PERFORMANCE CHARACTERISTICS OF FUNCTION AS A SERVICE PLATFORMS

KIM LONG NGO

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
NOVEMBER 2021

Abstract

Function as a Service (FaaS) is a new cloud technology where resource management is automatically handled by cloud providers. However, the automated resource management also reduces the transparency needed for software engineering tasks and additional FaaS' characteristics such as cloud function idle timeout, auto-scaling policies, response time to bursting workloads are unknown to software engineers. In this thesis, we propose a methodology to measure the cloud function instance idle timeout. Next, we characterize FaaS' scalability and elasticity using intensive workloads. Finally, we propose a strategy to improve the FaaS' performance under saturation scenario. The results show that cloud function instances are decommissioned if being left idle beyond certain period. Load and performance experiments reveal that different cloud platforms adopt distinct auto-scaling policies and when FaaS has reached the upper concurrency limit, a workload smoother can help to boost the system's success rates from 60 - 80% to 99 - 100%.

Acknowledgements

On completion of my thesis, I would like to express my heartfelt gratitude to my supervisors - Professor Zhen Ming (Jack) Jiang and Professor Marin Litoiu. Through their guidance, I had a chance to study about Serverless - Function as a Service technology as well as learnt how to conduct a formal academic research. I would like to thank Professor Joydeep Mukherjee for his continuous feedback on my research. I am much obliged to Professor Hamzeh Khazaei and Professor Manar Jammal for evaluating my thesis as committee members. In addition, I would like to offer thanks to York University for funding my research and giving me an opportunity to share my knowledge and experience to undergraduate students. I extend thanks to IBM Company for offering an opportunity to conduct a research internship during my study.

Finally, I would like to thank my family, my associate pastor and my girlfriend for being on my side whenever I am in need. Without their relentless support and inspiration, I could not have accomplished my research.

Table of Contents

List of Tables

List of Figures

Co-Authorship

This thesis is based on my research work listed below. In these research, I contributed in the following ways except where noted: materializing the initial research idea, researching related work, implementing cloud functions, conducting the experiments, analyzing result data and writing the first draft of the papers. Parts of this thesis have been submitted as follow:

- Kim Long Ngo, Joydeep Mukherjee, Zhen Ming (Jack) Jiang, Marin Litoiu, Has Your FaaS Application Been Decommissioned Yet? - A Case Study on the Idle Timeout in Serverless Infrastructure: Submitted to IEEE Software; Manuscript ID: SW-2021-08-0144.

  Contribution: Implementation, experimentation and evaluation.

- Kim Long Ngo, Joydeep Mukherjee, Zhen Ming (Jack) Jiang, Marin Litoiu, Evaluating the Scalability and Elasticity of Function as a Service Platform: Submitted to International Conference on Performance Engineering (ICPE - Industry);

Manuscript ID: ACM/SPEC ICPE 2022 Submission 35

Contribution: Implementation, experimentation and evaluation.

1 Introduction

Software industry has evolved rapidly with the transformation from monolithic architecture operating on Virtual Machines (VM) to Microservices running on lightweight containers such as Docker [1]. To relieve the software engineers from operational tasks such as resource management, a new technology called Function as a Service (FaaS) was popularized since 2014 [2]. FaaS is an event-driven cloud platform where software engineers can focus on business logic and leave the infrastructure management to cloud providers [3]. FaaS implementations or cloud functions are snippets of source code developed in supporting programming languages and executed inside a container. When cloud function is triggered, cloud provider will create a new container instance (i.e., cloud function instance), deploy the source code, initialize and execute the business logic within the instance [4] [5]. FaaS platform optimizes cloud provider's resource utilization by only deploying the cloud functions on-demand and decommissioning them after an inactive period.

The nature of automated resource management has brought both advantages

and challenges to the FaaS' applicability. On one hand, software engineers can be relieved from cloud function instance provisioning/decommissioning tasks. On the other hand, they will not have full control over the infrastructure like traditional technologies such as VM or Containers. Moreover, the engineers may not know when their cloud function instances are recycled and hence experience unexpected behavior. When a cloud function is first invoked, cloud platform needs to start up a container for execution. This process introduces additional delay in function execution and is addressed as cold start. Cold start can occur when cloud function first started or cloud instance is left idle beyond certain time period (*idle timeout*) or while scaling up the application. Since cold start introduces execution delays, the more cold start a function experiences, the more overheads resulted in system's overall response time. As a consequence, knowing when a cloud function instance is decommissioned will help software engineers understand FaaS' characteristics and thus appropriately design and implement their software systems with FaaS. One of the promising FaaS' features is resource auto-scaling, that is the automatic provisioning and de-provisioning of computing resources such as memory and CPU cycles, when the incoming traffic changes [6]. This feature while simplifying the operational aspect of software engineering also poses a challenge to performance planning. Software engineers may need to know in-depth about how each cloud provider scales their cloud functions, either vertically (by adding more resources to

the same container) or horizontally (by adding more containers), how reactive the scaling is or what are the limitations when scaling up or down. Hence, a load and performance benchmark examination is essential to justify if this technology can deliver the benefits of auto-scaling

To make FaaS a true 'serverless platform' (alternative name of FaaS), cloud providers do not specify their underlying implementation including how long a cloud function can be idle. However, researchers have shown the importance of knowing FaaS runtime properties and have partially characterized the FaaS function instance's idle timeout [7] [8] [9]. Software industry authors also studied this aspect and reported their experimental results on web blogs [10] [11] [12]. These studies are helpful to software engineers since they provide a thorough knowledge about FaaS runtime. Moreover, our experimental results show that FaaS providers regularly update their implementations. For instance, our study between 01/2020 and 07/2021 detects that AWS Lambda cloud function instance idle timeout has been reduced from 10 minutes to 5 minutes and this was not communicated.

On scalability aspect, most previous studies on FaaS focus on evaluating the performance by running heavy CPU, Memory, I/O and Network benchmarks on different FaaS platforms [7] [13] [14] [15]. There were only a few research studies that focused on the auto-scaling aspect of FaaS. Those studies only used a small number of concurrent clients and low intensity workloads. Furthermore, cloud providers are

constantly evolving the cloud platforms and auto-scaling characteristics change over time and hence cause deviations from the expected behavior.

In this thesis, we characterize FaaS instance idle timeout by presenting a methodology to measure this duration up to minute-accuracy. To keep the cloud function instance from decommissioning, a common workaround is keep-alive technique which is to poll the cloud function at regular interval to preserve the infrastructure [8]. We study the impact of keep-alive on the cloud function instance idle timeout. Furthermore, we also measure the cloud function instance idle timeout at different times (i.e., checkpoint) from 01/2020 to 07/2021 and report these evolutionary changes. Furthermore, we describe an empirical study of FaaS' auto-scaling and report the results of our systematic load and performance experiments. The results show auto-scaling strategy is implemented differently by different cloud providers. Then, we examine the use case when FaaS has spawned all the allowable cloud function instances to serve the traffic (i.e., the upper concurrency capacity limit has been reached or FaaS has been saturated). Next, we evaluate the effects of introducing a workload smoother which is a software component located in-front of a target system to smooth the workload and prevent overloading. Our studies were conducted on three popular cloud providers, namely Amazon AWS Lambda (AWS), IBM Cloud Function (IBM) and Azure Cloud Function (Azure).

1.1 Contributions

The contributions of this thesis are:

- We present a thorough methodology to measure cloud function instance idle timeout up to minute-accuracy. Furthermore, we also explore the cloud function instance maximum idle timeout when keep-alive technique is used and based on this finding, we advise software engineer in which use case they should use keep-alive technique.
- We summarize the cloud function instance idle timeout evolution through a period of one year and a half (01/2020 - 07/2021).
- We simulate intensive workload scenarios, evaluate and report FaaS' auto-scaling characteristics on different cloud platforms.
- We propose and show the advantages of a workload smoother which implements the *Queue-Based Load Leveling* microservice design pattern [16]. Our prototype implementation shows that it can achieve a 99 - 100% success rate compared to 60 - 80% when FaaS' system is saturated.

1.2 Thesis Organization

The remaining of this thesis is organized in the following chapters. Chapter 2 introduces the background and related work of this thesis. Chapter 3 discusses the characterization study of FaaS' idle timeout and its evolutionary changes. Chapter 4 presents the empirical study of FaaS' scalability and elasticity and report the advantages of employing a workload smoother when FaaS has been saturated. To be noted, when this thesis was under editing, Chapter 3 had been submitted to IEEE Software and passed the first round of review, it is being prepared for second round of review. Chapter 4 had been submitted to ACM/SPEC International Conference on Performance Engineering 2022 (ICPE - Industry) and accepted as a short industry/experience track paper. Chapter 5 concludes the thesis and outlines our future work.

2 Background and Related Works

This chapter presents the background and related works on FaaS cloud instance timeout followed by scalability and elasticity.

2.1 Background and Related Work for FaaS Idle Timeout

2.1.1 Background

FaaS is a new technology that allows software engineers to fully focus on business logic implementation and leave resource and server management to cloud providers. When traffic first arrives to a cloud function, FaaS platform will provision, deploy and initialize the source code package to a cloud function instance container with user's defined configurations (e.g., allocated memory, function entry point and so on) [4]. This process usually lasts between 300 ms to 24 seconds [17] and the first request is addressed as *cold start*, other execution is considered as *warm start*. FaaS charges the user based on the cloud function's execution duration hence it is essential to prevent a cloud function from running indefinitely. To achieve this, all

cloud platforms imposed an **execution timeout** which is the maximum duration a cloud function is allowed to execute the logic. Beyond this period, cloud platform will stop the execution and return a failed response. Cloud providers allow software engineer to configure their cloud function's execution timeout according to their requirement. Table 2.1 summarizes the up-to-date documented characteristics including maximum execution timeout obtained from cloud providers.

