (nodejs18.x), we have started our experiments with reproducing TriggerBench [JS22]. With the technological advancements, AWS modifies their infrastructure which is why the Node 12 was not supported and we were able to compare the improved version of Node 18. We have reproduced the AWS TriggerBench research where they worked with three trigger types, such as HTTP, queue, and storage.

On the AWS platform, Figures 7 and 8 depict the Empirical Cumulative Distribution Function (ECDF) of trigger latencies for several trigger types (HTTP, Queue, and Storage). Both graphs demonstrate comparable patterns, with HTTP triggers consistently displaying the lowest delay, followed by queue triggers, and then storage triggers, which exhibit the highest latency.

Figure 7 displays the original research findings of TriggerBench [JS22], indicating that HTTP triggers have a median trigger latency of roughly 41 milliseconds, queue triggers have a latency of around 111 milliseconds, and storage triggers have a substantially higher latency of 1345 milliseconds. The replicated outcomes shown in Figure 8 exhibit marginal variations in latencies: HTTP triggers have a median latency of 37 milliseconds, queue triggers at 108 milliseconds, and storage triggers at 1133 milliseconds. Although the overall performance of the trigger types is similar in both figures, there are minor differences in the precise latencies.

Disparities in the experimental configuration, provider management influences, or minor alterations in the execution could account for these changes when replicating the findings. Both graphs demonstrate that HTTP triggers have the fastest response times.

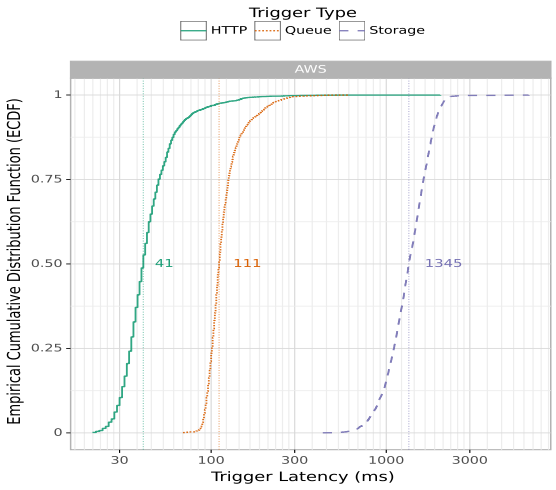

Figure 7: TriggerBench Research Results [JS22]

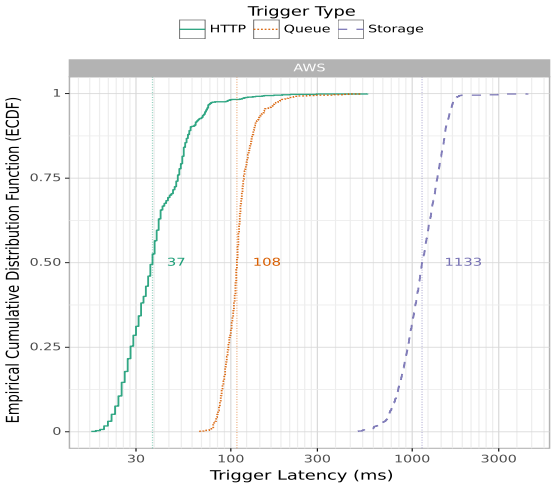

Figure 8: Reproduced TriggerBench Research Results

5.3 Experiment 2: Comparing AWS and Google Cloud Platform

In this experiment, we evaluate the efficiency of variant serverless triggers in JavaScript run times and assess the difference in warm/cold invocation on both AWS and Google Cloud Platform (GCP) while utilizing cloud provider frameworks which are discussed in Section 5.1.2. Additionally, we have applied the Empirical Cumulative Distribution Function (ECDF) to accurately assess which platform provides better performance for JavaScript serverless functions in terms of both latency and dependability. In both cloud platforms warm/cold starts were captured based on data produced from sequential triggering detailed in Section 4.2.3 of the four trigger types.

5.3.1 Results for Warm Start

After conducting our warm start Experiments 1 and 2, we have created Table 5 to validate our results with the existing TriggerBench [JS22] research. In summary, the reproduced TriggerBench values correspond to the original. Additionally, as we expected the the utilized frameworks have created an extra latency in serverless warm start durations. However, the ”Cross-provider Serverless Benchmarker” results with - HTTP 105 ms, queue 151 ms, and storage 1541 ms, as detailed in Table 5 - once again demonstrate that our experimental results in Figure 9 aligns with TriggerBench study.

Source	**HTTP**	**Queue**	**Storage**
TriggerBench Results	41	111	1345
Reproduced TriggerBench Results	37	108	1133
Cross-provider Serverless Benchmarker	105	151	1541

Table 5: Validation of Warm Start Median Results in Milliseconds.

When we looked at the AWS Empirical Cumulative Distribution (ECDF) plot for trigger latencies based on trigger type during warm invocations, as shown in Figure 9, we saw clear and unique performance metrics. HTTP and queue triggers show better performance, such as median latencies of 105 ms in HTTP and 151 ms in queue. On the other hand, storage triggers had the largest median delay at 1541 ms, indicating a considerably high latency. The median delay for database triggers is 650 ms, which is decent performance for the architectures requiring use. The initial examination of our bench marker highlights the effectiveness of HTTP and queue triggers and the delay factor associated with storage triggers.

The same experimentation was conducted in parallel on the Google Cloud Platform and the majority of warm invocations took less than 2000 ms. Figure 10 illustrates that HTTP triggers have a median latency of 428 ms, making them the fastest option within GCP but still larger than AWS. Google HTTP results once showed its suitability for jobs that require low latency compared to other trigger types, such as queue and storage. Database triggers

have a median delay of 1282 ms and queue triggers have a median delay of 1211 ms, making them slower trigger types. Storage triggers have a median latency of 1588 ms, making them the slowest and showing the largest substantial range in activation times. By comprehending these variations in latency, we can optimize the design of Google Cloud based systems.

