Università degli Studi di Padova

Department of Mathematics
"Tullio Levi-Civita"

Master's degree in Computer Science

Achieving Reproducibility in Cloud Benchmarking: A Focus on FaaS Services

Supervisor
Prof. Tullio Vardanega

Undergraduate
Saadat Nursultan

Academic Year 2022-2023

Abstract

In the dynamic realm of Cloud computing, the challenges of research reproducibility and precise performance estimation are paramount. This thesis addresses two intertwined concerns: the reproducibility problem and the accuracy of performance estimation within the Cloud computing environment. The study's primary objectives include a comprehensive examination of experiment reproducibility as outlined in the original SeBS and Faasdom papers. Through meticulous reproduction, the aim was to assess the feasibility of reproducing these experiments in the evolving Cloud computing landscape. A comparative analysis to evaluate the alignment between the findings in the original papers and the outcomes of the reproduced experiments was conducted. By addressing these critical challenges, the aspiration is to foster a more reliable and informed Cloud computing ecosystem, where research findings stand as sturdy pillars upon which future innovation can flourish. It was discovered that running FaaS benchmarking tools that lack up-to-date contributions pose several challenges because they are not well-adjusted to incorporate technological advancements. Furthermore, the findings suggest that the conclusions presented in the SeBS and FaaSdom related literature remain relevant.

Contents

List of Figures

List of Tables

Chapter 1

Defining the Challenge

1.1 Adoption of the Cloud

The evolution of technology has brought about a transformative shift in the way businesses operate and manage their IT resources. Before the advent of Cloud computing, companies relied on on-premise solutions where servers were physically located within their offices, and in-house developers were responsible for managing both hardware and software. However, this approach presented several challenges and limitations [11].

- **Limited capacity of hardware**: one of the primary drawbacks of on-premise solutions was the finite capacity of hardware. This limitation directly impacted the number of users who could simultaneously connect to and utilize the server resources. As businesses expanded, they often found themselves constrained by the scalability limitations of their on-premise infrastructure.
- **Waste of resources in case of over-provisioning**: on the other side, maintaining servers on-premise could lead to over-provisioning. Over-provisioning occurs when a company keeps servers running even when they are not fully utilized. This results in wasted resources, including excessive electricity consumption and unnecessary labour costs associated with managing and maintaining idle hardware.
- **Maintenance of the servers**: hosting servers in-house also required creating a suitable environment for the machines to operate efficiently. This included addressing issues such as cooling, electricity provisioning, and ensuring limited access to sensitive equipment. The operational overhead of maintaining servers in a physical location added complexity and costs to IT management.
- **Manual software update and licensing**: additionally, on-premise solutions demanded the tedious task of managing software licences. Organizations had to keep track of software licences, often resulting in complexities, especially when dealing with a myriad of software components. Furthermore, updating software typically required downtime, and ensuring backward compatibility added another layer of complexity to IT operations.

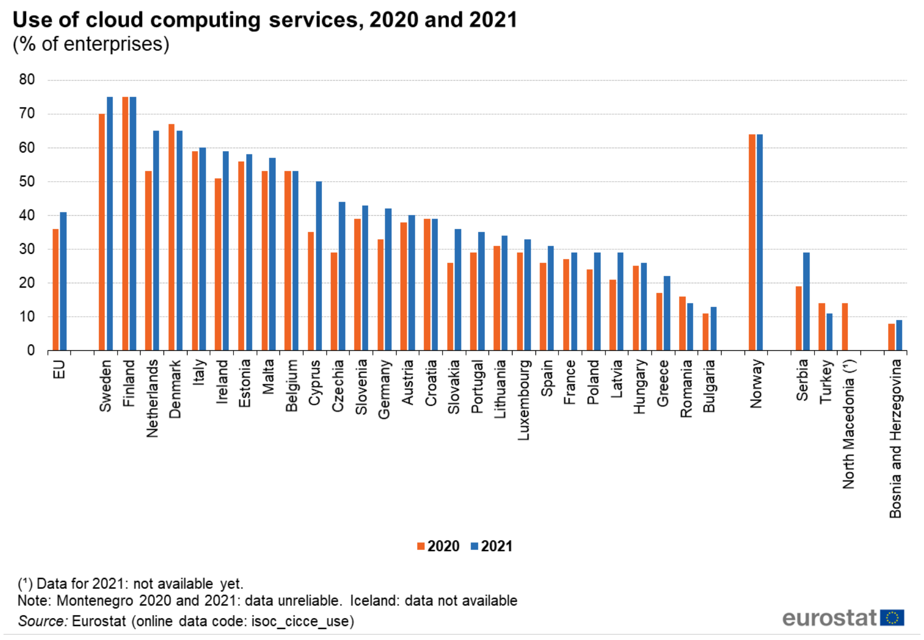


Figure 1.1: Usage of Cloud computing in enterprises in European Union (% of enterprises with internet access) [1].

Monitoring the support lifecycle, particularly for long-term support (LTS) software components, became a critical but time-consuming task.

By transitioning to the Cloud, organizations could offload many of the challenges associated with on-premise solutions [12]. Clouds changed the computing landscape with the promise of plentiful resources, economy of scale for everyone, and on-demand availability without up-front or long-term commitment. Reported costs of running an application in a Cloud are up to seven times lower than that of a traditional in-house server [13]. Figure 1.1 shows a steady growth of interest among the enterprises in European Union.

Thus, while organizations are interested in migrating their applications to the Cloud, they face a need to choose among various options in the market [14]. When organizations embark on the journey of selecting a Cloud provider, several critical characteristics come into play. Safety, cost, and efficiency stand out as pivotal factors that influence this decision-making process. Businesses keen on optimizing their solution development workflow are acutely aware of the need to assess the financial implications of adopting a particular Cloud service [15]. This evaluation involves comparing the costs associated with Cloud solutions against those of traditional or existing methods. Moreover, it is paramount to comprehend the performance limitations of any chosen service, such as execution time, memory consumption, and throughput. This understanding not only aids in optimizing resource utilization but also facilitates the comparison of performance constraints among a diverse range of Cloud providers.

1.2 Cost of Migration

Given the demand, there exists a vast amount of literature that compares and ranks Cloud service providers along with their respective services. From insightful blog posts [16, 17, 18] to comprehensive 70-page annual surveys [19, 20, 21], there is no shortage of information sources available to assist individuals and organizations in making informed decisions when selecting a suitable Cloud provider.

These resources come in two primary categories: qualitative and quantitative. Qualitative studies delve into various aspects of Cloud providers' offerings, such as their support systems, the strength of their user communities, the range of supported programming languages and runtimes, and more. These qualitative assessments provide valuable insights into the non-quantifiable factors that can greatly influence the user experience and overall satisfaction with a particular Cloud service.

On the other hand, quantitative studies take a more data-driven approach. They employ benchmarking as a powerful tool to collect numerical data, enabling a systematic and objective comparison of Cloud providers. Through benchmarking, one can evaluate and contrast performance metrics, cost-efficiency, scalability, and other quantitative attributes. This approach allows users to make well-informed choices based on empirical evidence rather than mere conjecture. By facilitating transparency, fostering competition, and promoting data-driven decision-making, systematic benchmarking and evaluation guide users and providers alike in their adoption and implementation of Cloud solutions.

Vazquez et al. [22] state that benchmarks should accurately depict the specific workloads that consumers intend to execute. It is essential to recognize that benchmarks for distinct purposes, such as those designed for social networking platforms versus database systems, should be distinct in nature. Even applications sharing the same computing infrastructure may exhibit varying demands concerning computing resources, storage, and networking [22]. This recognition has led to the development of diverse benchmarking tools, catering to various application domains in the market. Table 1.1 provides an overview of four prominent benchmark suites, each tailored to specific areas of interest.

Benchmark suite	Netperf [23]	HiBench [24]	YCSB [25]	CloudSuite [26]
Description	Allows measuring the network performance of various network protocols and systems through standardized benchmarking tests	Designed to evaluate and measure the performance of various big data processing frameworks and systems	Allows measuring the performance of numerous modern NoSQL and SQL database management systems with simple database operations on synthetically generated data	Consists of eight benchmarks representing popular online services and analytic workloads in datacenters
Initiated by	Hewlett Packard	Intel	Yahoo!	EPFL
Metrics Provided	Throughput (Mbps) Latency (ms) Packet loss rate (%) Jitter (ms) CPU usage (%) Memory utilization (%) Network bandwidth utilization (%) Number of concurrent transactions	Throughput (ops) Latency (ms) Data processing time (s) Network throughput (Mbps) Disk I/O (ops) Error rate (%) CPU usage (%) Memory utilization (%)	Throughput (ops) Latency (ms) Read/write operations Error rate (ratio) CPU usage (%) Memory utilization (%) Network throughput (Mbps) Number of concurrent transactions	Throughput (ops) Response time (s) Latency (s) CPU utilization (%) Memory utilization (%) Network throughput (Mbps) Disk I/O (ops) Disk throughput (MB/s) Number of concurrent transactions Energy Consumption (Wh) Cost Fault tolerance (MTBF)
Service	Network	PaaS	PaaS	IaaS
Primary Use Cases	Cloud infrastructure	Big data frameworks	Data serving systems	Data serving systems
Workloads	Network performance Bandwidth assessment	Micro-benchmarks Web search Machine learning Hadoop Distributed File System	Data serving	Data analytics Graph analytics In-memory analytics Web search Media streaming Web serving Data caching Data serving

Table 1.1: Prominent Cloud benchmarking suites and their characteristics.

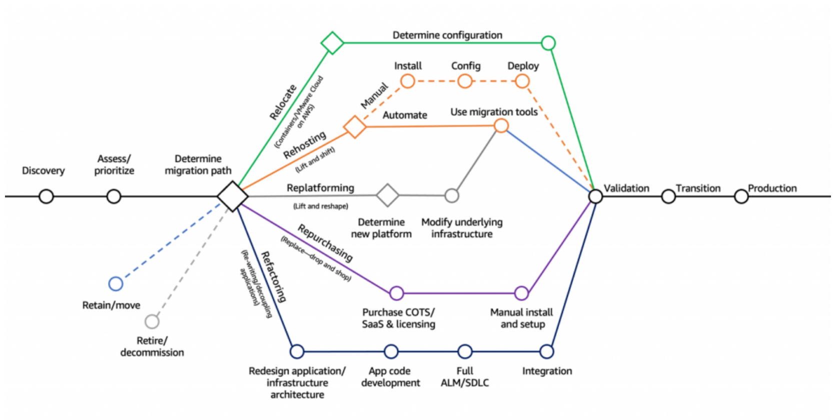


Figure 1.2: 7 R's mental model for migrating applications to the Cloud [2].