Characteristic	**AWS**	**IBM**	**Azure**
Max Execution Timeout (mins)	15	10	10
Max Allocated Memory (GB)	10	2	1.5
Billing Granularity (ms)	1	100	1

Table 2.1: FaaS Documented Characteristics

To minimize the overhead caused by cold start, after serving the current traffic, cloud platform will retain the infrastructure for a short time to accommodate future requests. If there is no traffic within this idle period, cloud platform will decommission and return the computing resources for other purpose. We address this idle period as cloud function **idle timeout**. To our knowledge, cloud providers do not officially document how long the idle timeout lasts.

2.1.2 Related Works

FaaS infrastructure retention has been studied when researchers explored FaaS' characteristics. Lloyd et al. [7] in their study about serverless computing reported that all FaaS VMs and containers were removed after being left idle for 40 minutes and by using keep-alive technique at 5-minute interval, host VMs were recycled every 4 hours. Maissen et al. [9] described that on AWS and IBM, it took around 10 minutes of no activity for cloud function instance to be recycled. This period for Azure cloud function was higher at 20 minutes.

We noted that previous work studied the cloud function instance idle timeout in combination with other FaaS' characteristics. Furthermore, experimental results were presented at the time of publication and it is challenging to reproduce the experiments for updated results. Our study is different from previous ones in which we specifically focused on cloud function instance idle timeout on three popular cloud platforms. Moreover, we also summarize the evolutionary changes related to cloud function instance idle timeout from 01/2020 to 07/2021 and discuss the use of keep-alive technique. Our proposed methodology are straightforward and source code is published for reproducing purposes [18].

2.2 Background and Related Work for FaaS Scalability and Elasticity

2.2.1 Background

2.2.1.1 FaaS Overview

Since its introduction in 2014, FaaS has gained the attention of both academic researchers and industry practitioners for its auto-scaling feature. FaaS offers two scaling types : vertical and horizontal scaling. Vertical scaling refers to increasing processing power by using more powerful resources such as strong CPU, more RAM memory and so on. Amazon AWS Lambda and IBM Cloud Function allow users to configure their cloud functions' memory size while Microsoft Azure Cloud Function automatically manages the memory configuration [19] [20] [21].

In contrast to vertical scaling, horizontal scaling adds container instances as required to elevate processing capacity. When traffic increases, FaaS platforms automatically provision more instances to prevent the system from being overwhelmed [6]. While providing auto-scaling, all FaaS cloud platforms also impose an upper concurrency limit. AWS Lambda restricts the maximum concurrency at 1,000 concurrent clients which means a Lambda function can handle at most 1,000 concurrent requests. Beyond this threshold, excessive requests will be throttled

and returned with "Too Many Request" error (i.e., HTTP-429) [22]. HTTP-429 indicates the incoming workload has exceeded the system's capacity and hence some requests are returned as failure. In addition, this error also signals to calling clients to refrain from further re-try. Software engineers can reduce AWS Lambda concurrency level by using *reserved concurrency* option in function's management console [23]. Similarly, IBM cloud function also caps the maximum concurrent executions to 1,000 but does not provide reducing option [21]. Under basic Consumption Plan, Microsoft Azure allows a cloud function to scale up to 200 instances [20]. However, different from AWS Lambda and IBM Cloud Function, each Azure cloud function instance can serve more than one request at a time and by default, this value is set to 100 concurrent requests per instance [24].

2.2.1.2 Scalability and Elasticity Overview

Kuperberg et al. [25] defined Application Scalability as "the application maintains its performance goals/Service Level Agreements (SLAs) even when workload increases". Accordingly, "Platform Scalability is the ability of the execution platform to provide as many (additional) resources as needed (or explicitly requested) by an application". The two definitions were later combined into one as "Scalability is the ability of the system to sustain the increasing workload by making use of additional resources" [26]. Hence, the scalability of a platform can be measured as the

number of additional resources it can provide so that the application can sustain a workload burst.

Elasticity, on the other hand, is described as the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an automatic manner [26]. Lehrig et al. [27] in their systematic literature review on definitions and metrics of scalability, elasticity and efficiency in cloud computing emphasize that elasticity is mainly related to "the change in resources and the reaction time is of importance, which indicates the importance of speed in an elastic system". That means elasticity of a system can be evaluated as the speed of scaling up a system from under-optimal to optimal states and the faster a platform can add more instances, the better the elasticity is.

2.2.1.3 Cloud Design Patterns Overview

Cloud design patterns are documented solutions pertaining to specific software engineering problems. Similar to source code design patterns, each cloud design pattern includes the discussion about the context, problem, solution, issues and considerations. Cloud design patterns act as a guideline to software engineers and architects while designing their applications. The goal of applying cloud design patterns is to ensure that the software system is reliable, scalable and secured in the cloud environment. Taibi et al. [28] reviewed 32 patterns that can be used

for FaaS, which were classified as orchestration, aggregation, event-management, availability, communication, and authorization. Some of these patterns, such as queue-based load leveling, have been used with microservices [29]. We implemented the queue-based load leveling pattern for FaaS. The goals was to see if this pattern can improve the performance and if so, by how much.

2.2.2 Related Works

There are many publications on benchmarking FaaS platforms and most of them focus on CPU, Memory, Network and IO utilization [7] [13] [14] [15]. There are only a few studies concentrated on the scalability of this technology. Kuhlenkamp et al. [30] presented an elasticity benchmark for four FaaS popular cloud providers, AWS Lambda, IBM, Azure and Google Cloud. They used Node.js and a variety of metrics such as reliability, request-response latency, throughput, execution cost to evaluate the FaaS' scalability. The benchmark was designed to send slow ramping requests (60 seconds). Martins et al. [31] defined a suite of seven tests to benchmark the raw performance of serverless platforms under normal and heavy load conditions. The test suite measured latency and throughput metrics to benchmark the performance of AWS Lambda, Azure, Google and IBM cloud functions. In the concurrent load test, the experiment started with a single invocation and, every 900 seconds, added another until a maximum of 30 concurrent invocations. The

results showed that AWS Lambda, Azure and Google cloud functions exhibited almost linear throughput increase when the number of concurrent requests accelerated. Our study is different from above research in the following ways: we evaluate FaaS' scaling characteristics using a high intensity workload [32]; we use Java as implementation language, which is a very popular programming language [33]; we investigate the response to burst workloads, where requests ramp up in short periods (i.e., 1, 3, 6 and 10 seconds); we also followed well-established methodologies conducted by Cooper et. al. [34] to examine the software system at saturation point and to mitigate its behavior.

3 A Methodology for Measuring Idle Timeout in Serverless Infrastructure

In this chapter, we describe our methodology followed by experimental setup to study FaaS' idle timeout. Next, we present the evaluation results and discuss the implications.

3.1 Methodology

In this section, we describe our methodology of measuring function instance idle timeout. We start with presenting how we uniquely identify a function instance in Section 3.1.1, followed by measurement methodology in Section 3.1.2. In Section 3.1.3, we outline how we measure the function instance idle timeout when keep-alive technique is used.

3.1.1 Function Instance Identifier

To measure the idle timeout of a function instance, we measure the longest duration that a container instance can be left idle before it is decommissioned and a new container instance is provisioned. This task is equivalent to identifying if a request is served by an available instance or a newly created instance. As a result, it is essential to develop a mechanism to identify each cloud platform's function instance. We leveraged available information supplied by cloud provider as well as followed previous study [7] to derive function's instance identifier:

- AWS Lambda cloud function instance identifier can be obtained using *logStreamName* which is located in execution context and uniquely tied to the function instance.

- Azure Cloud Function did not provide a direct mechanism to obtain the function instance identifier similar to AWS Lambda. However, using the Monitoring - Log query from the Management Console, we were able to retrieve all the requests' *customDimensions* which contained invocation and executing function instance identifier.

- IBM Cloud Function did not offer a straightforward approach to collect the instance identifier. Hence we followed the mechanism presented by Lloyd

et al. [7] to generate a universally unique identifier (UUID) for each function instance. When a cloud function is invoked, it will first look into the location *"/tmp/host.txt"* for the UUID. If this file does not exist, we know that this is a newly provisioned instance, upon which we generate and store a UUID in this file. The UUID is a 128-bit value and was created using *UUID.randomUUID()* in Java language. We implemented the synchronization lock mechanism while generating and storing the UUID. This concept is similar to the lazy-initialization with double lock method in singleton design pattern implementation. Such implementation makes our methodology thread-safe hence can be used reliably in multi-concurrent load tests. When the *"/tmp/host.txt"* already exists, subsequent invocations can directly access the file and retrieve the UUID without acquiring the lock to avoid performance degradation.

3.1.2 Function Idle Timeout Measurement

To measure a platform FaaS idle timeout, we first deployed our testing function (discussed in Section 3.2.3). Next, we invoked the function periodically with different invocation interval to detect the idle timeout. We started with 20-minute interval because we noticed that function instances would be decommissioned if they were left idle for this duration. Subsequently, we decreased the interval by

one minute, ran the test for five hours and check if the requests were served by the same instance. If all the requests were served by different function instances, we would then proceed to the next reduced interval. The experiments continued until we detected an interval that makes two consecutive requests served by the same instance. After we noted the idle timeout as x minute, we repeated our experiment with x and $(x + 1)$ minute for five hours for each interval to ensure consistent results. To investigate the cloud function idle timeout changes over time, we followed the method introduced by Iosup et al. [35] which investigated for each performance indicator the presence of yearly, monthly, weekly and daily. Similarly, we examined our cloud functions idle timeout over three extended periods [12/09/2020 - 01/10/2020], [27/03/2021 - 13/05/2021] and [19/07/2021 - 27/07/2021] to inspect and detect any implicit changes introduced by cloud providers.