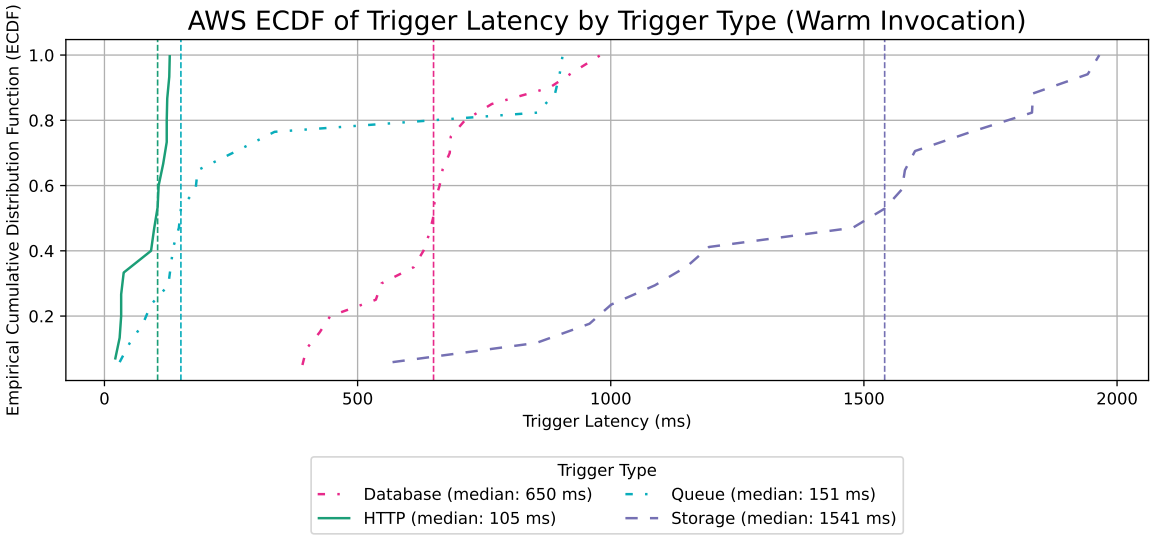

Figure 9: AWS Warm Start

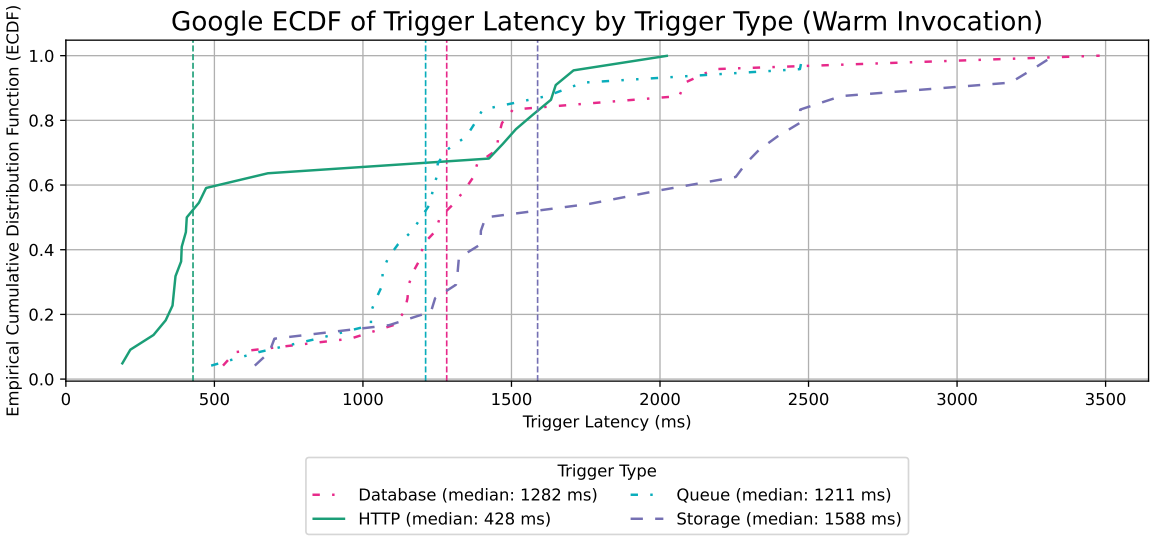

Figure 10: Google Warm Start

5.3.2 Results for Cold Start

Focusing on cold starts, we have investigated the JavaScript run times on AWS and GCP. Cold starts are critical for understanding the initial delay and resource allocation inefficiencies that may occur. We set up experiments to measure these aspects and hypothesize outcomes based on theoretical knowledge and prior empirical studies measuring cold start durations like Manner et al. [JMW18]

In AWS cold start invocations at Figure 11, the HTTP trigger has the lowest median latency of 1054 ms. The steepness of ECDF curves for both HTTP and queue triggers indicates that they provide similar outputs, with a queue median latency of 1096 ms. Additionally, database and storage triggers have median latencies of 1576 ms and 2151 ms, respectively. We believe that a wider range of trigger latency results in storage and database triggers is the indication of their more complex initialization processes and heavier dependencies.

Figure 12 with a median delay of 1640 milliseconds, HTTP triggers are the fastest at Google. Queue triggers and database triggers come in second and third, respectively, at 2522 and 2630 milliseconds. Based on cold start findings, it appears that the majority of HTTP invocations 80 percent do less than 2100 milliseconds. Storage triggers median delay, on the other hand, is the greatest at 2760 milliseconds, providing slower performance. A wider range of latencies is displayed by the more gradual ECDF curve for Storage, where about 80 percent of invocations take values between the range of 2600 and 3500 milliseconds.