1.3 Case Study

Benchmarking can help enterprises at the initial state of exploring Cloud solutions. To illustrate, consider the scenario of Company X, faced with the task of selecting an optimal Cloud provider.

Company X must first conduct a comprehensive analysis of their existing on-premise system and devise a migration plan (Figure 1.2). This preparatory phase enables Company X to gain insights into the architecture of the forthcoming Cloud application, involving workload patterns and data schemas. Armed with this information, benchmarking Cloud service providers for specific workload types becomes a straightforward process. The only missing element is the right tool that will allow consistent benchmarking. This is when benchmarking tools prove invaluable.

There exists a wide array of open-source benchmarking tools and suites designed for Cloud computing services. Notably, all benchmarks presented in Table 1.1 were initially developed as in-house projects and later were made open-source.

These tools offer a valuable advantage as they can be customized to align with the specific requirements of a business while also being cost-effective due to their open-source nature. For Company X the utilization of readily available benchmarking suites or versatile benchmark toolkits, such as PerfKit Benchmarker [27], which provides wrappers and workload definitions around popular benchmark tools, can prove to be instrumental. This empowers Company X to thoroughly assess the performance of various Cloud providers without the need for cumbersome migration of an entire existing application.

However, amidst the benefits, a significant challenge emerges. Many benchmarking tools, originally celebrated for their adaptability and cost-effectiveness, often originate from scientific research endeavours [28] which are no longer actively maintained. For instance, as of the time of this writing, both Netperf and YCSB projects have their last contributions in 2021 [23, 29]. This presents Company X with a pressing issue, as outdated versions of dependency packages can hinder the effectiveness of these tools. Furthermore, the reliance on external APIs introduces an additional layer of complexity, as changes in their interfaces over time can lead to installation and usage complications, undermining the very reproducibility and reliability sought in benchmarking processes.

This leads to a critical question: Is there a more fundamental problem that needs to be examined, one that goes beyond the immediate challenges related to reproducibility? This thesis aims to investigate open-source benchmarking tools, uncovering where reproducibility issues originate, their implications, and any larger concerns that might be causing these challenges.

1.4 The Problem of Reproducibility

The term ”reproducibility” has sparked considerable debate, with variations in its interpretation across various fields. In the context of informatics and data analysis, a reproducible analysis is characterized by its ability to be rerun by a different researcher, resulting in identical outcomes and conclusions [3]. Reproducibility shares connections with repeatability and replicability, but it is important to distinguish these terms.

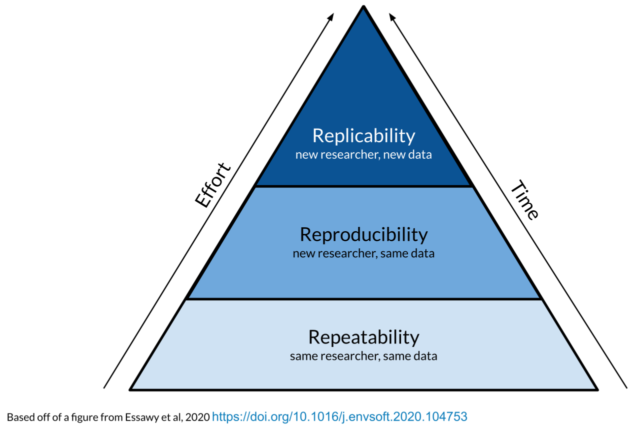

Replicability
new researcher, new data
Reproducibility
new researcher, same data
Repeatability
same researcher, same data
Effort
Time
Based off of a figure from Essawy et al, 2020 https://doi.org/10.1016/j.envsoft.2020.104753

Figure 1.3: Reproducibility’s relation to repeatability and replicability [3].

The challenge of reproducibility is a significant and prominent issue within the scientific community [30, 31, 32, 33]. Scientific research endeavours, ideally, serve as solid foundations upon which subsequent works can build and expand. Yet, the stark reality reveals a disconcerting trend where results obtained in research studies often elude successful reproduction. This lack of reproducibility stems from a myriad of potential reasons, each presenting a roadblock to the reliability and longevity of scientific contributions [34]. As research garners substantial support in the form of grants, cutting-edge equipment, and dedicated human effort, it paradoxically struggles to yield enduring results that can be effectively leveraged for the advancement of science.

To expedite and bolster the pace of scientific progress, there is a crucial need to streamline and enhance the research process, rendering it more dependable and readily reproducible [35]. Central to this endeavour is the essential notion that scientific work should be conducted in a manner that inherently facilitates reproducibility [36]. Thus, there arises a question of what makes an experiment reproducible, thereby unveiling the key attributes that researchers can cultivate to augment the reproducibility of their studies.

Although it presents a global challenge, the problem of reproducibility in science is not unsolvable. Numerous communities and institutions have actively engaged in addressing this issue by orchestrating initiatives that encourage emerging researchers to reproduce the findings of earlier, unverified studies [37, 38, 39, 40, 41]. This organized effort aims to systematically reproduce scientific experiments, tackling them discipline by discipline and paper by paper. In addressing this question, the scientific community embarks on a journey to fortify the very foundations of scientific inquiry, fostering an environment where research outcomes stand as robust pillars for the edifice of knowledge and innovation, poised to accelerate the course of scientific discovery.

1.5 Scope

This master's thesis addresses benchmarking Function-as-a-Service (FaaS) offerings from three of the most prominent Cloud service providers: Microsoft Azure, Amazon Web Services (AWS), and Google Cloud (Figure 1.4). The research is conducted with a focus on practicality and affordability, aligning with the broader reproducibility initiative in Cloud computing.

The primary objective of this study is to evaluate and compare the performance of FaaS services, namely the execution time of the functions, the time in which a client can get the response from the server, and how well the FaaS platforms can handle sudden spikes in arriving requests. This evaluation will involve the reproduction of two benchmarking studies: "FaaSdom: A benchmark suite for serverless computing" by Maissen et al. [42] and "SeBS: A serverless benchmark suite for function-as-a-service computing" by Copik et al. [9]. By undertaking the reproduction of these studies, the aim is to contribute to the ongoing efforts to enhance the reliability and applicability of scientific research in Cloud computing.

The scope of this thesis emphasizes quantitative analysis, which will involve data collection and statistical analysis.

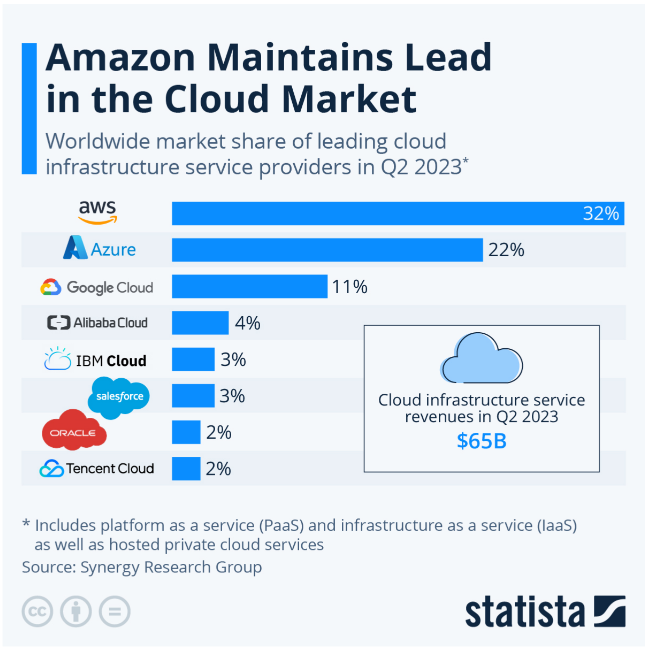


Figure 1.4: Worldwide market share of leading Cloud infrastructure service providers in Q2 2023 [4].

1.6 Research questions

This thesis aims to answer the following critical questions:

- **Is it possible to reproduce the results of previous Cloud benchmarking studies?** This question underscores the feasibility and challenges of reproducing Cloud benchmarking experiments, especially those lacking recent contributions.
- **What are the main factors contributing to the reproducibility of Cloud benchmarking results?** Understanding the key factors that influence reproducibility is essential for devising strategies to improve benchmarking practices.

1.7 Contributions

- Providing empirical data and insights on the performance of FaaS offerings from major Cloud providers.
- Offering a practical guide for businesses considering Cloud adoption, aiding in informed decision-making.
- Supporting the broader reproducibility initiative by reproducing and extending previous benchmarking studies.

1.8 Limitations

This study has a number of limitations, of which the author is fully aware:

- The focus on specific Cloud providers may not encompass the entire spectrum of FaaS offerings.
- The study may not account for all potential variations due to specific use cases and configurations.
- The assessment omits an examination of the expressivity of benchmarks and their alignment with business requirements.

Chapter 2

Understanding the Landscape

2.1 Cloud Computing

Cloud computing facilitates the convenient and on-demand utilization of computing resources like networks, servers, storage, applications, and services via network connectivity [43]. As illustrated in Figure 2.1, Cloud computing is typically classified into three primary models based on the degree of user management involvement: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). In the context of IaaS, users gain access to fundamental infrastructure resources, such as networking, storage, and virtual machines (VM), without needing to handle hardware management, while retaining substantial control over the infrastructure. PaaS, in contrast, eases the operational responsibilities associated with managing the core Cloud infrastructure. It provides developers with an array of tools and services, allowing them to concentrate on building tailored environments and applications. At the other end of the spectrum, SaaS delivers complete software products directly to end-users via the internet, eliminating the need for users to manage any underlying infrastructure. Leading Cloud providers like Amazon Web Services (AWS), Microsoft Azure, and Google Cloud offer a comprehensive range of Cloud services built upon these three model types.