3.1.3 Function Instance Keep-Alive Idle Timeout

Once we detected the function instance idle timeout as x minutes using the methodology in Section 3.1.2, we moved one step further to experiment how long a function instance can be re-used when keep-alive technique is used. To examine this characteristic, we periodically polled the function instance every x minutes and recorded the maximum period a cloud function instance can be re-used. Our experiments were carried out between [19/07/2021 - 27/07/2021].

3.2 Experimental Setup

In this section, we discuss how we setup our cloud functions on different cloud platforms.

3.2.1 Cloud Providers

Based on Eismann et al. [36] FaaS' use case study, we chose to evaluate FaaS on three most popular cloud providers, namely AWS Lambda, Microsoft Azure Function and IBM Cloud Function because they occupied majority of FaaS use cases (80%, 10% and 7% respectively). We did not study other providers like Google Cloud Function, as they only cover small use cases (3%).

3.2.2 Runtime

We focused on Java in this paper because Java and Node.js are known to be the most popular studied language in FaaS software industry research work by Scheuner et al. [33]. Among different versions of Java, we used Java version 8 because it is the latest supported version in IBM Cloud Function whereas AWS Lambda and Azure Functions can support up to Java 11. To ensure consistent results, we selected a version that was commonly supported on all cloud platforms.

3.2.3 Testing Function

We followed Spillner et al. [37] and Manner et al. [17] to setup a testing function which computes the n^{th} value of the Fibonacci (fib) sequence. Our function implementation also used recursive method which is compute bound. The $fib(n)$ is derived by evaluating the $fib(n-1)$ and $fib(n-2)$. To simulate the function processing, we used similar $fib(38)$ as the workload.

For AWS Lambda, we integrated the Lambda function with provided HTTP API Gateway to enable HTTP Representational State Transfer (REST) API communication. IBM and Azure Cloud Functions had this capability supported by default hence no additional implementation was required. Experimental system is shown in Figure. 3.1.

We configured the AWS Lambda and IBM Cloud Function instances to have 512MB memory and 15-second execution timeout as experimental results showed that these configurations were appropriate for the Fibonacci task. Azure Cloud Platform did not have memory configuration option hence we deployed our Azure Fibonacci cloud function on a single Function App running on Linux Operating System with 15-second execution timeout. These deployment settings help to avoid resource-sharing which can interfere with experimental results. In addition, for comparison purpose, we also created a Hello-World cloud function and deployed

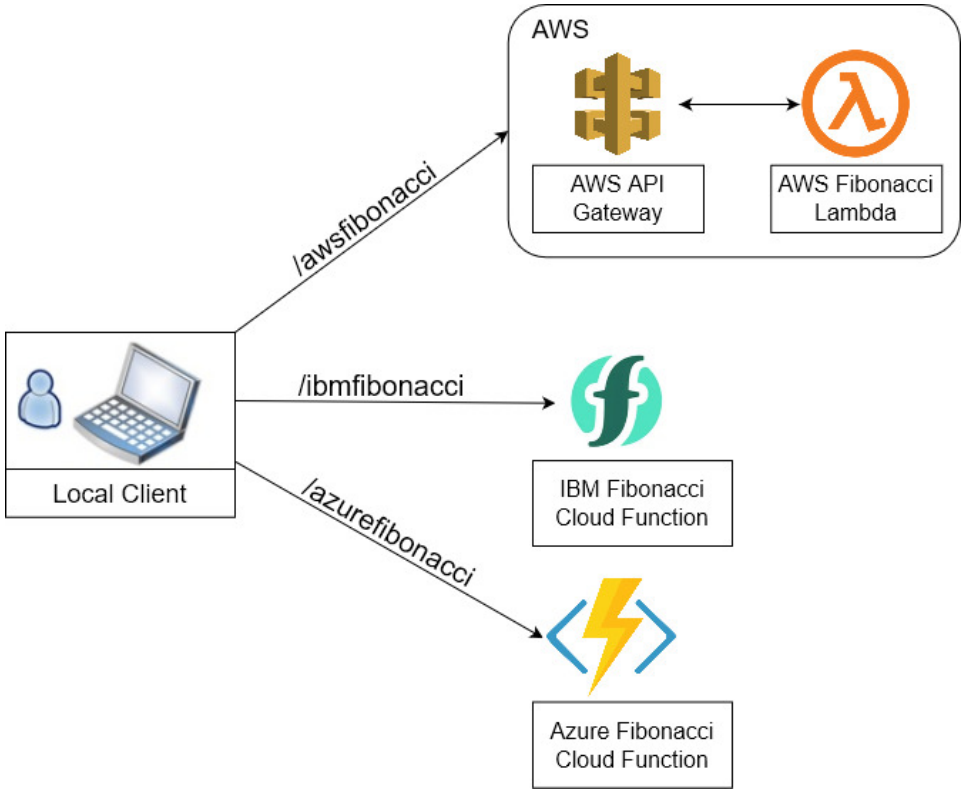

Figure 3.1: Idle Timeout Experimental Setup.

with the same cloud instance configurations. The Hello-World function returns a welcome message upon invoked and processes no business logic. The cold and warm start average response times of the two experimental cloud functions are shown in Table 3.1.

Response Time	**AWS**	**IBM**	**Azure**
Fibonacci (Cold start)	1,161	3,169	2,825
Fibonacci (Warm start)	778	695	628
Hello-World (Cold start)	698	1,495	2,663
Hello-World (Warm start)	79	169	81

Table 3.1: Average Response Time (in ms) during Cold and Warm Start.

3.2.4 Local Client

To measure a cloud function idle timeout with high accuracy, we developed a local Java client program. Our client program has configurable timers which periodically send the HTTPS requests to testing cloud functions. After receiving the cloud function's response, the client will extract the response and save important information to local files for evaluation. The information includes response timestamp, execution duration, cloud function name, request identifier, cloud function instance identifier, the calculated result (i.e., the Fibonacci n^{th} number). The local client program was deployed and run on a Raspberry Pi 4 Model B with 1.5GHz

64-bit Quad-core ARMv8 CPU, 4GB RAM and 32GB Memory storage.

3.3 Evaluation Results

Here we present our experimental results.

Cloud Function Idle Timeout	**AWS**	**IBM**	**Azure**
Cloud Function Idle Timeout (mins)	5	10	12
Maximum Idle Timeout Using Keep-Alive (mins)	145	336	2675
90^{th}-Percentile Idle Timeout Using Keep-Alive (mins)	140	138	1639

Table 3.2: Summary of FaaS Cloud Function Idle Timeout (Undocumented)

3.3.1 Function Instance Idle Timeout

Table 3.2 summarizes our experimental undocumented results for cloud function instance idle timeout, the maximum and 90^{th}-percentile idle timeout when keep-alive technique is used. The results show that an AWS function instance can be idle for at most **5 minutes**. Beyond this duration, AWS reclaims the function instance and subsequent request will experience re-initialization. On the contrary, IBM cloud function instance can stay idle for a maximum duration of **10 minutes**.

Next invocation beyond this period will experience cold start similar to AWS cloud platform. Azure cloud function instance offers the longest idle timeout at **12 minutes**. Beyond this period, Azure cloud platform will provision and re-initialize a new instance for subsequent requests.

Upon completed the cloud function instance timeout on AWS Lambda and IBM Cloud Function with 512MB allocated memory, we re-examined the experiments on these cloud platforms with 1024MB memory to understand if there is a relation between cloud function instance idle timeout and allocated memory. This additional experiment is not applicable to Azure Cloud Function as this platform does not support memory configuration. The results show no difference compared to results in Table 3.2 hence we conclude that there is no relation between a cloud instance idle timeout and allocated memory for AWS Lambda and IBM Cloud Function.

3.3.2 Maximum Function Instance Idle Timeout When Using Keep-Alive Technique

Based on the maximum idle timeout presented above, we configured our local timer clients to poll every 5, 10 and 12 minutes for AWS Lambda, IBM and Azure cloud functions, respectively, to examine how long the function instances can be re-used if they are kept warm.

The results show that an AWS Lambda cloud instance can be retained at most

145 minutes. The 90^{th}-percentile for the duration of service is **140 minutes**. The small gap between the maximum and 90^{th}-percentile implies that this cloud platform adopts a static recycling algorithm and most of the function instance will be recycled after 140 minutes.

In contrast, IBM cloud function instance can serve a periodic traffic continuously at most **336 minutes**. Nevertheless, the 90^{th}-percentile value drops to **138 minutes**. The enormous gap between the maximum and 90^{th}-percentile might infer that IBM cloud function instance is occasionally kept longer than usual, however majority of the function instances will be decommissioned after around 138 minutes.

Microsoft Azure cloud function exhibited an interesting behavior. When tested with the polling period between 6 to 12 minutes, a function instance can be used at most **20 minutes**. However, when we reduced the polling period to 5 minutes, we observed the function infrastructure was retained up to **44 hours 30 minutes**. This behavior may induce that Azure cloud function considers the 5-minute interval as frequent invocation pattern hence the platform keeps the function instance warm to serve the traffic. In contrast, a longer period between 6 to 12 minutes may not be seen as frequent and as a result the function instance is recycled after maximum 20 minutes.