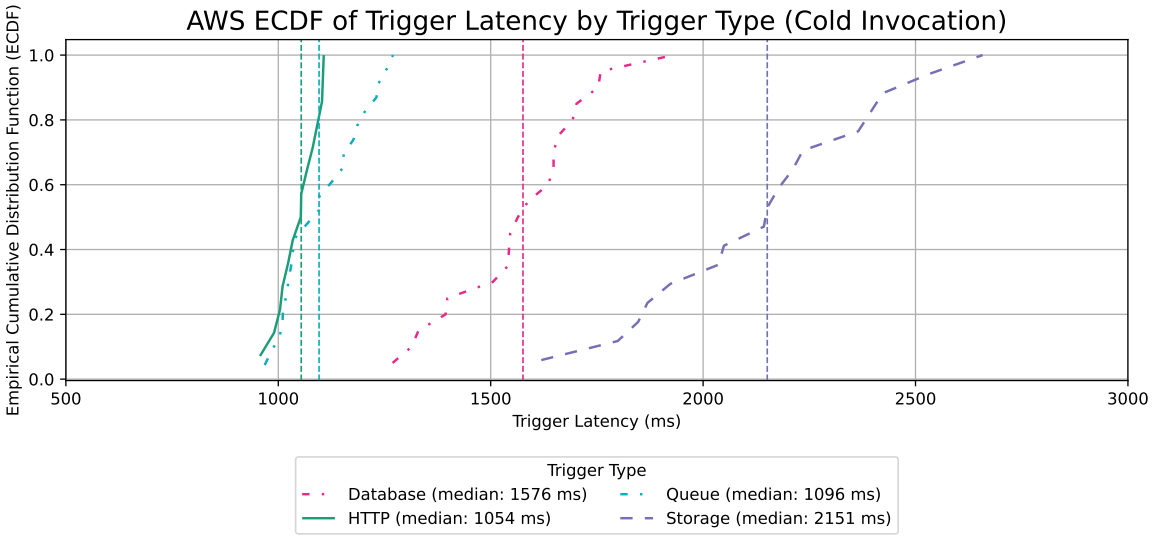

Figure 11: AWS Cold Start

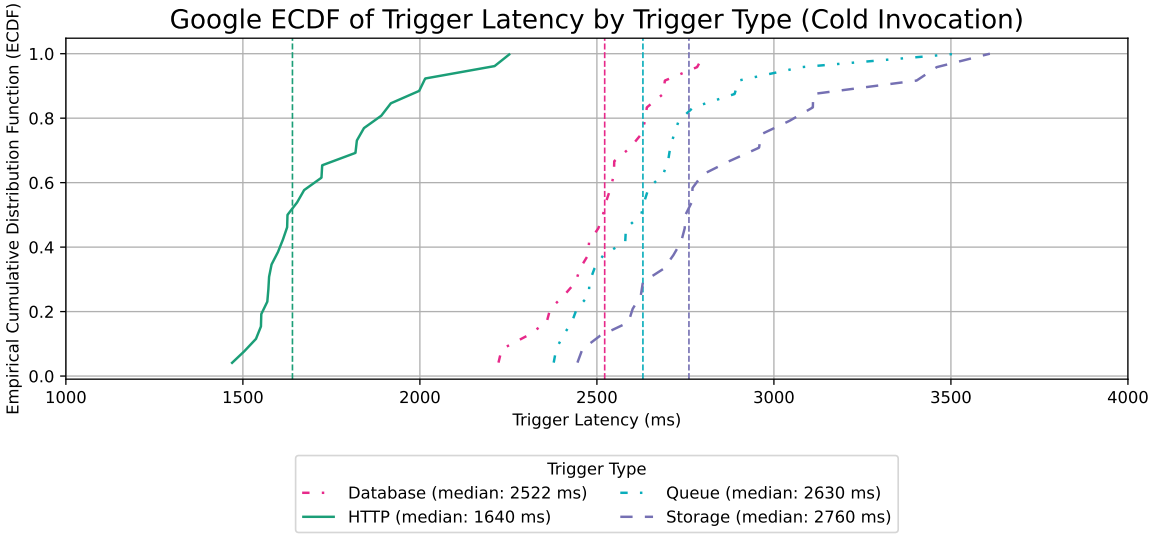

Figure 12: Google Cold Start

5.3.3 Discussion for Experiment 2

AWS warm starts demonstrate exceptional speed in handling HTTP and queue triggers, with median latency values of 105 ms and 151 ms, respectively. The storage trigger and database

trigger have higher latency occurrences with 1541 ms and 650 ms. In AWS, the newly added database trigger was twice as fast as the storage trigger however, it is still slower than HTTP and queue triggers. Differently, TriggerBench [JS22] research, our infrastructure has an extra latency occurrence due to the utilization of frameworks with +64 ms latency in HTTP, +40 ms latency in the queue, and +196 ms latency in storage. Both HTTP and queue have followed close latency values. However in our implementation, when there is a change in the S3 bucket storage, the system detects modification and handles larger data sets with detailed change specifications, which we believe is the reason for the larger latency occurrence in our infrastructure.

In comparison, Google's HTTP triggers, although they have a speed of 428 ms, do not exhibit the same level of performance as AWS's. In GCP the queue and database trigger performances are around 1200 ms and the storage trigger is still the slowest with 300 ms extra latency. In both cloud platforms, the warm storage latency was around 1550 - 1600 ms and AWS also outperformed Google in most of the trigger types with bigger differences in queue and database triggering.

During our analysis of cold start invocations on AWS and GCP, we discovered notable performance insights. On the AWS side, the trigger latencies in HTTP and queue increased to levels between 1050 - 1300 ms. This trend has followed in other trigger types adding an extra 1000 ms latency, except database which added 600 ms latency. On the other hand, in the Google Cloud function cold starts the latencies have been increased to around 1200 ms where all trigger types follow this characteristic. Like warm start HTTP was still the fastest option and storage was the slowest trigger type.

While running our cold/warm experiments in Experiment 2, compared to Experiment 1, GCP packages with 5.13 MB size and the usage of "aws-sdk" 97.2 MB size detailed in Section 5.1.2 causes a larger deployment space on the disc. We can also refer to the study conducted by Aakash Khochare [KKKS23] et al. They observed that a function with a package size of 100 KB had a cold start latency of around 500 ms, whereas a larger function with a size of 120 MB had a delay of approximately 2000 ms, which corresponds to our findings.

The results of our cold start experiment indicate that AWS offers reduced and constant latencies for time-sensitive operations, making it the preferred option for applications. Google continues to provide dependable performance for apps that are not highly sensitive to latency, despite experiencing greater latency in certain trigger types.