Cloud computing offers organizations and individual developers a significant advantage by eliminating the need for substantial upfront capital investment during application deployment, particularly in experimental and early development phases. Instead, they gain instant access to a vast pool of computing resources and services without having to commit to substantial upfront costs. Users only pay for the resources they have actually consumed, following a pay-as-you-go model. This not only accelerates time-to-market and fosters innovation but also leads to cost savings [13].

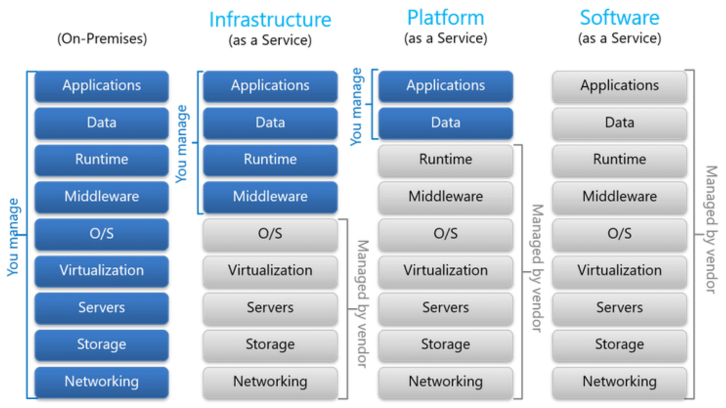


Figure 2.1: Shared responsibility model defines what customer can control, and the level of control changes according to the type of the service [5].

2.2 Serverless Computing

2.2.1 Serverless Application

Serverless computing gives developers an opportunity to build and run applications without having to manage Cloud infrastructure [44]. Although the model is called "serverless", virtual servers are still used as the underlying infrastructure, but they are abstracted away from app development. In a serverless model, a Cloud provider handles the routine infrastructure management and maintenance, including operating system updates and patches, security management, capacity planning, and system monitoring. Meanwhile, developers can focus on code for deployment [45].

Serverless can be considered a bridge between PaaS and SaaS in a sense that the developer doesn't handle server (virtual machine) entirely, as he would do in PaaS, but unlike in SaaS, the developer still has the ability to manage the front-end application code and business logic [45].

Additionally, serverless computing offers a ready-made scalability solution, including the ability to scale down to zero, a task that typically demands advanced expertise and substantial effort. Serverless Computing follows a "pay per use" pricing model where users are billed solely when their applications actively serve requests or events, aligning more closely with the original vision of Cloud computing as a utility service [46]. Some examples of serverless services are FaaS, serverless databases, event streaming and messaging, and API gateways [47, 48, 49].

With the availability of Function as a Service (FaaS) and other serverless components offered by Cloud providers, developers have the capability to construct fully-fledged serverless applications. These applications typically employ FaaS as the computing layer for hosting and executing business logic code, complemented by other fully managed or serverless services for functions like data storage, messaging, streaming, and user authentication/authorization [50]. Eismann et al. [51] derived several key characteristics of serverless applications through a comprehensive examination of 89 serverless applications from open-

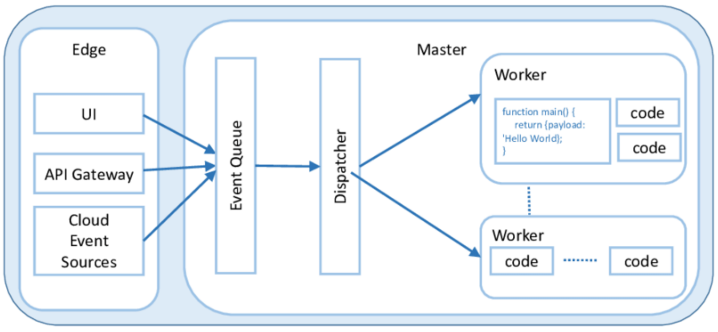


Figure 2.2: FaaS is a subset of serverless computing that is focused on event-driven triggers, where code runs in response to events or requests [6].

source projects, academic literature, industrial documentation, and domain-specific feedback:

- AWS stands as the dominant platform for serverless applications, encompassing 80% of all studied applications, with Azure accounting for 10%. This prevalence can be attributed to AWS being the first major Cloud provider to introduce serverless computing, with AWS Lambda pioneering the concept of FaaS [51].
- Serverless applications are primarily utilized for brief tasks with low data volume and sporadic workloads, primarily because providers impose limitations on the duration a function can run per execution. For instance, AWS Lambda permits execution times of up to 15 minutes per operation, while Azure Function in the Consumption plan has a timeout limit of only 10 minutes [51].
- Cloud storage, Cloud databases, and Cloud messaging services are commonly employed within serverless applications.
- HTTP triggers and Cloud events are the most frequently employed triggers in serverless applications. As stated by Microsoft, the HTTP trigger lets you invoke a function with an HTTP request [52].
- Most serverless applications employ a limited number of Cloud functions, with 82% of them utilizing five or fewer functions [51].
- The most popular programming languages for serverless applications are Python and JavaScript, likely due to interpreted languages such as Python, Ruby, and JavaScript having significantly shorter cold-start delays compared to compiled runtimes like Java and .NET. Node.js was the first language runtime supported by Amazon Lambda, since it has fast startup

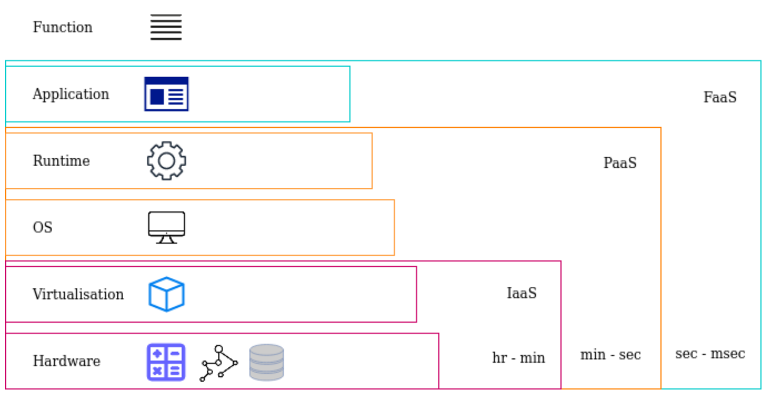


Figure 2.3: Service provision time reduces as the provisioned resource becomes lighter [7].

times and a low memory footprint suitable for customer-facing functions that demand low latencies. [53]. Moreover, JavaScript and Python are both relatively easy to learn and are widely used in other production areas [54], which in turn expands the customer base for the serverless platforms. However, C# remains the primary choice for serverless applications on Azure, primarily because it was the initial and most supported language offered by Azure [55, 56].

2.2.2 Function as a Service

The predominant form of serverless computing, known as FaaS exemplified by services like AWS Lambda and Azure Functions, revolves around the concept of using functions as the fundamental units of computation. These functions execute user-written code in response to various triggers, such as events or HTTP requests [46]. This approach offers an appealing alternative to implementing microservices-based architecture, a popular approach for constructing applications comprising small, self-contained components that can be independently scaled, with the aim of enhancing development efficiency and scalability. One of the primary advantages of FaaS over the microservices based architecture is the reduced provision time, as it is shown in Figure 2.3.

Behind the scenes of serverless functions, Cloud providers dynamically provision function instances, scaling them as needed [57]. The lifecycle of a function begins when it receives an event, such as an HTTP request, which is routed to an available function instance via a function routing service. In cases where no function instance is readily available, the resource manager, responsible for scheduling and overseeing all function instances, must create a new one based on user-specified configurations. This process involves establishing a new function instance and initializing it with the required container image and code assets, leading to what is termed a ”cold start” [42]. Once the function instance is fully prepared, it can process the invocation request and execute the business logic contained within the user's code. In instances where a function instance is already active, referred to as a ”warm instance”, this cold start phase can be bypassed, allowing the event to be processed directly.

2.3 Performance Benchmarking

2.3.1 Benchmarking Basics

Benchmarking in Computer Science is a systematic and empirical approach utilized to assess and compare the performance of various computer systems, tools, techniques, and more. Typically, a benchmark centres around a System Under Test (SUT), comprising components essential for executing a benchmark scenario [58]. This SUT also encompasses the complete application architecture, including the components of interest. Essential elements of a benchmark encompass a clearly defined rationale for the comparison, representative tasks or workloads, and the types of measurements, whether quantitative or qualitative.

The key considerations for constructing a benchmark are elucidated as follows. Effective benchmarking necessitates striking a balance among these characteristics and criteria:

1. **Relevance**: The degree to which the benchmark and its results are pertinent and applicable to a particular domain and relevant stakeholders. For those utilizing the benchmark results, relevance must also account for the specific context and use cases. Scalability poses a significant challenge to relevance, as benchmarks are expected to run on a broader system and simulate real application behaviours. Enhancing benchmark relevance in a specific domain often involves narrowing its applicability.

2. **Reproducibility**: Given the same underlying hardware infrastructure and system configuration, the benchmark should consistently yield similar results. Achieving absolute reproducibility can be challenging due to the variability in modern software systems. In practice, reproducibility can be enhanced by running the benchmark for a sufficient duration and/or in different sandboxes to encompass representative samples of variable behaviours. This may also entail conducting multiple runs to enhance consistency.

3. **Verifiability**: Other researchers and interested parties should be able to use the benchmark to verify its results and establish trustworthiness. A crucial approach to enhancing verifiability involves providing comprehensive details about the benchmark and the data used.

4. **Fairness**: Ensuring that systems can be compared based on their metrics without artificial constraints. Improving fairness typically involves designing benchmarks based on consensus among a panel of experts rather than individual parties and considering how the results will be utilized.

2.3.2 Serverless Benchmarking

Among the various Cloud computing paradigms, FaaS has emerged as a particularly intriguing area of research and development. The attention from the researcher's community produced a number of papers that came up with benchmarks to compare different providers and services in order to establish which provider has the lowest latency, the lowest cost and highest throughput [42, 9, 59, 60].

In the realm of serverless computing, there are generally two main categories of benchmarks: micro-benchmarks and application benchmarks [61]. Depending on the chosen benchmark type, serverless benchmarking can be classified into two distinct approaches: micro-benchmarking and application benchmarking.