3.3.3 Function Instance Idle Timeout Evolutionary Changes

Evolutionary Cloud Function Idle Timeout Changes	**AWS**	**IBM**	**Azure**
01 - 02/2020 [acc. Maissen]	10	10	20
09 - 10/2020	10	10	14
03 - 05/2021	5	10	12
07/2021			

Table 3.3: Evolutionary Changes of FaaS Cloud Function Idle Timeout (Undocumented)

Applying our methodology at different checkpoints from 09/2020 to 07/2021, we discovered that cloud function's instance idle timeouts have implicitly changed. We first noted the previous study's result reported by Maissen et al. [9] which were established during the [01 - 02/2020] then we conducted three experiments during: [09 - 10/2020], [03 - 05/2021] and [07/2021]. The measurement results summarized in Table 3.3 shows that IBM Cloud Function did not change the function idle timeout since 01/2020 to 07/2021. AWS Lambda, on the contrary, reduced this characteristic by 50% from 10 to 5 minutes some time between 10/2020 and 03/2021. Microsoft Azure cloud platform also shortened the instance's idle timeout by nearly 50% throughout the time. These changes were not officially documented

and communicated to software engineers.

3.4 Implications

FaaS technology provides on-demand initialization and execution. Once processing is completed, cloud function instance is retained for future requests to prevent expensive cold start. The duration for how long cloud function instance is retained without serving traffic varies between 5 to 12 minutes depending on the platform. Beyond this idle timeout, cloud platforms will decommission the instance and return the computing resources for other purposes.

By using keep-alive technique, software engineer can extend the function instance idle timeout up to hours. Nevertheless, all cloud providers will eventually reclaim their cloud function instances even while serving active traffic. We note that the decision to use keep-alive technique depends on the application and traffic pattern. For example, if the application only serves the traffic in a short period of the day, software engineers might not need to adopt keep-alive technique or they may simply apply the keep-alive for certain period. If the traffic pattern is random and there is a big performance differences between cold and warm start like our Hello-World cloud function, it might be helpful for software engineers to trigger keep-alive using regular timed events. This can offer faster response time and reduced operating cost. When there is only a small performance difference between

cold and warm start like our AWS Lambda Fibonacci cloud function, keep-alive technique may not be optimal as there will incur bigger waste in computing resources while minimal saving in operating cost.

Through checking Maissen et al. [9] study and conducting our research study, we observed that some cloud function instance's idle timeouts have evolved over time. To the best of our knowledge, these function idle timeouts have never been discussed by cloud providers. They are only studied and reported by research studies [7] [8] [9] or by industrial web blogs [10] [11] [12]. By reducing the function instance idle timeout, on one hand, it helps cloud providers to save computing resources. On the other hand, this might bring unexpected impact on applications, depending on their traffic patterns. For the cases where function's traffic frequency lies within the idle timeout period, software engineers will not notice any side effect. Nonetheless, in case the traffic is infrequent and beyond the idle timeout limit, the applications will experience more cold starts leading to an increase in overall response time. As a result, it is essential to develop a comprehensive methodology and conduct periodic experiments to report all implicit infrastructure changes to ensure Quality of Service (QoS) for FaaS applications.

3.5 Summary

In this chapter, we described the methodology to measure cloud function instance idle timeout. We conducted further experiments to inspect how long a cloud function instance can be retained if software engineers use keep-alive technique. The comparison shows that cloud function instance idle timeout has changed over time. In the next chapter, we present the methodology to benchmark FaaS' scalability and elasticity using multi-concurrent clients. We also propose a strategy to improve FaaS' overall performance when FaaS has reached its upper concurrency limit.

4 A Methodology for Benchmarking FaaS' Scalability and Elasticity

In this chapter, we present our research questions and methodology followed by experimental setup to benchmark FaaS' scalability and elasticity. Next, we discuss the results, outline future work as well as threats to validity.

4.1 Research Questions and Methodology

In this section, we describe our Research Questions (RQ) and corresponding methodologies.

4.1.1 Research Questions

To assess FaaS' scalability as well as performance improvement, we formulate two research questions (RQs):

- **RQ-1 (Scalability)**: What are the FaaS' scalability and elasticity charac-

teristics under heavy load test scenarios?

In **RQ-1**, we focus on FaaS' scalability and elasticity in handling different workload intensities. Based on the scalability and elasticity defined by Herbst et al. [26] and Kuhlenkamp et al. [38], we chose to examine the FaaS' scalability using multi-concurrent clients. The collected measurements help us to characterize the scaling patterns and performance on each cloud platform.

- **RQ-2 (FaaS Under Saturation and Performance Improvement Patterns)**: What is FaaS' performance under saturation? Does workload smoother pattern help to improve the performance?

  FaaS' scaling capacity is not unlimited as discussed in Section 2.2.1.1. When FaaS' capacity reaches the saturated level (i.e., when the throughput of a system stops increasing [34]), we apply a workload smoother design pattern to verify if this pattern can improve the system overall performance.

4.1.2 Methodology

4.1.2.1 RQ-1 Methodology

We used JMeter client [39] to send multi-concurrent requests to test cloud functions. We note that previous studies experimented with concurrency level at 1, 5, 10, 20, and 30 concurrent requests [40] [31]. To simulate intensive load and performance

experiments, we decided to test the platforms with 100 concurrent clients.

In addition, the workload intensity was also controlled by the following JMeter parameters:

- **Ramp Up Time** refers to how long JMeter takes to create 100 testing threads (i.e., concurrent clients). We noted that this parameter was also used by Somu et al. [41] while benchmarking serverless application. We used four discrete intervals which are one second (1s), three seconds (3s), six seconds (6s) and ten seconds (10s). The shorter the ramp up time, the more intensive the workload is.

- **Total Number of Requests** refers to the size of the examining workload. We designed three workload sizes which were 1000, 3000 and 7000 requests representing low, medium and high levels.

We evaluate FaaS' scalability and elasticity based on the following four metrics which were inferred from the definitions discussed in Section 2.2.1.2:

- The number of function instances spawned versus different ramp up times and workload levels: In this metric, we counted the number of cloud function instances provisioned based on a unique cloud instance identifier (discussed in Section 3.1.1). This metric helps us to characterize the scaling patterns adopted by each cloud platform. For instance, some cloud platforms provision

the number of instances equivalent to the number of concurrent clients. In contrast, other platforms enhance the concurrent capability of each cloud function instance and therefore only a small number of instances is required.

- The duration cloud providers needed to ramp up to expected number of cloud function instances: In this metric, we measured the time it took cloud platforms to scale up to required number of instances. Using the timestamps and unique function instance identifier embedded in returned response, we calculated the ramp up duration as the time difference between the first response and the response which was first served by the lastly added instance. For example, if a test requires cloud platform to provision 100 instances in total, the duration for ramping up would be measured as the time difference between the first response's timestamp and the response's timestamp which was first served by the 100^{th} instance. This metric indicates how quickly a cloud provider can append more instances. The shorter it takes, the better the elasticity is [26].

- The system's overall throughput: System throughput is a commonly used performance benchmark metric. It is measured as the number of requests processed in a unit of time. We calculated the throughput as follow:

$$Throughput = \frac{Total\ Number\ Of\ Request}{Execution\ Time\ In\ Second} * 60 \tag{4.1}$$

We further verified our calculated result with the throughput measured by JMeter to ensure consistency. Throughput shows how many requests the system has served in a unit of time (i.e., minute in our case). The higher the throughput, the better the system performance is.

- The median response time: We measured the median response time which is also a commonly used performance metric. The median response time shows the duration a system needs to produce the result. Different from average response time, median response time is known to be less sensitive to short-term fluctuations such as cold-start or unexpected timeout [42]. The lower the response time is, the better the performance delivers.

The experiments were conducted over extended period of April 5^{th} - 12^{th}, 2021.

4.1.2.2 RQ-2 Methodology

As discussed in Section 2.2.1.1, all FaaS platforms impose upper concurrency limit. Hence, in RQ-2, we aim to study when the workload reaches the maximum concurrency limit and what the system's behavior is at that point. We also study if a workload smoother design pattern can help to improve the system's performance. Among proposed design patterns [28] [29], we choose to implement a workload smoother based on the *Queue-Based Load Leveling Pattern* because this pattern

is effective when the target system intermittently experiences high load. By introducing a request queue, this pattern helps to store excessive requests for later processing instead of immediately returning failure response.

We followed the strategy discussed by Cooper et al. [34] about measuring the response time as throughput increases until the point at which it stops increasing, i.e., the system is saturated. We first examined the FaaS system without the workload smoother. When the throughput reached a consistent level, we stopped the experiment and recorded the total number of requests that were serviced successfully and those that were not. Next, we sent the same total number of requests to a system with the workload smoother and recorded similar performance metrics. This methodology helps us to evaluate if a workload smoother can improve FaaS' overall performance and by what degree.