5.4 Experiment 3: Comparing AWS and Google HTTP Triggers Across Different Run Times (JavaScript, Python, Java) with Warm and Cold Starts

The experimentation approach entails evaluating and contrasting the latency and performance of HTTP triggered serverless functions in AWS and GCP. We conduct this evaluation under both warm and cold start conditions, utilizing several run times such as JavaScript, Python, and Java. We choose HTTP as a trigger type due to its excellent latency performance, making it ideal for systems that demand low latency. Widely used and adaptable, this approach ensures a consistent performance assessment in various settings. The selected programming languages encompass a wide range of serverless setups. JavaScript is a widely used scripting language for web applications and real-time interactions. On the other hand, Python, an interpreted language, executes at run time and is renowned for its simplicity and extensive libraries for data processing and machine learning.

5.4.1 Results

The boxes on the graph represent the interquartile range (IQR), which includes the center 50 percent of data points for each category, such as Java, Python, and JavaScript, for both cold and warm invocation types. The range spans from the 25th to the 75th percentile, with the line inside indicating the median. Additionally, the little circles are the individual outlier data points, which extend from the box.

According to AWS run time results in Figure 13, the cold start latency has been validated once more, with Java exhibiting the longest latencies. In particular, the time it takes to initiate Java programs from a cold start is roughly two times greater than the time it takes to initiate Python programs. On the other hand, the warm start latency data is 31 ms for Python, 29 ms for JavaScript, and 104 ms for Java. The impact of a cold start is significant. For example, in JavaScript, the 555 ms cold start occurrence compared to the warm start causes fifteen times more latency, which causes lag in response. We expected Java, a compiled language, to be the slowest. Close results were also found in Wang Liang [WLZ+18] et al.'s study, which is around 974 ms for Java version 8 cold start duration.

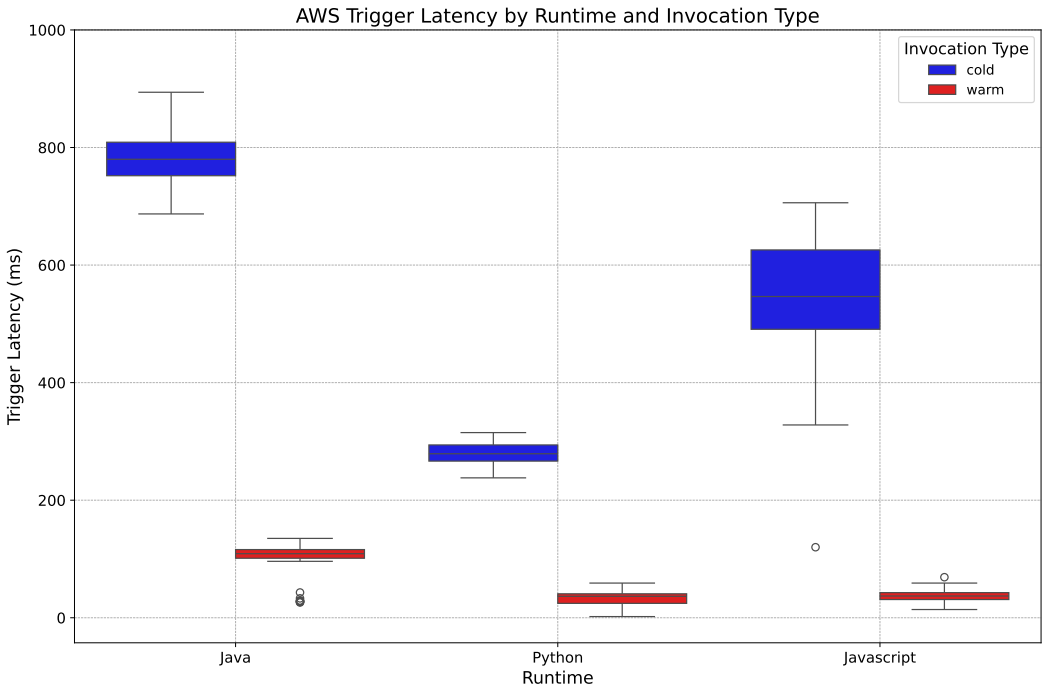

Figure 13: AWS HTTP Trigger Latency by Runtime

The Google Cloud results in Figure 14 display the latency of HTTP triggered Google Cloud Functions. The box figure illustrates the distribution of trigger latency for cold and warm invocations across three run times: Java, Python, and JavaScript. The delay at the initial startup is considerably greater for all run times when compared to the latency after the system has been running for a while.

Java has the longest cold start latency, averaging 1056 ms. The vast range indicates significant variation, validating the anticipated additional time required for Java, a compiled language, to initialize. Conversely, Java exhibits a significantly reduced warm start latency, averaging 53 ms, signifying a significant reduction in latency upon environment initialization.

Python exhibits a relatively mild cold start latency, with an average of 862 ms, which is lower than that of Java but greater than that of JavaScript. Python notably reduces the warm start latency to an average of 45 ms, demonstrating continuous performance enhancement once the environment warms up.

JavaScript has the shortest cold start latency compared to the other two run times, with an average of 995 ms. JavaScript exhibits a surprisingly low warm start latency, with an average of 30 ms, which demonstrates its effectiveness in processing consecutive invocations following the initial cold start.