Micro-benchmarking focuses on assessing specific aspects of serverless functions by using individual functions for evaluation. This involves evaluating factors such as CPU and memory performance, disk I/O performance, and network performance using a single function. For instance, an AWS Lambda function can be designed to handle a specific function where it receives parameters from its triggering events and then conducts floating-point operations to measure the latency of CPU-intensive computations [61]. Another example of micro-benchmarking, as seen in FunctionBench [59], employs a single function coupled with Cloud storage to measure network performance by downloading and uploading objects.

Conversely, application benchmarking entails the use of applications composed of multiple serverless components and typically measures the overall response time from start to finish. As an illustration, BeFaaS utilizes an e-commerce benchmark that encompasses 17 functions and relies on a Redis instance as an external service for state persistence [60]. Another example of an application benchmark might involve an Image Processing application that performs image transformation using the Pillow library [62]. This application retrieves an input image from shared block storage, applies various effects to it, and then uploads it to another shared storage [59].

2.4 Benchmarks of Interest

2.4.1 FaaSdom

FaaSdom is an open-source serverless benchmark suite. It automatizes the deployment, execution and clean-up of tests, as well as able to provide insights on the performance via a graphical user interface. As shown in Figure 2.4, the tool consists of four main modules and is integrated with IndexDB for data storage and Graphana for data visualization. The tool currently supports only four Cloud platforms: AWS, Azure, Google, and IBM.

FaaSdom benchmark collection consists of two compute intensive benchmarks (*faas-fact* and *faas-matrix-mult*) and two IO-intensive benchmarks (*faas-netlatency* and *faas-diskio*) written in both Node.js and Python. Notably, the benchmark suite can be extended by adding custom benchmarks.

- *faas-fact*: performs integer factorization.
- *faas-matrix-mult*: performs multiplication of large matrices many times.
- *faas-netlatency*: measures network latency by sending a small-sized HTTP response as soon as the benchmark function is invoked.
- *faas-diskio*: measures disk I/O performance.

The benchmark deployment process is parallellised, including creation of Cloud resources, the build and packaging of the functions and their upload to the Cloud. The invocation process yields processed output including invocation parameters, execution time, throughput, and statistics relate to the execution time: average, mean, max latency and execution time distribution.

Maissen et al. [42] evaluated metrics such as call latency (roundtrip), cold start, and throughput. To evaluate call latency across different language runtimes

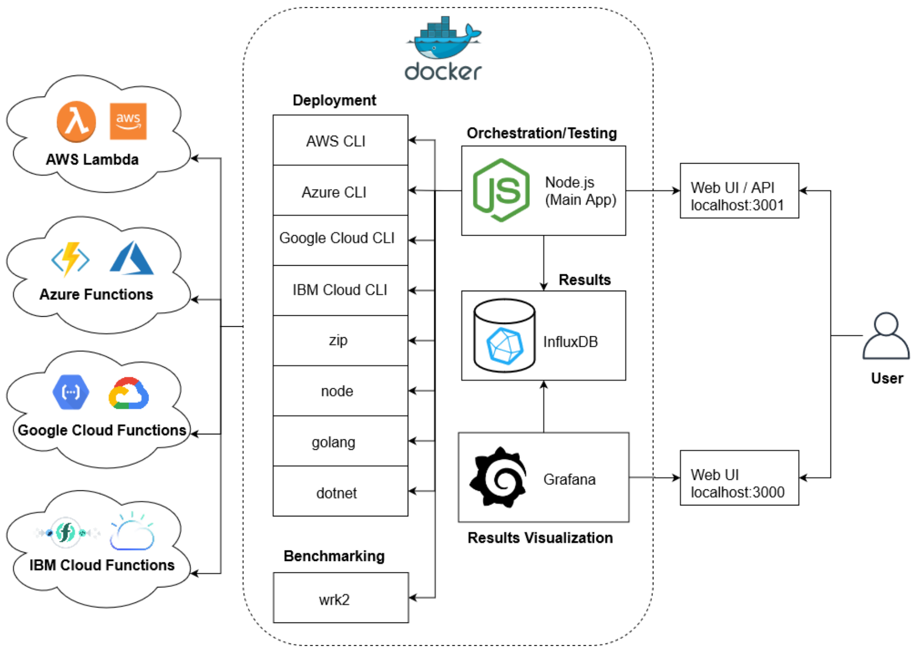


Figure 2.4: Architecture of FaaSdom application fully depends on docker containers [8].

and Cloud service providers, they performed experiments using Node.js version of *faas-netlatency* benchmark deployed at all available regions for each provider using 128 MB of memory where available. Figure 2.5 presents the results obtained from the original study.

CPU bound Throughput/Latency experiment was performed involving *faas-matrix-mult* benchmark from FaaSdom toolkit. The experiment involved function invocations at different rates in the range from 10 to 1000 requests per second for each provider/language runtime configuration. FaaSdom tool relies on another HTTP-based benchmarking tool wrk2 [63], an extended version of wrk [64], to perform the Throughput/Latency experiment.

The evaluation of experiments reveals that AWS Lambda is the best-performing Cloud provider with the lowest call latency, average cold start latency, and most stable response. Furthermore, they came to the conclusion that the time of the execution is linearly correlated with the size of allocated memory.

2.4.2 SeBS

SeBS (Serverless Benchmark Suite) is and open-source suite of diverse FaaS benchmarks that allows automated performance analysis of commercial Cloud platforms like AWS, Azure, Google Cloud, and open source platforms like OpenWhisk. The tool enables automatic deployment and invocation of benchmarks, as well as parallel experiments that model and analyse the behaviour of FaaS system. SeBS regards cost efficiency as a primary metric to determine the most efficient configuration for a specific workload. To ensure seamless binary compatibility with the Cloud, SeBS encapsulates benchmarks and their

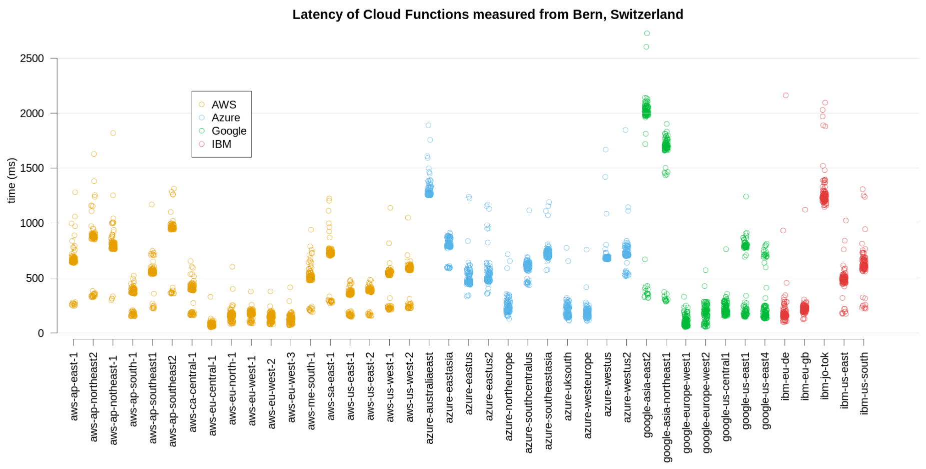


Figure 2.5: Latency of Cloud functions across different regions and providers obtained by the authors of FaaSdom tool [8].

dependencies within Docker containers, resembling function execution workers. Additionally, a unified interface has been implemented for each platform (Figure 2.6).

The benchmark suite consists of total 10 representative workloads and can be extended with custom benchmarks. There are five types of benchmarks: webapps, multimedia, utilities, inference, and scientific.

- *dynamic-html*: dynamically generaties HTML file from a predefined template.
- *storage-uploader*: uploads a file from a given URL to a Cloud storage.
- *thumbnailer*: creates a thumbnail of an uploaded image.
- *video-processing*: uses a static build of ffmpeg to apply a watermark on a video and converts it to a gif file.
- *compression*: compresses a set of files and returns an archive to the user.
- *data-visualisation*: passes DNA data to a function which generates specified visualisation and caches results in the storage.
- *image-recognition*: recognizes an object on the image using Res-Net-50 model.
- *graph-pagerank*: performs irregular graph computations to identify page ranking.
- *graph-mst*: computes minimum spanning tree of a graph.
- *graph-bfs*: performs breadth first search on a graph.

In their experiments, the authors employed various invocation methods, including an abstract trigger interface, and utilized both Cloud SDK and HTTP triggers. The invocation process yields measurements and an unchanged output

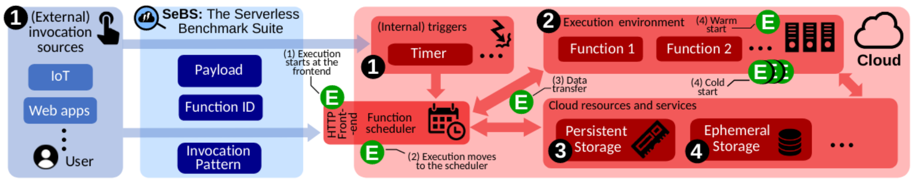


Figure 2.6: Integration of SeBS benchmarking tool with a FaaS platform. [9].

of the benchmark application. The SeBS functions were implemented within function wrappers and leveraged the provider's log querying capabilities.

Their research revolved around conducting a comprehensive performance and cost analysis across three major Cloud service providers: AWS, Azure, and Google Cloud Platform (GCP). To achieve this, they executed a total of 200 cold invocations and 200 calls to a pre-warmed container on each platform. The functions were invoked in batches of size 50 to ensure different sandbox parameters.

The study's findings highlighted a significant correlation between memory allocation and CPU as well as I/O allocation. Furthermore, running *uploader* and *compression* benchmarks unveiled a wide distribution of latencies, characterized by numerous outliers. This wide variability in latency hinders the ability to attain predictable and consistent performance from serverless functions.

When evaluating the performance of benchmarked FaaS platforms, their results showed that AWS Functions consistently outperformed the other Cloud service providers. After analysing the results of experiments, they came to the conclusion that serverless applications benefits from the larger resource allocations, particularly in terms of memory usage. However, the study emphasized the importance of developing precise methodologies for measuring the performance of short-duration serverless functions to select the right configuration.