Based on similar concurrency limit discussion, we decided not to experiment our workload smoother on IBM Cloud Function because a concurrency level of 1,000 with no reducing option is beyond the scope of this study. Therefore, we only experimented our workload smoother on AWS Lambda and Azure Cloud Function. We set up and configured AWS Lambda cloud function to have a maximum concurrent executions at 100 using *reserved concurrency* option. To prevent overloading, we configured the workload smoother so that there were at most 100 concurrent requests sending to AWS Lambda functions. JMeter client was configured to test

the system with 150, 200 and 250 threads with 10-second ramp up time. These workloads were equivalent to 1.5x, 2x and 2.5x times of the FaaS' capacity and thus would help us to assess the system in a saturated scenario. Microsoft Azure cloud function operates slightly different from AWS Lambda where each Azure function instance can serve multiple concurrent requests. To ensure the same experimental methodology across cloud providers, we set Azure function instance to only serve at most one concurrent request. Experiments showed that although Azure cloud platform claims the number of function instances can scale up to 200, in all our RQ-2 experiments we only observed about four instances spawned. As a result, we configured our workload smoother such that it would send at most five concurrent requests to Azure cloud function and setup JMeter to create five, seven and ten threads (equivalent to 1.25x, 1.75x and 2.5x of FaaS' capacity) ramped in 50 seconds.

We conducted experiments for each cloud provider twice to avoid cloud instability. AWS Lambda systems were assessed on May-5$^{\text{th}}$ and May-25$^{\text{th}}$, 2021. Azure cloud platform was evaluated on May-28$^{\text{th}}$ and June-5$^{\text{th}}$, 2021. In these experiments, we measured performance metrics, including median response time, coefficient of variation (CV) which is defined as a ratio in percentage between standard deviation over mean response time, the number of passed, failed, total requests, success rate which is the ratio in percentage between the number of passed requests over

the total number of requests, throughput (request per minute) and the number of instances provisioned.

4.2 Experimental Setup

4.2.0.1 Cloud Providers

Based on Eismann et al. [36] FaaS' use case study, we chose to evaluate on three most popular cloud providers, namely AWS Lambda (AWS), IBM Cloud Function (IBM) and Microsoft Azure Function (Azure) because they occupied majority of FaaS use cases (80%, 10% and 7% respectively). Google Cloud Function [43] although supports FaaS however, only accounted for a small use case percentage (3%) and therefore not generalized enough.

4.2.0.2 Runtime

We focused on Java in this paper because Java and Node.js are known to be the most popular studied language in FaaS software industry research according to Scheuner et al. [33]. Among different versions of Java, we used Java version 8 since it is the latest supported version in IBM Cloud Function whereas AWS Lambda and Azure Functions can support up to Java 11. To ensure consistency, we selected a commonly supported version on all cloud platforms.

4.2.0.3 Tools

To manage our project's dependencies, we used Apache Maven (version 3.6.2) [44]. Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central place.

We used Apache JMeter (version 5.2.1) [39] to conduct multi-concurrent client load and performance tests. JMeter is a pure Java application designed to load test functional behavior and measure performance. It was originally designed for testing Web Applications but has since expanded to other test functions.

While load testing the cloud functions, we profiled how many threads were spawned and run by JMeter using VisualVM tool [45]. VisualVM is a visual tool which offers great profiling capabilities. With VisualVM, we could confirm that JMeter was operating on expected number of threads.

4.2.0.4 Testing Function

We developed a CPU-intensive cloud function which calculates the factorial of a given number similar to Somu et al. [41] experiments. However, our cloud function is a single function compared to their multiple chained cloud functions. We followed the "FaaS Principles and Best Practices" suggested by IBM and Microsoft Azure

cloud providers which recommend that each cloud function should perform only one action and not to make functions call to other cloud functions [46] [47]. We noted that Amazon AWS Lambda also advises not to perform workflow orchestrations within Lambda functions because this orchestration makes the workflow fragile and difficult to modify. Furthermore, the chained pattern also results in increasing cost of execution since each cloud function has to wait for the results of another cloud function [48]. A better approach would be applying AWS Step Functions which is a low-code visual workflow service used to orchestrate AWS services including Lambda functions [49]. Nevertheless, the corresponding product from Microsoft Azure - Durable Function [50] did not support Java programming language hence we could not experiment a more complicated testing function using multiple-function workflow use case. Our logic implementation was designed to compute factorial value using loop iteration. There was no caching to avoid Garbage Collection which might impact the performance. We chose 4000 as the given number to testing function as this number is large enough to simulate intensive processing. Our source code is published for reproducing purpose [51].

With AWS Lambda and IBM cloud function, we allocated 512MB memory since this is appropriate for the factorial calculation. On Microsoft Azure, we deployed the function on a standalone function app which ran on Linux Operating System and did not share resource to any other functions. To experiment the cloud

functions with JMeter, we converted all testing cloud functions to have HTTPS endpoints. IBM and Azure cloud platforms supported this capability by default hence no additional effort was required. For AWS Lambda, we used AWS API Gateway to expose HTTPS endpoint for Lambda function. Fig. 4.1 illustrates our **RQ-1** experiments.

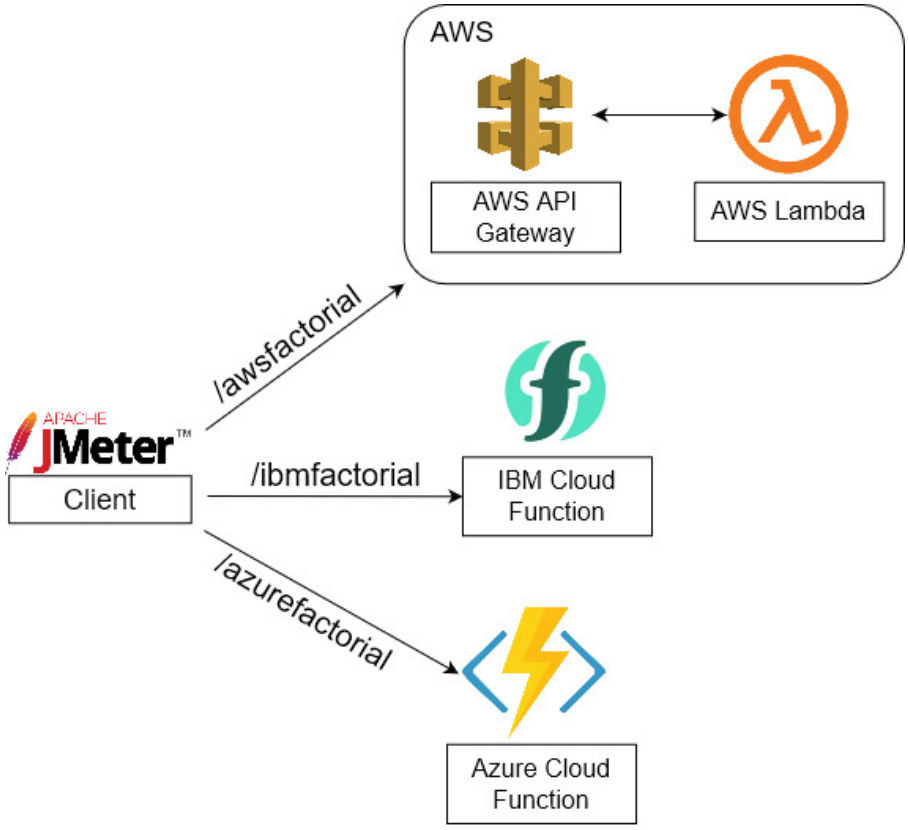

Figure 4.1: Load and Performance Experiment Benchmark With JMeter On Three Cloud Platforms

For **RQ-2**, we re-used the **RQ-1** structure as depicted in Fig. 4.1 to exper-

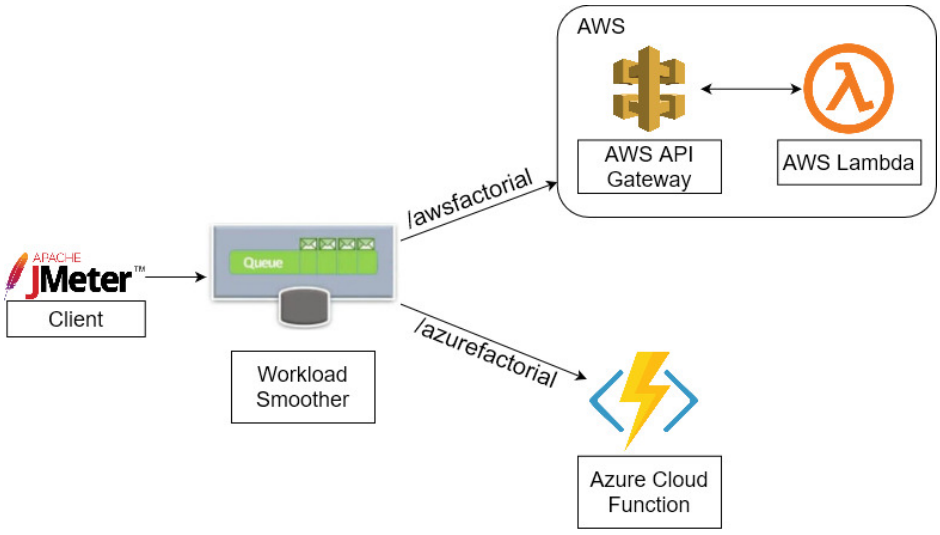

Figure 4.2: Cloud Functions With Workload Smoother

iment a system without a workload smoother introduced. Next, we developed a workload smoother as shown in Fig. 4.2. We implemented the workload smoother as an application powered by Spring Boot Technology [52]. Internally, the workload smoother had two fixed-size thread pool executors which execute the submitted tasks using one of the available threads in the pool [53]. Each thread pool executor corresponded to one target FaaS system, i.e., AWS Lambda and Azure Cloud Functions. We configured each thread pool size aligned with the maximum concurrent capacity that each FaaS system could handle. Furthermore, each thread pool has implicit unbounded queue which automatically en-queues the request when all the threads are busy. Once a thread becomes available, the requests will be de-queued and forwarded to the FaaS system. The workload smoother also exposed

two corresponding HTTPS endpoints for AWS Lambda and Azure cloud function invocation and was deployed on U.S-East region (i.e., Ohio) EC2 t2.large instance (2 vCPU and 8GB Memory). We used t2.large instance because this instance type can sustain high CPU and network requirements hence eliminate potential performance bottleneck. It should be noted that deploying our workload smoother on Amazon AWS EC2 environment did not introduce performance bias to AWS Lambda because each service in the system was built as a standalone component and communicated to via HTTPS APIs.