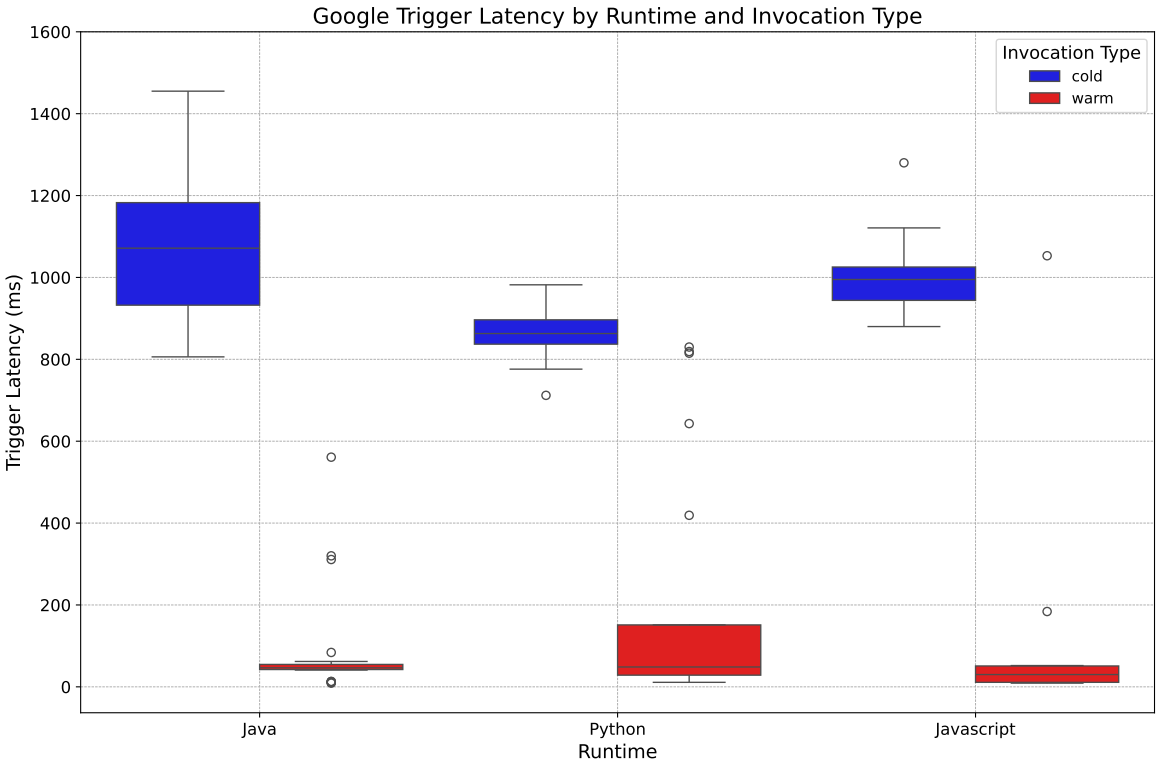

Figure 14: Google HTTP Trigger Latency by Runtime

5.4.2 Discussion for Experiment 3

At the beginning, trigger latency in milliseconds may appear insignificant, however, the impact of cold start duration becomes a crucial factor when we benchmark them. In AWS warm starts, Python and JavaScript have low latencies of 31 ms and 29 ms, respectively. But, Java show a bit larger latency of 103 ms. Nevertheless, in cold starts, Java presents a latency of 773 ms, which is higher than the latencies observed in other programming languages such as Python at 274 ms and JavaScript at 555 ms. Despite its limitations in terms of latency in serverless functions, Java remains a viable choice due to its numerous other advantages, like its durability, scalability, and strong performance in long-running processes.

When comparing the GCP findings, it is still evident that JavaScript has the most efficient run time for low warm start latency. This makes it particularly ideal for applications where quick subsequent invocations are essential. Additionally, Python and JavaScript results were close to AWS results with latency values of 31 ms and 29 ms respectively. In comparison to AWS Java 103 ms warm starts, GCP has a considerably faster speed of 34 ms.

When we compare cloud providers, it is clear that Java warm start invocations in GCP perform better than those on AWS, however, AWS cold starts of 773 ms were lower than GCP of 1056 ms. Also, across Python and JavaScript GCP shows higher cold starts, such as 855 ms in Python and 984 ms in JavaScript.

When we compare the HTTP trigger results of Experiment 2, where frameworks and dependencies have been utilized, with Experiment 3, we can say that the frameworks caused a larger application size, and this is directly proportional to trigger latency. For example, in Experiment 2, the JavaScript warm start HTTP latency was 105 ms, and for Experiment 3, we have observer 29 ms, which aligns with Experiment 1's 37 ms HTTP trigger latency while reproducing TriggerBench [JS22] research. Additionally, our data indicates that avoiding "aws-sdk" made AWS Lambda function two times faster in cold starts with latency values of 1054 ms to 555 ms.

5.5 Experiment 4: Comparing AWS and Google Cloud Platform Chain of Functions

In this experiment, we wanted to evaluate the effectiveness of chain formation of different AWS and GCP triggers in warm/cold invocation. The experiment was designed to precisely assess the latency at each stage in a succession of cloud functions, mirroring a typical real-world usage scenario where several functions are executed in a sequence.

While assessing the cold start, the experiment entails quantifying the duration it takes for a system to start from a cold state and its overall efficiency while executing a series of interconnected serverless operations in both AWS and GCP environments. We believe this experiment specifically highlights the overlaying initialization latencies in mostly preferred micro-service architectures with variant event-triggering options. During this experimentation total of 100 innovations have been conducted per cloud provider (AWS and GCP) and during each invocation a chain of four trigger types have been triggered while passing and carrying the same ”TraceId”s, including HTTP, queue, storage, and database triggers. The functions were written in JavaScript with the utilization of frameworks previously described in Section 5.1.2. The objective was to evaluate the performance of serverless architectures, while managing a sequence of functions across various triggering scenarios, with a focus on measuring the effects on latency and efficiency. We will focus specifically on the accumulation of trigger latency, as the occurrence of many cold starts might have a detrimental impact on the overall performance of the infrastructure.

5.5.1 Results for Warm Start

Our warm start AWS observations in Table 6 revealed that the HTTP has the lowest latency average and median values among the evaluated components, measuring 166,18 ms, and 129 ms respectively. This indicates that it was the most efficient component in our configuration. On the other hand, the storage latency exhibited an average of 1467.27 ms and a median of 1600 ms, indicating notable delays in responding to changes in the S3 bucket. Database posed the most significant issues, with average and median values of 1707.48 ms and 1804 ms, respectively, clearly identifying it as a bottleneck in our system that is explained in Section 5.5.3. Our analysis determined that the difference in these latencies accumulated significantly added to the standard deviation of the total system delay.