	SeBS	FaaSdom
Supported FaaS providers	GCP, Azure, AWS	GCP, Azure, AWS, IBM, OpenWhisk
Test scenarios and workloads	Dynamic HTML Storage uploader Thumbnailer Video processing Compression Data visualisation Image recognition Graph pagerank Graph MST Graph BFS	Latency test CPU test factors CPU test matrix Filesystem test
Benchmarking metrics	CPU utilization* (%) Memory utilization* (bytes) I/O* (ops) Code size (bytes) Provider execution time (ms) Client execution time (ms) Benchmark execution time (ms)	Throughput (rps) Latency (ms)
Languages supported	Python, Nodejs	Nodejs, Python, Go, .NET
Provided documentation	Installation guide Usage guide Provider configuration Benchmark specifications	Installation guide Benchmark specifications Provider configuration
Interface type	Command-line interface	Web user interface
Customization and flexibility	Allows to configure batch size, N, number of every type of start	Allows to configure N, duration of the test, requests per second
Reporting and visualization	No visualization, has textual output and outputs processed csv table	Has integration with Grafana for visualization. Tabular data stored in InfluxDB
Community and support	Has an active repository where one of the authors answers questions and helps resolve problems.	The repository was not active since June 15th, 2020. Provides no tutorials or platform to report issues
Open Source vs. commercial	Open-source BSD 3-Clause License	Open-source Apache License
Regression testing	Allows performing regression tests locally	No regresion testing
Cost and performance estimation	Allows to estimate both performance and cost	Allows to estimate both performance and cost
Security	Requires adding Cloud credentials in the config file. Saves secret credentials in resulting files	Asks for the credentials on program start. Uses Cloud providers' login and access verification services
Updates and maintenance	Last update 23 August, 2023	Last update June 15th, 2020

Table 2.1: Qualitative comparison of SeBS and FaaSdom. Metrics denoted with the star (*) are local to the Cloud provider.

Chapter 3

Experimental Process and Analysis

3.1 Proposed Methodology

Aligned with the scope of this Master's thesis, the experiments were performed to evaluate and compare the performance of Amazon Lambda, Google Cloud Functions and Azure Functions. The evaluation was intended to be based on quantitative analysis of the data obtained from benchmarking FaaS platforms using SeBS and FaaSdom open-source benchmarking tools. Given that the tools emerged from scientific research endeavours culminating in the publication of scientific literature, another aim of this thesis was to evaluate the reproducibility of those conducted studies.

The experiments had a dual purpose: either to validate the findings highlighted in scientific papers by Maissen et al. [42] and by Copik et al. [9], or to identify disparities with recently acquired data. These experiments were specifically designed for collecting performance data related to FaaS platforms. Specifically, the intent of this study was to employ the SeBS and FaaSdom tools for benchmarking, thereby investigating potential causes for any reproduction failures. Ultimately, the overarching objective was contributing to the ongoing efforts to enhance the reliability and applicability of scientific research in Cloud computing (Figure 3.1).

To achieve research objectives, the experimental settings and procedures described in two original papers were reproduced. Subsequently, the acquired raw experimental data was processed. Graphical representations were generated to provide a bird's-eye perspective of the data and can be found in the section 3.4 Data Analysis. Those results were rigorously compared with the original findings presented in the source papers. The evaluation primarily focused on two key facets:

- **Comparative Analysis**: The disparities between the results obtained through the experimentation and those originally documented in the source papers were analysed. This entails both qualitative and quantitative comparisons, enabling us to discern any divergences in outcomes.

- **Identification of Obstacles**: The investigation diligently catalogued and analysed any impediments encountered during the process of reproducing the experiments. This qualitative examination is invaluable for elucidating the challenges and intricacies inherent in utilizing the benchmarking tools.

As a part of this thesis, an experiment that is referred to as *perf-cost* in Copik et al.'s paper was performed. The collected data was used to create box plots, to obtain a statistical data similar to one in the original paper. Additionally, overarching trends in the data were identified, aiding in the drawing of conclusions about the consistency of FaaS providers' ranking. All events hindering the reproducibility of the original study were documented and communicated to the original paper's co-author and SeBS repository maintainer Marcin Copik [65] [66] [67] [68].

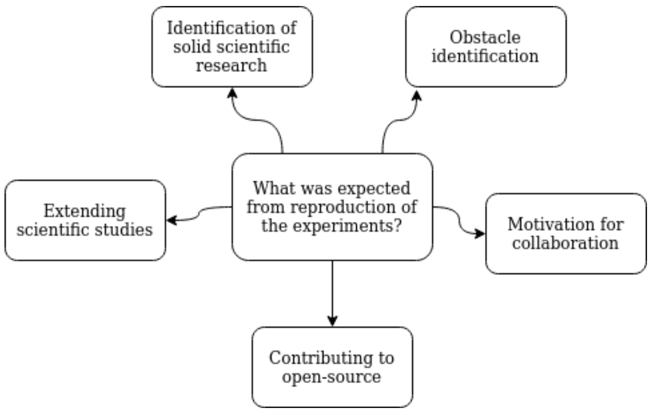

Identification of solid scientific research
Obstacle identification
Extending scientific studies
What was expected from reproduction of the experiments?
Motivation for collaboration
Contributing to open-source

Figure 3.1: Expected outcomes of experiment reproduction.

Similarly, in the reproduction of Maissen et al.'s work [42], a grounded and methodical approach is being maintained. Call latency (*faas-netlatency*), throughput (*faas-matrix-mult*), and CPU-bound assessments (*faas-fact*) were being conducted. Once the data was gathered, the results of experiments were visualized with scatter plots and line graphs, just as the original paper did. With this approach, a compatible overview of providers' performance was created for the evaluation of differences from the original paper. All blocking events encountered during the execution of experiments were thoroughly documented, and a new repository was created to contain all the updates performed to the original repository.

Both experiments were conducted on remote virtual machines (VMs). Running an experiment on a remote VM provided several benefits. The key advantages that were considered are isolation and experiment reproducibility.

- **Isolation**: Remote VMs provide a sandboxed environment that is isolated from the local machine. This isolation ensures that the experiment doesn't interfere with the local system or other running processes.
- **Experiment reproducibility**: By documenting the VM's configuration and software environment, one can ensure that the experiments are reproducible, which is crucial for future research and validation.

The graphs for visualisation of results were constructed using a Python program running on Google Colab platform [69]. The code used for the construction of the graphs as well as the source data obtained during the experiments are available online at GitHub code sharing platform:

- Produced copy of SeBS repository, which contains the results of SeBS related experiments obtained during this study [70]
- Produced copy of FaaSdom repository, which contains reworked source code of FaaSdom tool [71].

3.2 Experimental Setup

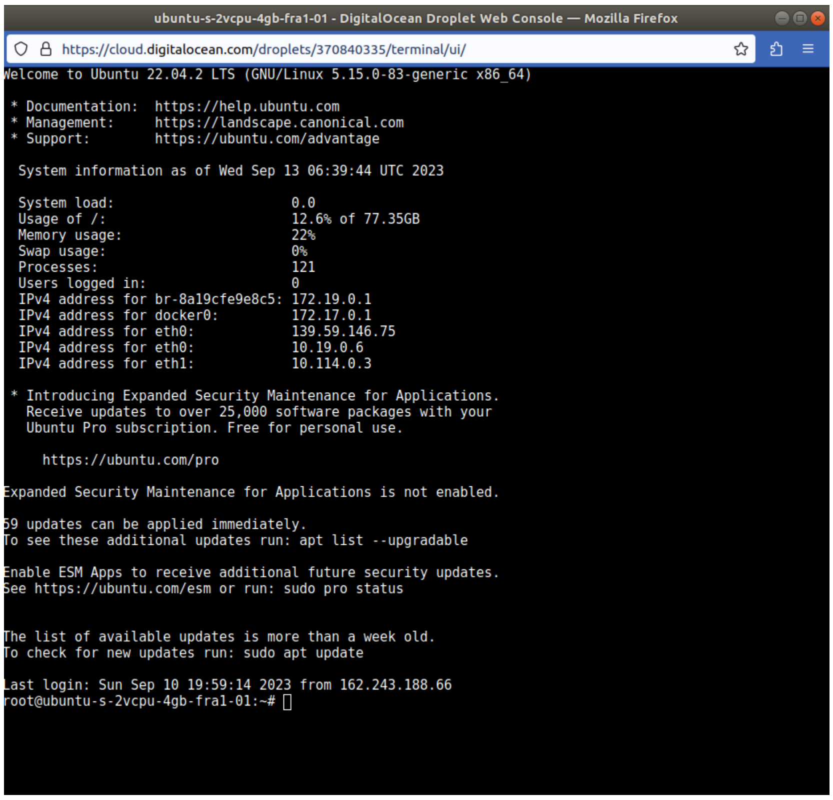
The experiments were run on an environment composed of two identical virtual machines (VM) on DigitalOcean platform [72] called Droplets. DigitalOcean provides intuitive web interface for Cloud service hosting. This testbed provides a large amount of resources, which makes it suitable to carry out the experiments. Each VM had the following characteristics:

- Location: Frankfurt
- OS image: Ubuntu 22.04 (LTS) x64
- Numebr of CPUs: 2 vCPUs
- CPU memory: 4GB
- Disk type: SSD
- Disk memory: 80GB

The Droplets were accessed via a web browser application, called the Droplet Console [73]. It was considered as an alternative to using *ssh* command in a local terminal. The Droplet Console has a command-line user interface (Figure 3.2), similar to one in the computer terminal, and can be used to run commands on the Droplet. The advantages of using Droplet Console is a straightforward SSH access to the connected Droplet without the need for a password or manual SSH key configuration.