4.3 Evaluation Results

We describe the experimental results with respect to the research questions **RQ-1** (Section 4.3.1) and **RQ-2** (Section 4.3.2).

4.3.1 RQ-1: What are the FaaS' scalability and elasticity characteristics under different load test scenarios?

In this section, we present the scalability and elasticity characteristics of FaaS under different types of workload according to the four discussed metrics.

4.3.1.1 The Number of Function Instances Spawned Versus Different Ramp Up Time and Workload Levels

Number of Request	Ramp Up Time											
	AWS				IBM				Azure			
	1s	3s	6s	10s	1s	3s	6s	10s	1s	3s	6s	10s
1000	99	82	54	29	100	108	101	109	4	4	3	4
3000	100	86	67	43	107	108	112	113	5	6	5	5
7000	97	88	70	66	113	105	120	123	7	7	7	7

Table 4.1: Number of Instances spawned by different Ramp Up Time and Workload Levels

Table 4.1 records the number of instances spawned by three cloud providers across different ramp up times and workload levels. For AWS Lambda, when the workload became more intensive by reducing the ramp up time, the number of instances spawned increased accordingly. This pattern was consistent across three experimental workload levels. In addition, when the workload level expanded from 1000 to 7000 requests, AWS Lambda exhibited two different patterns. For long ramp up time (6s and 10s), the number of instances increased in accordance with the increase in workload (54 - 70, 29 - 66 instances respectively). However, for

short ramp up time (1s and 3s), the number of instances only changed by a small number (99 - 97, 82 - 88 instances, respectively).

On the contrary, IBM cloud platform shows a relatively similar number of 100 to 120 instances spawned across different ramp up times. There was no clear pattern in the relationship between changing the ramp up time and more instances getting provisioned. Nevertheless, when workload levels extended from 1000 to 7000, the number of instances spawned increased accordingly in most cases (three out of four cases except for the ramp up time of 3s).

Microsoft Azure cloud function shows that the number of instances spawned was almost constant across different ramp up times. When the workload level increased from 1000 to 7000, we noticed that Azure cloud platform nearly doubled the number of instances from four to seven to process the workload.

For this metric, we observed that IBM Cloud Platform scaled their instance fleet to around 100 - 120 instances in all testing configurations. These values were quite close to 100 concurrent clients which leads us to the conclusion that IBM cloud platform provisioned the number of instance similar to the number of concurrent clients. AWS Lambda exhibited similar characteristic only when the workload was intensive (i.e., 1 second ramp up time in all workload sizes). For other ramp up times, the number of instances were less than the number of concurrent clients and the lower the intensity was, the lesser the number of instances spawned. Azure

Cloud Function scaled their fleet to only a small number of four to seven instances, this aligns with Microsoft Azure's claim that each of their cloud function instances can process multiple concurrent requests hence a small number of instances is adequate [24].

4.3.1.2 The Duration Cloud Providers Needed To Ramp To Expected Number of Cloud Function Instance

Number of Requests	**Ramp Up Time**											
	AWS				**IBM**				**Azure**			
	1s	3s	6s	10s	1s	3s	6s	10s	1s	3s	6s	10s
1000	0.90	3.33	3.73	3.73	0.037	1.05	2.16	9.02	32.34	36.20	25.29	30.74
3000	0.65	2.64	6.12	6.79	0.12	8.33	17.06	10.36	43.30	51.58	43.21	42.06
7000	0.72	2.60	7.92	10.94	26.59	15.36	24.66	33.32	68.04	63.02	62.69	62.35

Table 4.2: Ramp Duration To Expected Number of Instances (in seconds)

Table 4.2 shows the duration cloud providers ramped their fleet to expected number of instances. For AWS Lambda, we noted that when the ramp up time reduced, the time taken to increase the fleet decreased three to 15 times (3.73s vs 0.9s, 10.94s vs 0.72s). This pattern was observed on all workload levels. In the best scenario, we recorded that AWS Lambda could scale their fleet to 100 instances within 0.65s or on average, one new instance was added in 6.5ms.

Similar to AWS Lambda, IBM cloud platform slowly provisioned and deployed

the function instances when the ramp up time was long (6s and 10s). However, when this duration was shorter (1s and 3s), IBM cloud platform increased the instances 84 to 243 times faster (9.02s vs 0.037s, 10.36s vs 0.12s). In one best scenario, we observed this platform ramped 100 instances in just 37ms. Nevertheless, we also noticed that the IBM performance was inconsistent. In 1s-ramp up time and 7000-request level, it took IBM more than 26 seconds to increase the fleet capacity. We hypothesize that this might be due to the VM cold provisioning which takes more time to provision a new instance [7].

Different from AWS Lambda and IBM Cloud Function, Azure cloud platform required 30 - 60 seconds to deploy more instances to the optimal level. This might be attributed to the characteristic that each Azure instance can process multiple requests concurrently hence Azure cloud platform only considers to add extra instances at a later stage of the test.

For this metric, we noted AWS Lambda and IBM cloud function demonstrated similar behavior. When the workload arrives at intensive pace, these platforms quickly provisioned a number of cloud function instances in a short time period which shows good elasticity. Azure cloud function operated differently in which each instance could serve multiple concurrent requests and hence the duration to scale up the fleet was longer.

4.3.1.3 The Throughput of the System

Number of Request	Ramp Up Time											
	AWS				IBM				Azure			
	1s	3s	6s	10s	1s	3s	6s	10s	1s	3s	6s	10s
1000	14.65	10.87	8.41	5.58	4.19	4.47	4.06	1.35	1.68	1.37	1.74	1.94
3000	20.26	25.56	18.04	13.35	2.85	3.75	7.05	5.71	3.17	3.23	3.74	3.84
7000	34.29	34.44	27.30	19.59	13.34	12.81	14.60	9.81	5.67	5.84	5.87	6.09

Table 4.3: Cloud Function Throughput By Different Ramp Up Time and Workload Levels (thousand requests per minute)

Table 4.3 presents the throughput measured in requests per minute on all cloud platforms. We observed that AWS Lambda produced a consistent throughput increase when the workload became more intensive (i.e., ramp up time was shorter and more requests were sent). This pattern demonstrates a good performance since more resources added to the system should result in more requests processed and consequently increased the system's throughput.

IBM cloud platform on the contrary, exhibited an in-consistent throughput changing pattern. The results fluctuated because in some cases, we noticed a number of requests returned after 30 - 35 seconds. We hypothesize that these calls might be served by newly provisioned Virtual Machine (VM) and it would require more time to complete. As a result, the overall execution duration increased and further reduced the system's throughput.

Azure cloud function showed a consistent throughput across different ramp up times. When the workload level increased from 1000 to 7000, more cloud function instances were added to the system leading to a corresponding throughput boost.

4.3.1.4 The Median Response Time

Number of Request	**Ramp Up Time**											
	AWS				**IBM**				**Azure**			
	1s	3s	6s	10s	1s	3s	6s	10s	1s	3s	6s	10s
1000	219	204	180	113	325	355	415	376	2,644	2,561	2,926	2,878
3000	125	116	98	88	384	380	360	354	1,346	1,241	1,165	1,125
7000	97	89	90	98	303	296	318	320	768	754	773	754

Table 4.4: FaaS Median Response Time By Different Ramp Up Time and Workload Levels (ms)

Table 4.4 displays the median response time recorded on all testing platforms. For AWS Lambda, at low and medium level workload (i.e., 1000, 3000-request), when the ramp up time decreased, the response time increased around 42% - 93% (125ms vs 88ms, 219ms vs 113ms). Nonetheless, when the workload level was high (i.e., 7000-request), there was no major difference between the response times. This result might be due to the cold-start impact, where FaaS system needs to provision and initialize the function instances before execution. This activity introduces

additional delay in response time for early requests (i.e., cold-start requests), and subsequent requests are processed faster (i.e., warm-start requests). Among the three workload sizes, the 7000-request had more warm-start requests than the other two levels. As a result, the median response time of this workload level was less susceptible to cold-start response time and hence delivered stable median response time.

IBM cloud platform produced a consistent median response time across all the experiments. This could be due to the fact that IBM cloud platform provisioned and deployed a similar number of instances in all testing scenarios and therefore the workload was equally distributed, thus producing stable results.

Microsoft Azure cloud function exhibited a consistent median response time across different ramp up times. Nevertheless, when the workload level increased, the median response time was greatly improved with a reduction from 2,600ms to 1,200ms then 760ms. This improvement pattern was similar to AWS Lambda mentioned above.

Summary of RQ-1

Through RQ-1 experiments, we conclude that all three cloud platforms provide good horizontal scalability and elasticity to address different types of workloads and intensity levels.

AWS Lambda produced stable results and consistent behavior. When the workload became more intensive, AWS Lambda automatically increased the number of instances to address the surge and maintained performance.

IBM cloud platform was observed to ramp their instance fleet to similar number of instances over all testing configurations. Occasionally, IBM cloud function suffered long cold-starts which reduced the system overall performance.