Secondly, during examination of the Google Cloud Function’s latency performance, we saw a clear and consistent pattern in the latency characteristics which is shown in Table 7. The analysis revealed that HTTP Latency has the highest level of efficiency, exhibiting the lowest average latency of 545.04 ms and a median of 276 ms. This indicates a strong ability to effectively process HTTP requests. On the other hand, Queue Latency and Storage Latency showed significantly higher average and median values, at 1537.43 ms and 1546.43

	HTTP	Queue	Storage	Database	Total
Median	129	430	1600	1804	3795
Max	702	2336	2597	3086	6785
Min	62	96	121	423	1083
Standard Deviation	141.55	583.99	678.91	883.39	1510.99
Average	166.18	632.04	1467.27	1707.48	4047.74

Table 6: AWS Chain of Latency, Warm Start

ms respectively. This suggests that there are more substantial delays, which are likely caused by the need to handle and process complicated data. The Database Latency, albeit lower than Queue and Storage, had an average of 1274.43 ms and a median of 1162 ms. The combined effect of these delays was evident in the Total Latency, with an average of 4903.34 ms and a median of 3743 ms, highlighting how each component affected the overall performance of the system.

	HTTP	Queue	Storage	Database	Total
Median	276	1267	1313	1162	3743
Max	1762	2851	3246	3175	9911
Min	156	2851	3246	3175	9911
Standard Deviation	566.21	657.86	955.41	955.78	2847.58
Average	545.04	1537.43	1546.43	1274.43	4903.34

Table 7: Google Cloud Platform Chain of Latency, Warm Start

5.5.2 Results for Cold Start

A final experiment was conducted to relate the best business scenarios and see the cold start scenario. In a single operation with a single function, cold start duration may appear as a small latency; however, when multiple cloud functions collaborate in a latency-sensitive implementation, these latency levels compound, leading to a significant latency. Furthermore, the real-world application may suffer from unstable latency, which will harm the computing experience.

Table 8 indicates the latency measurements obtained from an experiment that involved a sequence of cloud function triggers in AWS Cloud Functions. The data shows the latency averages of different triggers (HTTP, Queue, Storage, and Database), which add up to an average total latency of 6927,21 milliseconds. Compared to the more efficient AWS trigger types like HTTP and queue, the database and storage triggers granted significantly higher latency.

Our experiments with Google Cloud resulted in Table 9 in which, we found that HTTP triggers worked better than other forms of triggers when it came to cold start invocations.

	HTTP	Queue	Storage	Database	Total
Median	1141	1296	1752	2204	6364
Max	1494	1913	3045	5695	10746
Min	848	1077	1258	1650	5381
Standard Deviation	113.41	202.21	340.8	1043.14	1388.23
Average	1150.94	1334.17	1805.84	2636.25	6927.21

Table 8: AWS Chain of Latency, Cold Start

However, event-driven triggers resulted in longer latency periods. Throughout all calculations, we noticed a substantial delay in activation, with durations reaching a maximum of 10294,27 milliseconds during execution.

	HTTP	Queue	Storage	Database	Total
Median	1787	2690,5	2792,5	2687	9980,5
Max	2713	4337	4162	3603	13815
Min	1574	2310	2527	2517	9403
Standard Deviation	198.72	397.56	238.15	222.54	593.19
Average	1816.16	2778.47	2902.27	2784.05	10294.27

Table 9: Google Cloud Platform Chain of Latency, Cold Start

5.5.3 Discussion for Experiment 4

The data analysis of AWS warm starts has provided crucial insights into the performance dynamics of the system. The minimal variability in HTTP latency indicates that web service calls are being efficiently managed. On the other hand, the greater fluctuation and longer waiting time in Database and Storage activities suggest that these aspects are significant causes of system delays. While queue operations are not as delayed as database transactions, they nonetheless exhibit significant fluctuation, which can affect performance in high-load scenarios. To address the database bottleneck, we first need to revise database initialization and network connection processes. When a storage cloud function is first executed, the database client is set up with the correct credentials and region settings. At this stage, the client is ready for execution but has not made any network connections yet. When we want to make changes in the database to trigger the database cloud function, the algorithm specifically requires the client to communicate with the DynamoDB servers. At this point in the cold start, the latency issue kicks in while the AWS SDK establishes a network connection. As we are measuring the invocation time right before editing database, we have been including the network connection time into the database trigger latency. Compared to Experiment 2 database warm results, our Infra module was in high workload, directly triggering the database receiver function that keeps the database network connected. However, especially in this experiment, until the chain of triggering reaches the database, the DynamoDB network connections are

lost from the cloud provider, and we observed high latency in database triggering. The combined impact of these elements is seen in the total latency, which had an average of 4047,74 ms and a median of 3795 ms.

Additionally, we obtained some significant warm start observations in GCP. The high performance of HTTP processing indicates that the web service components have been optimized well. Nevertheless, the increased and fluctuating larger latencies have also been seen in Queue and Storage processes like AWS results which also shows that these event triggers were not responsive. We saw that the Queue, Storage, and Database latencies consistently displayed uniform minimum and maximum values. This led us to investigate the possibility of abnormalities in data gathering or fundamental system restrictions that restrict latency to certain thresholds. In addition, we noticed potential anomalies in the warm start data, where intermittent cold starts had a large impact on both the mean and the variability. This happened because we were measuring the passage of time, and the existence of certain delays in warm start conditions had a greater impact on the overall result. However, the problem was not as noticeable in the cold start data, as they usually had consistently higher values, and the effect of any warm start instances was rather insignificant.

When we compare, how AWS and Google Cloud Platform handle cold starts, the AWS data indicated that database latency has been the primary contributor to the average total latency of 6927,21 milliseconds. On the other hand, the storage latency was the main latency contributor in GCP and the total latency was much larger nearly double that of AWS which is 10294,27 milliseconds. In the cold start of both experiments HTTP outperformed other trigger types and provided an optimal solution, but still, the AWS HTTP function was 637 milliseconds faster than GCP.

6 Conclusion

Our analysis involved assessing serverless function triggers on both AWS and Google Cloud Platform, uncovering notable disparities in how each platform handles cold starts and overall trigger performance. While studying frameworks, we have applied cloud frameworks described in Section 5.1.2 in Experiment 2 which obtained additional insights that have a considerable impact on the time of cold starts. In cold starts, Experiment 2 showed that utilizing AWS SDKs led to HTTP trigger latencies that were twice as long as the 555 ms recorded in the HTTP JavaScript runtime results of Experiment 3. Similarly, the utilization of Google frameworks on GCP increased cold start latency from 995 ms to 1640 ms. This finding supports earlier research of Aakash Khochare [KKKS23] et al. that has shown a connection between larger deployment sizes and increased delays in cold start times. On the other hand, AWS and GCP the warm starts in Experiment 4 implementations indicates significantly reduced delays. Specifically, the median warm start delay of AWS was 129 ms and 276 ms in GCP, which represents a speed of 10 times faster in AWS and 5 times faster in GCP compared to their cold start performances.