The Droplets that were accessed via the Droplet Console met the following requirements [73]:

- **The Droplet must be running a supported operating system.** The Droplet Console is supported only for DigitalOcean-provided Linux distributions: Ubuntu, Debian, CentOS and Fedora. The Droplets used for the experiments were running Ubuntu operating system.
- **The Cloud firewall and any host firewalls must accept SSH traffic on the port that *sshd* uses.** The configuration of every firewall should allow SSH traffic on the port the SSH daemon listens on. The default firewall configuration on Droplets and FaaS platforms allowed a correct connection.


ubuntu-s-2vcpu-4gb-fra1-01 - DigitalOcean Droplet Web Console — Mozilla Firefox
https://cloud.digitalocean.com/droplets/370840335/terminal/ui/

```
Welcome to Ubuntu 22.04.2 LTS (GNU/Linux 5.15.0-83-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Wed Sep 13 06:39:44 UTC 2023

  System load:                      0.0
  Usage of /:                       12.6% of 77.35GB
  Memory usage:                     22%
  Swap usage:                       0%
  Processes:                        121
  Users logged in:                  0
  IPv4 address for br-8a19cfe9e8c5: 172.19.0.1
  IPv4 address for docker0:         172.17.0.1
  IPv4 address for eth0:            139.59.146.75
  IPv4 address for eth0:            10.19.0.6
  IPv4 address for eth1:            10.114.0.3

 * Introducing Expanded Security Maintenance for Applications.
   Receive updates to over 25,000 software packages with your
   Ubuntu Pro subscription. Free for personal use.

     https://ubuntu.com/pro

Expanded Security Maintenance for Applications is not enabled.

59 updates can be applied immediately.
To see these additional updates run: apt list --upgradable

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status


The list of available updates is more than a week old.
To check for new updates run: sudo apt update

Last login: Sun Sep 10 19:59:14 2023 from 162.243.188.66
root@ubuntu-s-2vcpu-4gb-fra1-01:~#
```

Figure 3.2: Although the Droplet Console has a user interface similar to a computer terminal, it is a web-based app running on the Droplet user's browser.

The author was acknowledged of the possible latency caused by the architecture of the Droplet Console (Figure 3.3). However, as seen in Figure 3.4, the significant delay occurring during the communication between the local workstation and the DigitalOcean Droplet does not impact the time measured during the benchmarking experiment. The latency pattern between the Droplet and the FaaS platform would resemble the one that would occur between a local workstation and FaaS platform via *ssh* connection, and only that round-trip time (RTT) latency is measured during the benchmarking.

Prior to the installation on the virtual machines, the benchmark suits of interest were downloaded from their official GitHub repositories [74] [8]. GitHub is a web platform that is commonly used to host open source software development projects. It is also a Cloud-based service for software development and version control using Git [75], which contributes to its success as a platform for collaboration of researchers and developers.

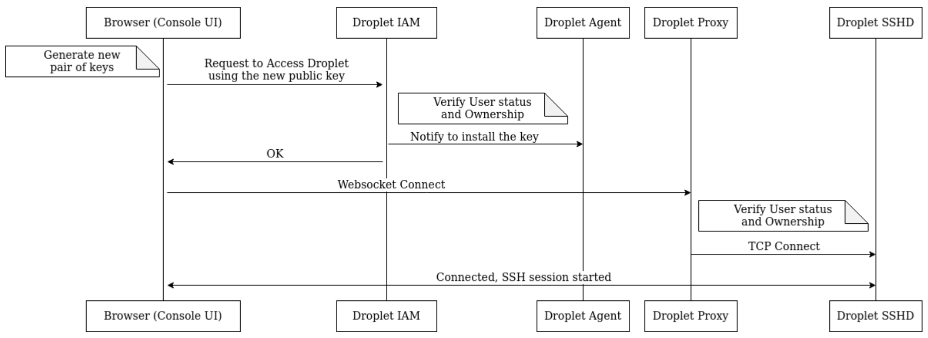

Browser (Console UI)
Droplet IAM
Droplet Agent
Droplet Proxy
Droplet SSHD
Generate new pair of keys
Request to Access Droplet using the new public key
Verify User status and Ownership
Notify to install the key
OK
Websocket Connect
Verify User status and Ownership
TCP Connect
Connected, SSH session started

Figure 3.3: Connection to a Droplet Console uses a proxy server to translate WebSocket connection to a raw TCP connection [10].

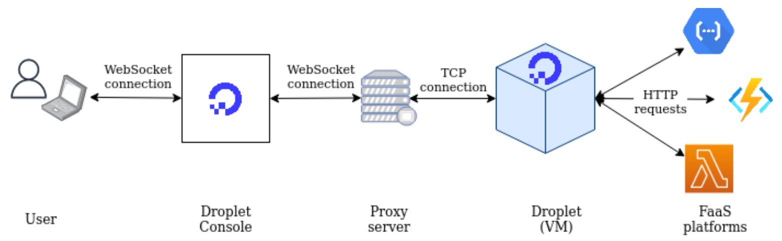


Figure 3.4: Experiment setup architecture focused on connection type between entities.

3.3 Execution

3.3.1 SeBS

The initial step entails the installation of dependencies. However, during the adherence to the installation guidelines as outlined in [76], it became apparent that the packages required for the libcurl library's headers were not explicitly stated in the documentation. Consequently, a more comprehensive installation guide was sought and identified in [77].

Furthermore, the setup process necessitated the creation of accounts on the respective Cloud provider's platforms and the acquisition of credentials, typically composed of a public and a private key pair, as instructed in [78].

To validate the connectivity of the tool with each Cloud service provider, this tool offers a set of regression tests. After verifying the correct interaction of the SeBS tool and the platforms, a performance measuring experiment was initiated in accordance with the methodology outlined in the original research paper [9]. Concurrent function invocations were performed, sampling to obtain 200 cold invocations, achieved by automated forced container eviction between each invocation batch. Next, the function executions were sampled to obtain 200 warm invocations. Function invocations were performed in batches of size 50 in order to include invocations in different sandboxes. The regions used for running experiments on Amazon Lambda, Azure Functions and Google Cloud Functions were us-east-1, west-europe and europe-west1 respectively.

perf-cost experiment was executed for the following benchmarks: *uploader*, *thumbnailer*, *compression* and *graph-bfs*. All functions subjected to benchmarking adhere to the Python programming language. Subsequently, all resulting data was meticulously processed and organized into CSV tables before being securely stored in a remote GitHub repository [70].

3.3.2 FaaSdom

The installation process for FaaSdom was straightforward, guided by the comprehensive installation manual [79], which provided step-by-step instructions for the installation of necessary dependencies. Similarly to the SeBS tool, the setup of FaaSdom tool entailed the creation of accounts on each respective Cloud provider's platform and the acquisition of credentials.

However, during the operational phase of the project, it became apparent that certain aspects required attention. Specifically, outdated versions of Docker images and the Node.js runtime were identified. Subsequently, when attempting to interact with the Cloud providers' services, issues arose, manifesting as either the non-display of webpages or difficulties in proper authentication. Resolution efforts involved the updating of the GCP CLI, Node.js, and Azure Docker images. Adjustments were also necessary in accordance with recent Azure functions API specifications to facilitate function deployment on the Azure platform. Following these updates, the program successfully established connections with the Cloud service providers, enabling function deployment across all target platforms.

After the execution of the latency experiment, an issue was uncovered concerning the integration with InfluxDB tables, leading to the inadvertent loss of experiment output data. Upon thorough examination of the source code, a decision was made to adopt a data processing approach similar to that employed in the SeBS experiments, involving the storage of data in the CSV format. Consequently, the utilization of Grafana for data visualization was foregone.

To gather performance data across different language runtimes and Cloud service providers, CPU bound Throughput/Latency experiment was performed involving *faas-matrix-mult* benchmark from FaaSdom toolkit. As in the original study, the experiment involved 10 manual function invocations for each provider/language runtime configuration. After performing time bound Latency experiment, the distributions of latencies was collected across different providers.

3.4 Data Analysis

3.4.1 SeBS

Figures 3.5-3.8 illustrate results obtained from SeBS experiments. Warm invocation function execution time (s) versus allocated memory (MB) box plots were constructed to verify the linear relationship stated in [9]. Azure Functions platform did not allow static instance memory allocation, and two rounds of experiment, 200 invocations each, were performed. Notably, the whiskers include data from the 2nd to 98th percentile.

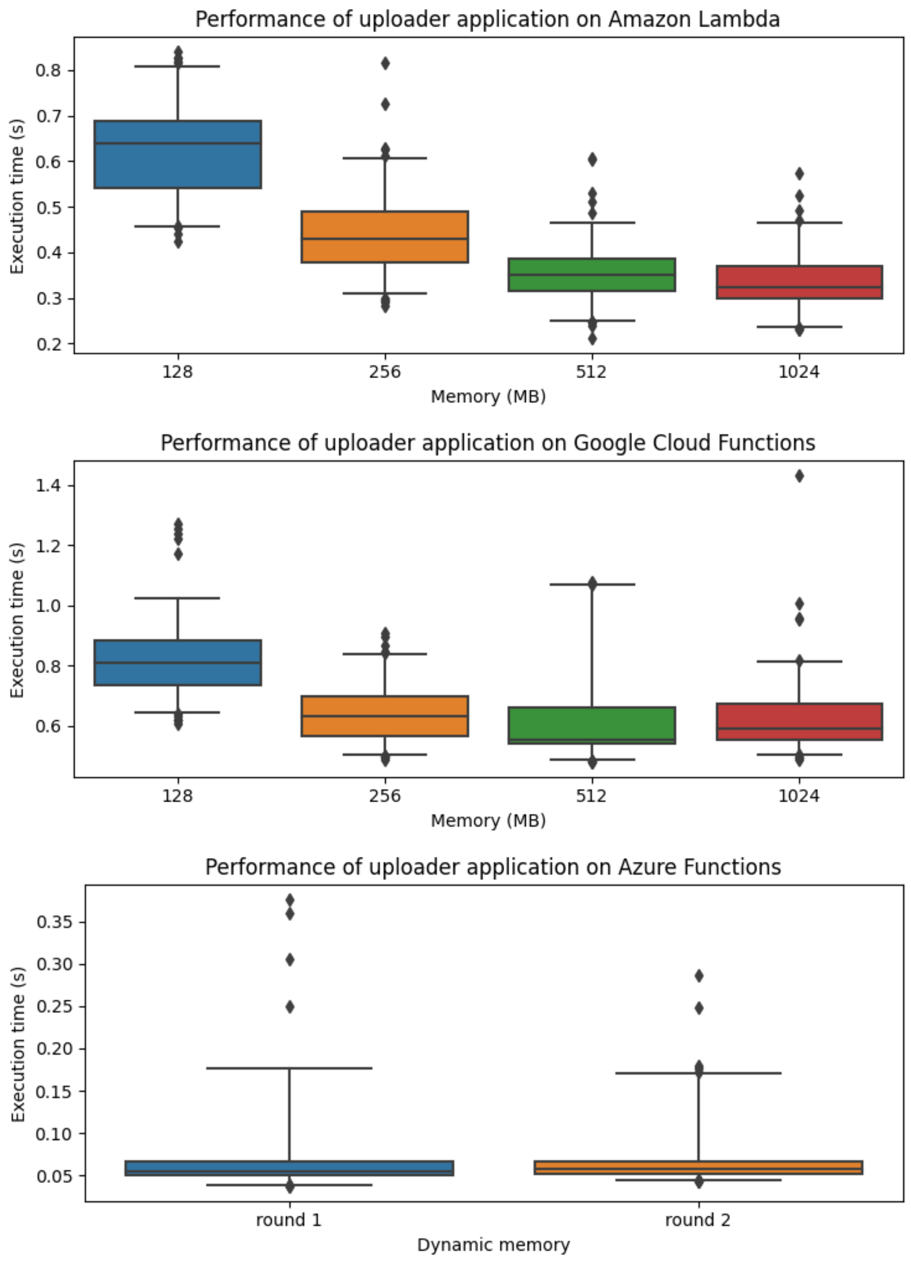


Figure 3.5: Execution time vs Allocated memory of SeBS *uploader* application on AWS Lambda, Google Cloud Functions, and Azure Functions.