Microsoft Azure cloud function showed that their cloud function instance could handle multiple concurrent requests by provisioning less number of instances compared to the other cloud providers. Throughput and median response time of this cloud provider were stable and consistent across different ramp up times and workload levels.

4.3.2 RQ-2: What is FaaS' performance under saturation? Does workload smoother pattern help to improve the performance?

Here we present the FaaS' performance under saturation and the advantages achieved by introducing a workload smoother.

Metrics	1.5x Capacity		2x Capacity		2.5x Capacity	
	Direct	**WLSM**	**Direct**	**WLSM**	**Direct**	**WLSM**
Median Resp. Time (ms)	80	201	84	268	103	372
Coefficient of Variation	49.60%	56.03%	51.41%	47.50%	62.69%	39.56%
Throughput (req./min)	96,036	37,088	118,873	38,706	97,707	36,378
Number of Instances	100	95	101	98	101	96
Pass	349,206	351,600	364,624	380,400	298,005	371,750
Fail (HTTP 429)	2,288	0	15,702	0	73,558	0
Total Requests	351,494	351,600	380,326	380,400	371,563	371,750
Success Rate	99.35%	100.00%	95.87%	100.00%	80.20%	100.00%

Table 4.5: Performance Comparison on AWS Lambda FaaS without (Direct) and with Workload Smoother (WLSM).

4.3.2.1 AWS Lambda with Workload Smoother

Table 4.5 shows comparison results between direct invocation to AWS Lambda (Direct) and through a workload smoother (WLSM). In all experiments, a direct invocation to AWS Lambda resulted in better median response time. In general, it took the cloud function 80 - 103ms to produce the calculation result. When there were more concurrent clients added, response time tended to fluctuate more with CV increased from 49% to 63%. AWS Lambda provisioned about 100 - 101 instances to serve the workload as expected when we configured *reserve concurrency*

to 100. The system overall throughput could reach 118,000 requests per minute. Nevertheless, a large number of requests were throttled with "Too Many Request" error in 2x- and 2.5x-capacity configuration (15,000 and 73,000 requests, respectively), these results demonstrated that AWS Lambda functions were extensively overloaded and the success rates could only reach about 95% and 80% correspondingly.

In contrast, by having a workload smoother, the system could achieve 100% success rates in all scenarios. Excessive requests were queued and later de-queued for processing hence no request was throttled with "Too Many Request" error. However, since the requests had to pass through one additional component and potentially stayed in the queue, certain performance metrics were lower compared to direct invocation. In particular, it took a request 200 - 370ms to complete. Although it was much longer to process a request, the system response times were less fluctuated with CV reduced from 56% to 48% then 40%. There were 96 instances spawned by cloud platform which was slightly less than the 100 concurrent client configured in the workload smoother. This gap might be attributed to workload intensity reduction by introducing the workload smoother.

Metrics	1.25x Capacity		1.75x Capacity		2.5x Capacity	
	Direct	**WLSM**	**Direct**	**WLSM**	**Direct**	**WLSM**
Median Resp. Time (ms)	84	151	82	154	85	187
Coefficient of Variation	59.22%	34.42%	53.98%	33.97%	156.61%	24.36%
Throughput (req./min)	2,942	1,755	3,866	2,413	3,843	2,921
Number of Instances	4	4	4	4	4	4
Pass	15,167	15,651	22,174	26,444	15,543	25,314
Fail (HTTP 429)	545	64	4,473	205	9,929	166
Total Requests	15,712	15,715	26,647	26,649	25,472	25,480
Success Rate	96.53%	99.59%	83.21%	99.23%	61.02%	99.35%

Table 4.6: Performance Comparison on Azure FaaS Cloud Function without (Direct) and with Workload Smoother (WLSM).

4.3.2.2 Azure Function with Workload Smoother

Table 4.6 shows the performance comparison of Azure cloud functions when workload smoother was introduced. Similar to AWS Lambda function, direct invocation on Azure cloud function produced better median response time. On average, the task was completed in 80 - 85ms. Nonetheless, the CV values increased from 59% to 156% which shows major response time fluctuation. In all experiments, we observed Azure cloud platform only spawned four instances regardless of workload intensity. The throughput of the Azure cloud functions were between 3,000 - 3,800

requests per minute. The success rates deteriorated when the workload became more intensive, 96% - 83% - 61% for 1.25x, 1.75x and 2.5x-capacity, respectively. Failure requests were returned with "Too Many Request" error similar to AWS Lambda. Considering that there were only four instances spawned (which was far below the 200-instance capacity threshold claimed by Azure) and a large number of requests were throttled, this raised a concern about the Azure cloud platform's auto-scaling algorithm. Cloud provider did not add more instances to address the workload but throttled the excessive requests while the number of instances had not reached the maximum level.

On the contrary, when workload smoother was added, the system achieved more than 99% success rates in all three experiments. The 1% failure might be due to the mismatch between the number of threads in workload smoother and the number of instances provisioned. We configured five working threads in the workload smoother while Azure cloud platform only spawned four instances. As a result, a small number of 60 - 200 requests were returned with "Too Many Request" error. As a trade-off for improving the success rates, median response time was almost doubled compared to direct invocation, one request was processed in 150 - 187ms. Nevertheless, the CV in all three experiments were reduced which means response time were relatively consistent (34% - 33% - 24%). Throughput was largely reduced in between 1,700 to 3,000 requests per minute.

Summary of RQ-2

RQ-2 experimental results show that when the FaaS cloud function system is saturated, excessive requests will be throttled as failure causing low success rates. To address this issue, we can add a workload smoother to queue these excessive requests and thus improve the system's success rate. The more intensive the workload is, the higher the success rate can be achieved. Nonetheless, since there is one additional component and the requests may need to be queued before service, other performance metrics such as median response time and throughput will be impacted.

4.4 Discussion

In this section, we discuss about the findings from the two research questions.

- *FaaS' Scalability and Elasticity*: All testing FaaS platforms provide good auto-resource management with different strategies and therefore, software engineers should analyze and select the appropriate cloud platform according to their requirements. It is also important to note the upper concurrent limit enforced by each cloud provider to avoid throttling issue.

- *Improving the Performance of FaaS*: We observe that when FaaS system

is saturated, excessive requests will be throttled with "Too Many Request" error. This behavior shows that testing FaaS platforms incorporated the *Throttling Design Pattern* where the system returns error message if it can not handle the workload due to shortage of computing resource. By introducing a workload smoother which implements the *Queue-Based Load Leveling Pattern*, we can prevent throttling, thus ensuring high success rate. The corresponding trade-off is the increase in response time and reduced overall throughput. As a result, this pattern should only be applied if the system intermittently experiences high loads. Conversely, if the system frequently encounters intensive workload, it is better to re-perform capacity planning to prevent shortage of resource. Another pattern software engineers might consider is *Priority Queue Pattern* which instead of throttling all excessive requests indiscriminately, the system can serve the requests from high-priority clients and return errors for low-priority ones. Although the throttling issue still happens but the severity will be mitigated.

4.5 Threats To Validity

In this section, we outline the potential internal, external and construct threats that might affect the correctness of our results.

4.5.1 Internal Validity

Our results may be impacted by the cloud platform's memory configuration. AWS Lambda is known to allocate CPU power proportionally to allocated memory and hence a higher memory configuration may yield better results. Furthermore, since each platform adopts different scaling pattern (e.g., one Azure cloud function instance can handle multiple concurrent requests), we have tried to keep all configurations from RQ-1 to RQ-2 the same except for Azure cloud function tests in RQ-2. The reduction of our workload smoother internal threads (i.e., 100 threads for AWS and five threads for Azure) was not a bias to Azure cloud function but instead was the operational configuration for each platform. Our intention was not to compare AWS and Azure platforms but to inspect FaaS' system performance when a workload smoother was introduced.

4.5.2 External Validity

Our experiments were conducted on free-tier instances provided by all cloud providers. Hence the results and findings may change depending on other subscriptions such as Dedicated or Enterprise. These subscription packages may maintain a pool of warm instances to serve the requests immediately, and therefore potentially improve the experimental results. In addition, our findings may also not apply to other cloud

platform such as Google Cloud Functions. Further research needs to be done to characterize the behavior for these platforms.

The testing experiments were carried out on platforms mostly run on Linux operating system hence the findings may not be the same if cloud function instance runs on Windows (in case of Microsoft Azure cloud function). Finally, we have developed our testing cloud function based on the recommended guidelines from cloud providers, however there can be still other scenarios where FaaS may be implemented differently and thus our findings may not be applicable for these use cases.

4.5.3 Construct Validity

Our cloud functions were all deployed on US-East region to minimize the distant network connection. However there may be intermittent and unexpected delay introduced at the client's or server's side which affects the measurements. We conducted our experiments twice to average and mitigate these delays.

4.6 Summary

In this chapter, we discussed the methodology to benchmark FaaS' scalability and elasticity. All FaaS platforms offer auto-scaling feature to address workload variations although each of them adopts different scaling pattern. Furthermore, we

also presented a strategy to improve FaaS' performance when it has reached its upper concurrency capacity. By introducing a workload smoother which implements *Queue-based Load Leveling Pattern*, software engineer can improve the success rate in trade-off for some performance metrics such as response time and throughput. We also outlined other cloud design patterns which can be used to mitigate the performance issues.

5 Conclusion and Future Work

In this chapter, we conclude our research work with highlighted remarks and present the future work.