During our examination of cold start invocations on AWS and GCP, we noted substantial disparities in performance. AWS experienced elevated latencies for HTTP and queue triggers, with durations ranging from 1050 to 1300 ms. Similar increases in latency were observed for all trigger types, except for database triggers which contributed only 600 ms. In contrast, cold starts of Google Cloud functions consistently exhibited latencies of around 1200 ms for all types of triggers.

The latest version of Node.js 20, provides exceptional run time performance on GCP and AWS, namely in low warm start latency. This makes it well-suited for applications that require rapid reaction times. However, in terms of handling cold start invocations, Node.js ranked second with latencies of 555 ms in AWS and 955 ms in GCP, after Java 21. Python 3.12 had consistent performance in both cold and warm beginnings, making it ideal for a wide range of applications.

Our long chains of sequential serverless trigger executions revealed that AWS encounters significant delays while dealing with database triggers, resulting in a total average latency of 6927.21 milliseconds. On the other hand, GCP faces challenges with storage triggers, resulting in a greater total latency of 10294.27 milliseconds. Notwithstanding these challenges, HTTP triggers performed quite well in both settings, with AWS's HTTP functions exhibiting a 646-millisecond speed over GCP.

In summary, our research offers valuable insights into the performance comparison between AWS and GCP. It highlights the importance of improving cold start procedures to make FaaS infrastructures more efficient. Developers and system architects can enhance and meet performance standards for modern applications by comprehending the distinctive features of

each platform and making well-informed decisions.

To summarize, our research enhances the scholarly comprehension of serverless trigger performances and provides experimental results to better influence architectural serverless trigger decisions in different setups. Our research emphasizes the importance of thorough benchmarking across different triggers, run times, and cloud providers. It offers crucial information for both academic investigation and real-world implementation in the rapidly changing field of digital technologies.

7 Limitations

To achieve cost optimization during infrastructure deployment, our first approach involved utilizing Accenture's AWS and Google Cloud accounts for our research. Nevertheless, to replicate the TriggerBench [JS22] study, it was necessary to have administrative accounts. This was because the process involved setting up and dismantling the necessary infrastructure, as well as assigning roles to functions using Pulumi [pul24] Infrastructure as a Service. We did not have the authority to assign this software within Accenture's accounts, which led us to develop our serverless benchmarker, Infra module, and serverless functions.

We selected the Europe West 3 regions offered by our cloud providers as our main testing locations for conducting these experiments in the Netherlands, to write suitable implementations for Accenture Netherlands and other local companies. In AWS Europe West 3 relates to Paris, France, and in GCP Europe West 3 is Frankfurt, Germany. Additionally, choosing this region was to have geographic price benefits and meet regulatory contexts in Europe, thereby promoting the relevance and applicability of our findings for local enterprises.

For evaluating the current iterations of Google Cloud function triggers and comparing them to their AWS equivalents. An obstacle we faced was the outdated language versions in clouds which is why we could not reproduce the experiment 5.2 with the same Node version. This required us to update to the most recent run-time versions and proceed with a comparison, as the previous ones were obsolete and no longer supported.

During the measurement of cold start durations, we conducted tests where we triggered functions every 30 minutes. This process lasted for three days and resulted in the collection of more than 200 data points. Conducting tests on all potential combinations of cloud providers, trigger types, and run times presented a significant problem. So, in this study, we focused on implementation scenarios that were sensitive to latency and ran tests on sequential HTTP triggers to learn more about how their performance changes over time.

8 Future Research

Our research findings have established a strong basis for further investigations into the performance of serverless services on different cloud platforms and run time environments. Although our analysis largely concentrated on AWS and Google Cloud utilizing JavaScript, there is considerable potential to expand this research to other programming languages and cloud providers.

Encompassing additional cloud providers: such as Azure, IBM Cloud, and Oracle Cloud, in our research would enhance our comprehension of how diverse cloud infrastructures manage serverless services. Every cloud provider possesses distinct operating characteristics and optimizations that can have a substantial impact on the performance of serverless applications, especially in terms of cold start latencies and run-time efficiency.

Comparative analysis across cloud providers: investigating new cloud providers can reveal larger patterns and specific performance metrics. Conducting this analysis will not only confirm the accuracy of our findings but also offer valuable information about the potential of serverless functions to adjust and expand comparison in various environments.

Examining event-driven triggers in Python and Java: Subsequent investigations may delve into the efficiency of event-driven triggers utilizing Python and Java. This would offer a thorough comparison to our existing results derived from JavaScript implementations. Conducting tests in both Python and Java can assist in identifying optimizations and difficulties that are specific to each language within the framework of event-driven architectures.

By exploring this future research, the performance standards for serverless computing and contribute to the advancement of more efficient, dependable, and scalable cloud-based applications.

References

[ABC14] Ioannis Arapakis, Xiao Bai, and B. Barla Cambazoglu. Impact of response latency on user behavior in web search. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, page 103–112, New York, NY, USA, 2014. Association for Computing Machinery.

[APP21] Ioannis Arapakis, Souneil Park, and Martin Pielot. Impact of response latency on user behaviour in mobile web search. In *Proceedings of the 2021 Conference on Human Information Interaction and Retrieval*, CHIIR '21, page 279–283, New York, NY, USA, 2021. Association for Computing Machinery.

[awsa] Amazon web service lambda functions java implementation examples. `https://github.com/serverless/examples/tree/master/aws-java-simple-http-endpoint`. (Accessed: April 5, 2024).

[awsb] AWS SDK for JavaScript. `https://aws.amazon.com/sdk-for-javascript/`. (Accessed: March 22, 2024).