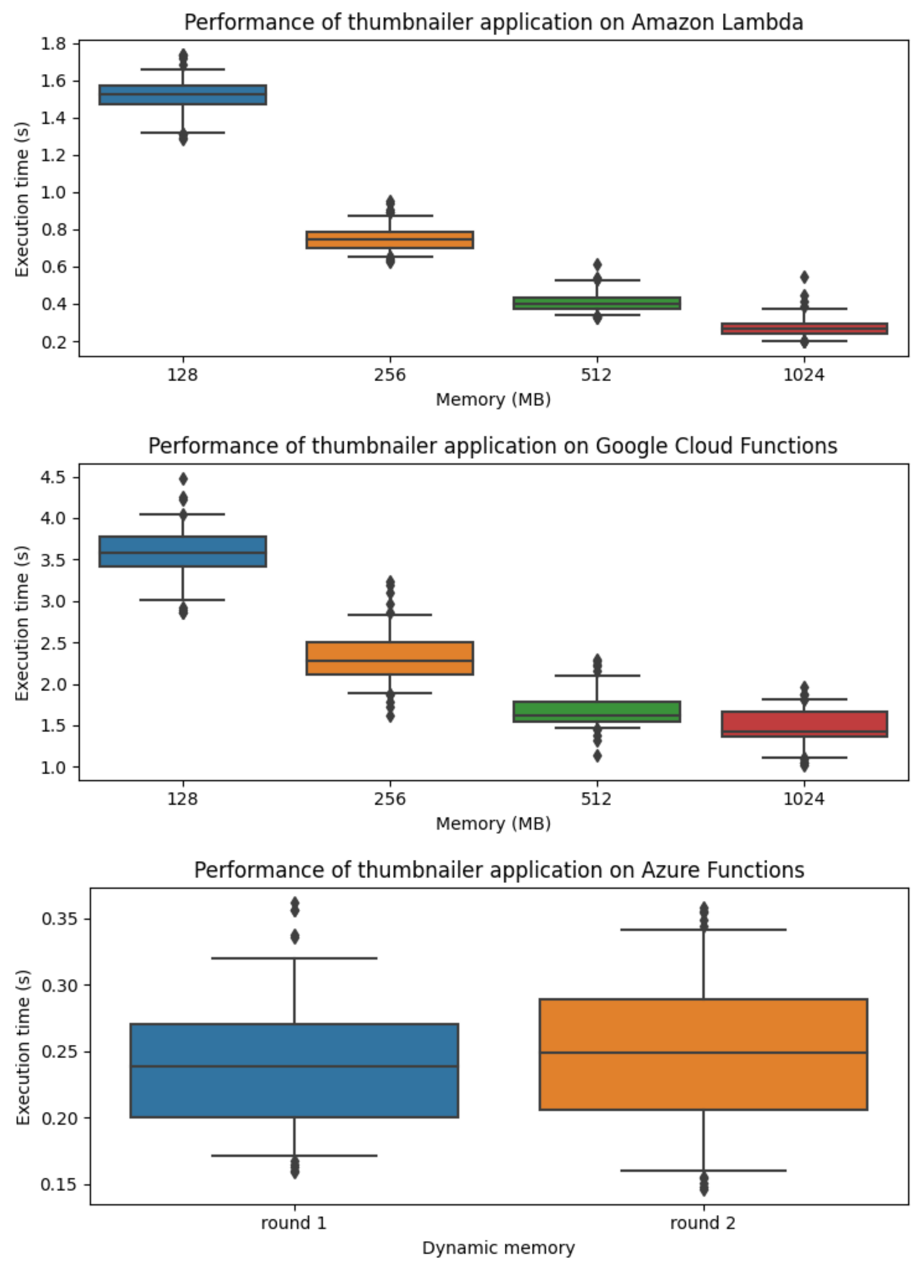


Figure 3.6: Execution time vs Allocated memory of SeBS *thumbnailer* application on AWS Lambda, Google Cloud Functions, and Azure Functions.

Figure 3.5 demonstrates the performance of the *uploader* application on AWS Lambda, Google Cloud Functions, and Azure Functions. Four memory configurations were tested (128 MB, 256 MB, 512 MB, and 1024 MB) on Amazon Lambda and Google Cloud Functions. Overall, the median execution time on Amazon Lambda is smaller for each configuration. Moreover, in both platforms, the execution time decreases gradually with the increase of memory. Another trend that should be mentioned is that all boxes have around the same range except for the 512 MB on Google Cloud Functions, which is more skewed towards the higher values. Regarding Azure Function, which has two rounds of experiments, the execution time was smaller compared to the previous two platforms. In both rounds, the distribution is the same, but the outliers of the first round of experiment were slightly further from the median.

Figure 3.6 illustrates the performance of the *thumbnailer* application. The same trend can be seen in that the time drops with the increase in memory, but the drop is sharper this time. The median execution time on Amazon is from 2 to 7 times smaller than one on Google. Also, the distribution of invocations on Google are twice as wide as on Amazon platform. Azure Functions has the lowest median execution time.

The performance of *compression* application is shown in Figure 3.7. The same patterns were observed in Figure 3.6, but in these experiments, the execution time is around 9 times longer on Amazon Lambda and Azure Functions, whereas it's about 6 times longer on Google Cloud Functions. Execution time on Azure Functions has a greater variability.

The *graph-bfs* applications performance is depicted in Figure 3.8. It can be clearly seen that the execution time doesn't depend on the memory for this application, and the median is approximately the same for all platforms. However, the outliers on Google Cloud Function are much further from the median time compared to the other platforms. In one case, the execution time took about 16 times more than the median time.

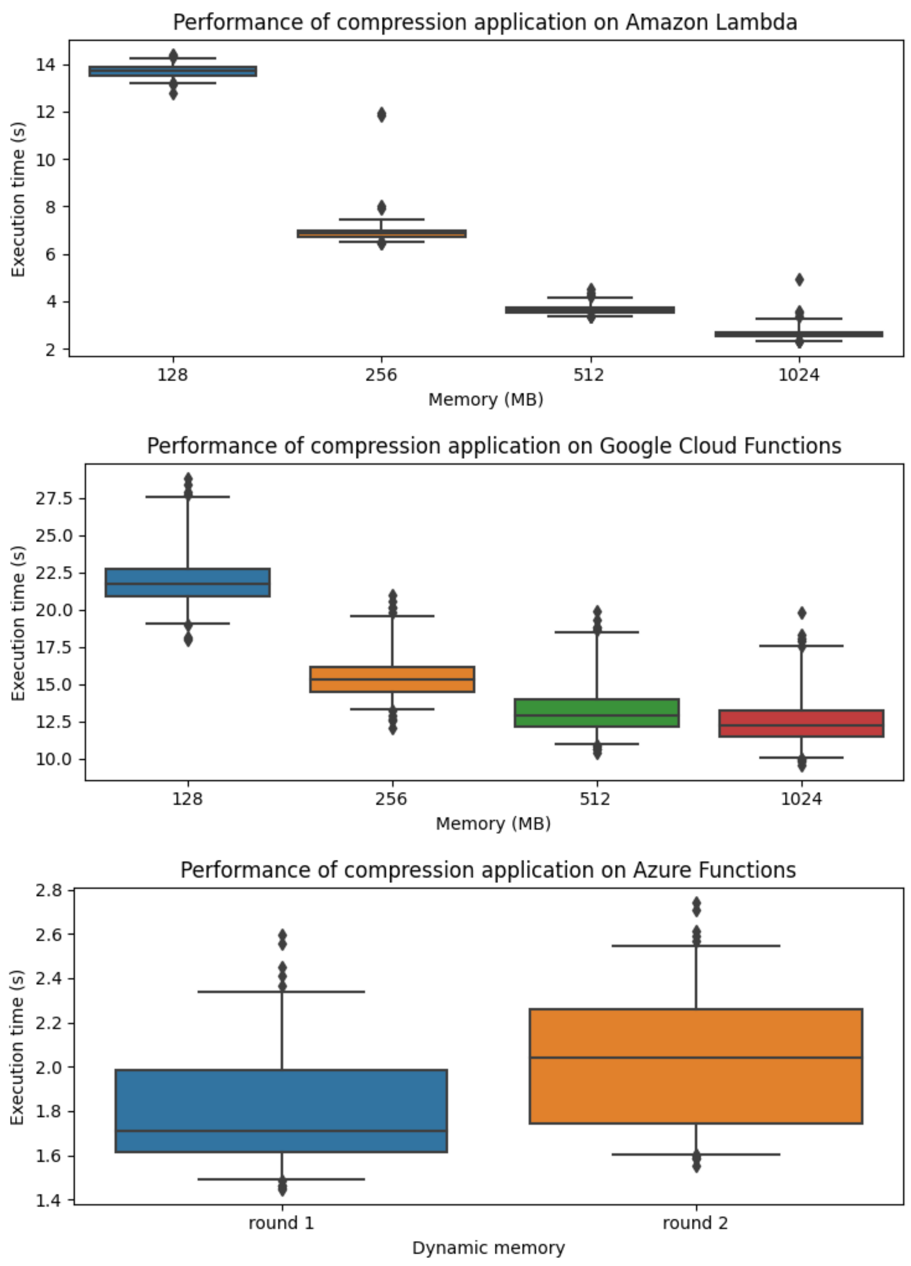


Figure 3.7: Execution time vs Allocated memory of SeBS *compression* application on AWS Lambda, Google Cloud Functions, and Azure Functions.