5.1 Conclusion

The work in this thesis can be summarized as follow:

- We performed FaaS' idle timeout characterization study on popular cloud platforms. Our methodology suggests to use periodic invocation with reducing interval to determine the idle timeout of a cloud function. Each cloud function instance can be uniquely identified by using provided information or UUID created when the cloud function instance is first invoked. The results show that all cloud platforms decommission their FaaS cloud function instances after a period of 5 - 12 minutes.

- To retain the FaaS' infrastructure, a common workaround is keep-alive technique. We applied this technique and measured the maximum duration a

cloud function instance can be kept. By regularly polling the cloud function before it is scheduled for decommission, software engineers can keep the cloud function instance alive up to hours. However, eventually, all cloud platforms reclaim their computing resources regardless of active traffic.

- Comparing the FaaS' idle timeout at different checkpoints since 01/2020 till 07/2021, we observed that FaaS' idle timeout has been reduced by cloud providers. We noted that these evolutionary changes are neither transparent nor communicated to software engineers. Consequently, software system might observe performance degradation without showing the root cause of the problem. We suggested that it is essential to present a thorough methodology to regularly examine and report cloud instance idle timeout for public's benefits.

- We benchmarked FaaS' scalability and elasticity using well-known metrics such as: the number of instance spawned, the duration for ramping up, median response time and throughput. These metrics help us to discover the scaling pattern adopted by each cloud provider. Some cloud platforms scale the number of instances equivalent to the number of concurrent clients whereas other platforms enhance each instance's concurrency capability so there is only a small number of instances required. In general, all FaaS platforms offer

horizontal auto-scaling feature to address workload variations hence software engineer can consider to migrate to FaaS if they need this capability.

- We examined FaaS' performance when it has reached the upper concurrency limits. We proposed to use a workload smoother which implements *Queue-Based Load Leveling Pattern* to queue the excessive requests and smooth the workload. Experimental results showed that when FaaS' system is saturated, excessive requests will be throttled as failure and hence reduce the success rates. By introducing a workload smoother, software engineer can prevent the throttling issue. Overloaded requests are queued for later processing thus resulting in 20 - 40% improvement in success rates.

- We discussed the cloud design patterns that FaaS has adopted and further outline other patterns which software engineers can consider to apply.

5.2 Future Work

In future research, we plan to study FaaS with actual use cases. First, we plan to report the recommended/ideal use cases to develop a software system using FaaS to achieve low cost and high performance. Second, at the time of study, multiple FaaS functions combined in a workflow is only available in AWS Lambda with Java programming language (i.e., AWS Step Function). Azure cloud platform (i.e.,

Azure Durable Function) does not support this capability with Java. Hence, in future, we would like to expand our study with this type of functions to present a more generalized view. Finally, we also plan to improve the performance and robustness of FaaS by investigating the use of additional design patterns proposed in the traditional or cloud computing platforms.

Bibliography

[1] Use containers to build, share and run your applications. https://www.docker.com/resources/what-container. Last accessed: 09/22/2021.

[2] Amazon web services announces aws lambda. https://press.aboutamazon.com/news-releases/news-release-details/amazon-web-services-announces-aws-lambda. Last accessed: 09/08/2021.

[3] Building applications with serverless architectures. https://aws.amazon.com/lambda/serverless-architectures-learn-more/. Last accessed: 09/08/2021.

[4] Understanding container reuse in aws lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/. Last accessed: 09/08/2021.

[5] How cloud functions works. https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-about. Last accessed: 09/08/2021.

[6] Lambda function scaling. https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html. Last accessed: 09/13/2021.

[7] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In 2018 IEEE International Conference on Cloud Engineering (IC2E), pages 159–169, 2018.

[8] Wes Lloyd, Minh Vu, Baojia Zhang, Olaf David, and George Leavesley. Improving application migration to serverless computing platforms: Latency mitigation with keep-alive workloads. In 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pages 195–200, 2018.

[9] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. Faasdom. Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems, Jul 2020.

[10] Til that aws lambda terminates instances preemptively. https://xebia.com/blog/til-that-aws-lambda-terminates-instances-preemptively/. Last accessed: 09/03/2021.

[11] Aws lambda cold starts after 10 minutes. https://mikhail.io/2019/08/aws-lambda-cold-starts-after-10-minutes/. Last accessed: 09/03/2021.

[12] Cold start / warm start with aws lambda. https://blog.octo.com/en/cold-start-warm-start-with-aws-lambda/. Last accessed: 09/03/2021.

[13] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, and Maciej Malawski. Performance evaluation of heterogeneous cloud functions. Concurrency and Computation: Practice and Experience, 30(23):e4792, 2018.

[14] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. Evaluation of production serverless computing environments. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pages 442–450, 2018.

[15] Timon Back and Vasilios Andrikopoulos. Using a microbenchmark to compare function as a service solutions. In Service-Oriented and Cloud Computing, pages 146–160. Springer International Publishing, 2018.

[16] Microservice catalogue. https://microservices.io/patterns/microservices.html. Last accessed: 09/09/2021.

[17] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. Cold start influencing factors in function as a service. In 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pages 181–188, 2018.

[18] Cloud function idle timeout implementation. https://doi.org/10.5281/zenodo.5722868. Last accessed: 12/01/2021.

[19] Configuring function memory. https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html. Last accessed: 09/09/2021.

[20] Azure functions hosting options. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale. Last accessed: 09/09/2021.

[21] Ibm cloud function doubling time limit executions. https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-limits. Last accessed: 09/09/2021.

[22] Aws lambda quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html. Last accessed: 09/09/2021.

[23] Aws reserved concurrency. https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html. Last accessed: 09/30/2021.

[24] Azure functions http output bindings. https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook-output#hostjson-settings. Last accessed: 09/09/2021.

[25] Michael Kuperberg, Nikolas Herbst, Joakim Kistowski, and Ralf H. Reussner. Defining and quantifying elasticity of resources in cloud computing and scalable platforms. 2011.

[26] Nikolas Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. International Conference on Autonomic Computing, pages 23–27, 01 2013.

[27] Sebastian Lehrig, Hendrik Eikerling, and Steffen Becker. Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics. In 2015 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA), pages 83–92, 2015.

[28] Davide Taibi, Nabil El Ioini, Claus Pahl, and Jan Niederkofler. Patterns for serverless functions (function-as-a-service): A multivocal literature review. In Proceedings of the 10th International Conference on Cloud Computing and Services Science - CLOSER,, 05 2020.

[29] Cloud design patterns. https://docs.microsoft.com/en-us/azure/architecture/patterns/. Last accessed: 09/30/2021.

[30] 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, page 1576–1585, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Horácio Martins, Filipe Araujo, and Paulo Rupino Cunha. Benchmarking serverless computing platforms. Journal of Grid Computing, 18:691–709, 12 2020.

[32] Uma Tadakamalla and Daniel Menascé. Characterization of IoT Workloads, pages 1–15. Springer International Publishing, 06 2019.

[33] Joel Scheuner and Philipp Leitner. Function-as-a-service performance evaluation: A multivocal literature review. Journal of Systems and Software, 170:110708, 2020.

[34] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In Proceedings of the 1st ACM Symposium on Cloud Computing, page 143–154, 2010.

[35] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. On the performance variability of production cloud services. In 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pages 104–113, 2011.

[36] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. A review of serverless use cases and their characteristics, 2021.

[37] Josef Spillner, Cristian Mateos, and David A. Monge. Faaster, better, cheaper: The prospect of serverless scientific computing and hpc. In Esteban Mocskos and Sergio Nesmachnow, editors, High Performance Computing, pages 154–168, Cham, 2018. Springer International Publishing.

[38] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. Benchmarking scalability and elasticity of distributed database systems. Proc. VLDB Endow., 7(12):1219–1230, 2014.

[39] Apache jmeter. https://jmeter.apache.org/. Last accessed: 09/26/2021.

[40] Andrei Palade, Aqeel Kazmi, and Siobhán Clarke. An evaluation of open source serverless computing frameworks support at the edge. In 2019 IEEE World Congress on Services (SERVICES), volume 2642-939X, pages 206–211, 2019.

[41] Nikhila Somu, Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Panopticon: A comprehensive benchmarking tool for serverless applications. In 2020 International Conference on COMmunication Systems NETworkS (COMSNETS), pages 144–151, 2020.

[42] Zhen Ming Jiang and Ahmed E. Hassan. A survey on load testing of large-scale software systems. IEEE Transactions on Software Engineering, 41(11):1091–1118, 2015.

[43] Google cloud functions. https://cloud.google.com/functions. Last accessed: 09/26/2021.

[44] Apache maven. https://maven.apache.org/. Last accessed: 09/26/2021.

[45] Visualvm tool. https://visualvm.github.io/. Last accessed: 09/26/2021.

[46] Faas principles and best practices. https://www.ibm.com/cloud/learn/faas. Last accessed: 09/12/2021.

[47] Best practices for performance and reliability of azure functions. https://docs.microsoft.com/en-us/azure/azure-functions/functions-best-practices. Last accessed: 09/13/2021.

[48] Best practices for organizing larger serverless applications. https://aws.amazon.com/blogs/compute/best-practices-for-organizing-larger-serverless-applications/. Last accessed: 09/12/2021.

[49] Aws step function. https://aws.amazon.com/step-functions/. Last accessed: 09/12/2021.

[50] Durable functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview. Last accessed: 09/12/2021.

[51] Factorial cloud functions. https://doi.org/10.5281/zenodo.5748653. Last accessed: 12/01/2021.

[52] Spring boot. https://spring.io/projects/spring-boot. Last accessed: 09/26/2021.

[53] Java threadpoolexecutor. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html. Last accessed: 09/26/2021.