[aws19] Tracking the state of lambda functions. `https://aws.amazon.com/blogs/compute/tracking-the-state-of-lambda-functions/`, 2019. Accessed: May 28 , 2024.

[aws22] Aws lambda runtime environment. `https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html`, 2022. Accessed: May 12, 2024.

[aws24] Aws x-ray developer guide. `https://docs.aws.amazon.com/xray/latest/devguide/aws-xray.html`, 2024. (Accessed: July 11, 2024).

[CKB+20] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing, 2020.

[Den22] Rui Deng. Benchmarking of serverless application performance across cloud providers: An in-depth understanding of reasons for differences, 2022.

[EvEI18] Sacheendra Talluri Erwin van Eyk, Lucian Toader and Alexandru Iosup. Serverless is More: From PaaS to Present Cloud Computing. `https://www.researchgate.net/publication/328088482_Serverless_is_More_From_PaaS_to_Present_Cloud_Computing`, September 2018.

[Fir24] Firebase Documentation Team. Firebase admin sdk documentation. Online, 2024. Accessed: date-of-access.

[gooa] Google Cloud CLI Documentation. `https://cloud.google.com/sdk/docs`. (Accessed: March 10, 2024).

[goob] Google Cloud Logging NPM Package. `https://www.npmjs.com/package/@google-cloud/logging`. (Accessed: June 12, 2024).

[goo23] Create and deploy an http cloud function with java. `https://cloud.google.com/functions/docs/create-deploy-http-java`, 2023. (Accessed: June 3, 2024).

[GPB+21] Martin Grambow, Tobias Pfandzelter, Luk Burchard, Carsten Schubert, Max Zhao, and David Bermbach. BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms. `https://arxiv.org/abs/2102.12770`, 2021.

[gra24] Grafana k6 Documentation. `https://grafana.com/docs/k6/latest/`, 2024.

[JMW18] T. Heckel J. Manner, M. Endreß and G. Wirtz. Cold start influencing factors in function as a service. *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2018.

[JS22] Oskar Grönqvist Henrik Tao Henrik Lagergren Jan-Philipp Steghöfer Philipp Leitner Joel Scheuner, Marcus Bertilsson. Triggerbench: A performance benchmark for serverless function triggers. *IEEE International Conference on Cloud Engineering (IC2E)*, 2022.

[KKKS23] Aakash Khochare, Tuhin Khare, Varad Kulkarni, and Yogesh Simmhan. Xfaas: Cross-platform orchestration of faas workflows on hybrid clouds. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 498–512, 2023.

[KWB+20] Jörn Kuhlenkamp, Sebastian Werner, Maria C. Borges, Dominik Ernst, and Daniel Wenzel. Benchmarking elasticity of faas platforms as a foundation for objective-driven design of serverless applications. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 1576–1585, New York, NY, USA, 2020. Association for Computing Machinery.

[loc] Locust.io Official Website. `https://locust.io/`. (Accessed: 2024).

[LWC+23] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing. *ACM Trans. Softw. Eng. Methodol.*, 32(5), jul 2023.

[MAC20] Horácio Martins, Filipe Araujo, and Paulo Rupino Cunha. Benchmarking Serverless Computing Platforms. `https://www.researchgate.net/publication/342750052_Benchmarking_Serverless_Computing_Platforms`, December 2020.

[mat24] Matplotlib. `https://matplotlib.org/`, 2024. Accessed: May 07 , 2024.

[MFKS20] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. Faasdom: a benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, DEBS '20, page 73–84, New York, NY, USA, 2020. Association for Computing Machinery.

[nod23] node-schedule. `https://www.npmjs.com/package/node-schedule`, 2023. (Accessed: July 12, 2024).

[NSK20] Umesh Bellur Nikhila Somu, Nilanjan Daw and Purushottam Kulkarni. Panopticon: A comprehensive benchmarking tool for serverless applications. *12th International Conference on Communication Systems Networks*, 2020.

[pul24] Pulumi is an open source infrastructure as code tool for creating, deploying, and managing cloud infrastructure. `https://www.pulumi.com/docs/`, 2024.

[SA20] Khondokar Solaiman and Muhammad Abdullah Adnan. Wlec: A not so cold architecture to mitigate cold start problem in serverless computing. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pages 144–153, 2020.

[SDSL22] Joel Scheuner, Rui Deng, Jan-Philipp Steghöfer, and Philipp Leitner. Crossfit: Fine-grained benchmarking of serverless application performance across cloud providers. In *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, pages 51–60, 2022.

[sea24] Seaborn. `https://seaborn.pydata.org/`, 2024. Accessed: May 21 , 2024.

[ser] Serverless Examples – A Collection of Boilerplates and Examples of Serverless Architectures Built with the Serverless Framework on AWS Lambda, Microsoft Azure, Google Cloud Functions, and More. `https://github.com/serverless/examples`. (Accessed: February 2024).

[ser14] Serverless framework documentation. `https://www.serverless.com/framework/docs`, 2014. Accessed: 2 July, 2024.

[SET+22] Joel Scheuner, Simon Eismann, Sacheendra Talluri, Erwin van Eyk, Cristina Abad, Philipp Leitner, and Alexandru Iosup. Let's trace it: Fine-grained serverless benchmarking using synchronous and asynchronous orchestrated applications, 2022.

[SvD12] Martin L. Scholtus and Dick van Dijk. High-frequency technical trading: The importance of speed. Tinbergen Institute Discussion Paper TI 2012-018/4, Econometric Institute and Tinbergen Institute, Erasmus University Rotterdam, P.O. Box 1738, 3000 DR Rotterdam, The Netherlands, Feb 2012. JEL codes: G10, G14, G20. Email addresses: scholtus@ese.eur.nl (Martin L. Scholtus), djvandijk@ese.eur.nl (Dick van Dijk).

[UAG21] Dmitrii Ustiugov, Theodor Amariucai, and Boris Grot. Analyzing tail latency in serverless clouds with stellar. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 51–62, 2021.

[VFA20] Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee. Cold start in serverless computing: Current trends and mitigation strategies. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–7. IEEE, 2020.

[WLZ+18] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 133–145, USA, 2018. USENIX Association.