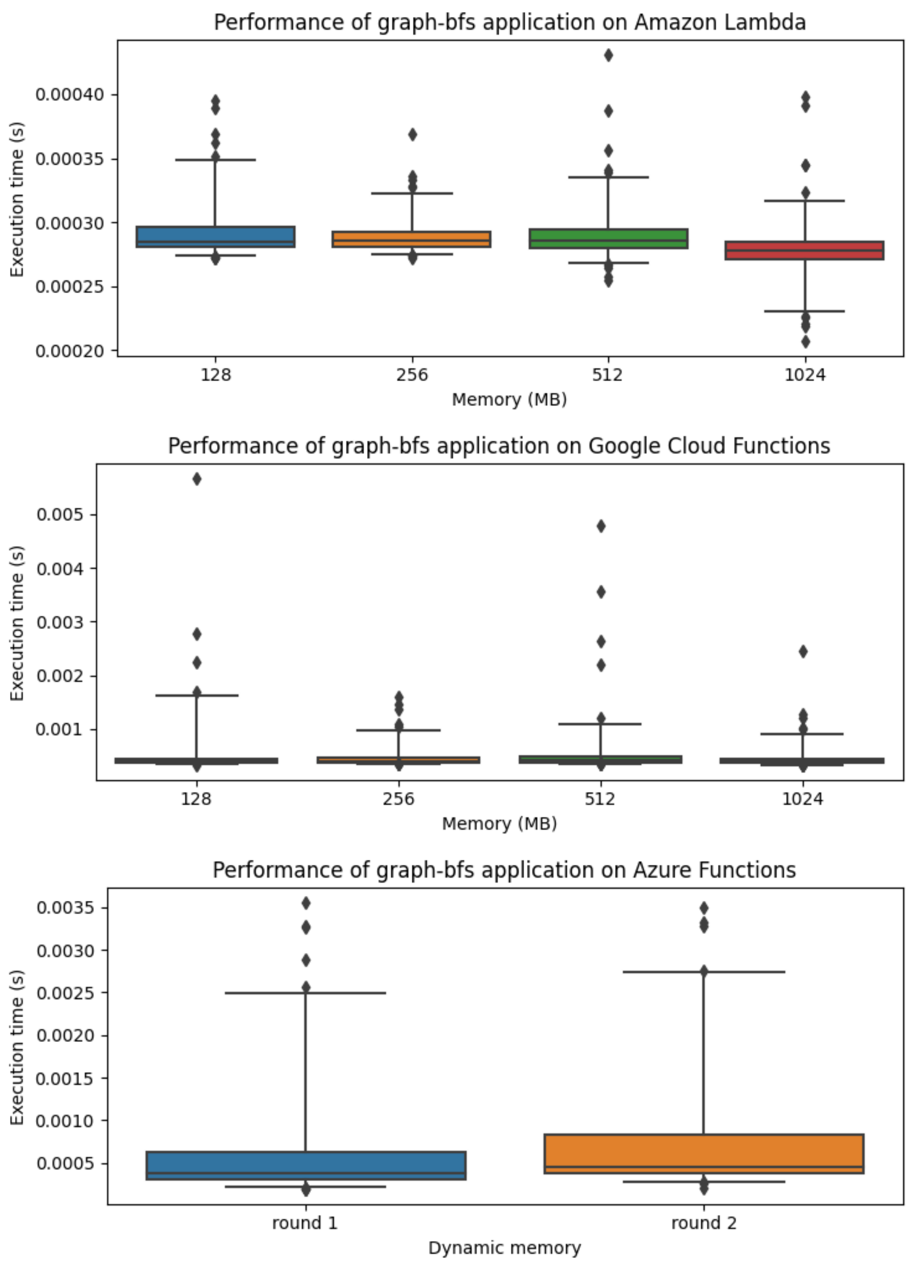


Figure 3.8: Execution time vs Allocated memory of SeBS *graph-bfs* application on AWS Lambda, Google Cloud Functions, and Azure Functions.

The results for the *compression* and *uploader* experiments ran on Azure Functions platform yielded a notable degree of dispersion within the dataset. Therefore, a decision to implement an outlier removal was made. The strategy of outlier removal was grounded in the calculation of z-scores. Outliers can affect an analysis in several ways. They can skew the data and affect the mean and standard deviation, making it difficult to obtain accurate estimates. Outliers can also affect the regression line and lead to incorrect predictions. Therefore, it is crucial to identify and exclude outliers before conducting any further analysis.

The z-score is a statistical measure that indicates how many standard deviations a data point is away from the mean. The z-score can be calculated using the following formula:

$$z = \frac{(x - mean)}{std}$$

where x is the data point, $mean$ is the mean of the dataset, and std is the standard deviation of the dataset.

To identify outliers using the z-score, one can set a threshold value. A z-score of 3 was chosen, meaning that more than 99% of data would be covered by the calculated interval. Consequently, any data point with a z-score greater than 3 or less than -3 was considered an outlier. SciPy library [80] was used to calculate the data points within the confidence interval and filter outliers. After removing outliers, the statistics in the graph for function invocations on Azure platform became more distinguishable.

In the pursuit of assessing the impact of cold starts on performance, [9] cold startup overhead was estimated as a ratio of every cold start benchmark execution time (N) to every warm start execution time (N), resulting in N^2 values. The plots (Figure 3.9) illuminate that cold start invocations can exhibit latencies up to 1.7 times greater than warm invocations. Interestingly, for functions with longer average runtimes, such as *compression*, the impact of cold starts appears negligible, with Azure demonstrating comparatively lower overheads when contrasted with the other two providers.

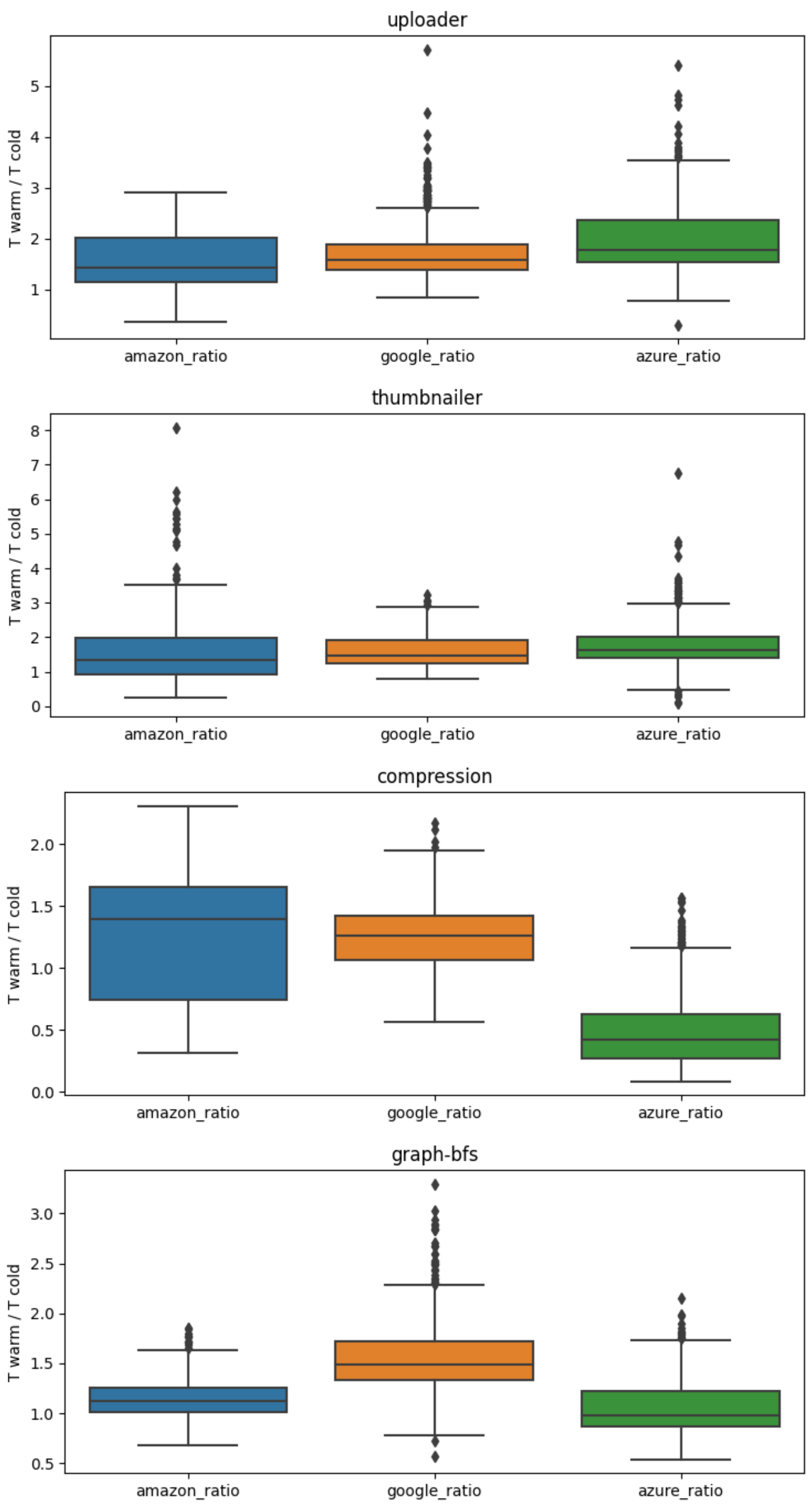


Figure 3.9: The ratio of cold start client time over warm start client time helps to see the impact of a cold start on an end-to-end response time.

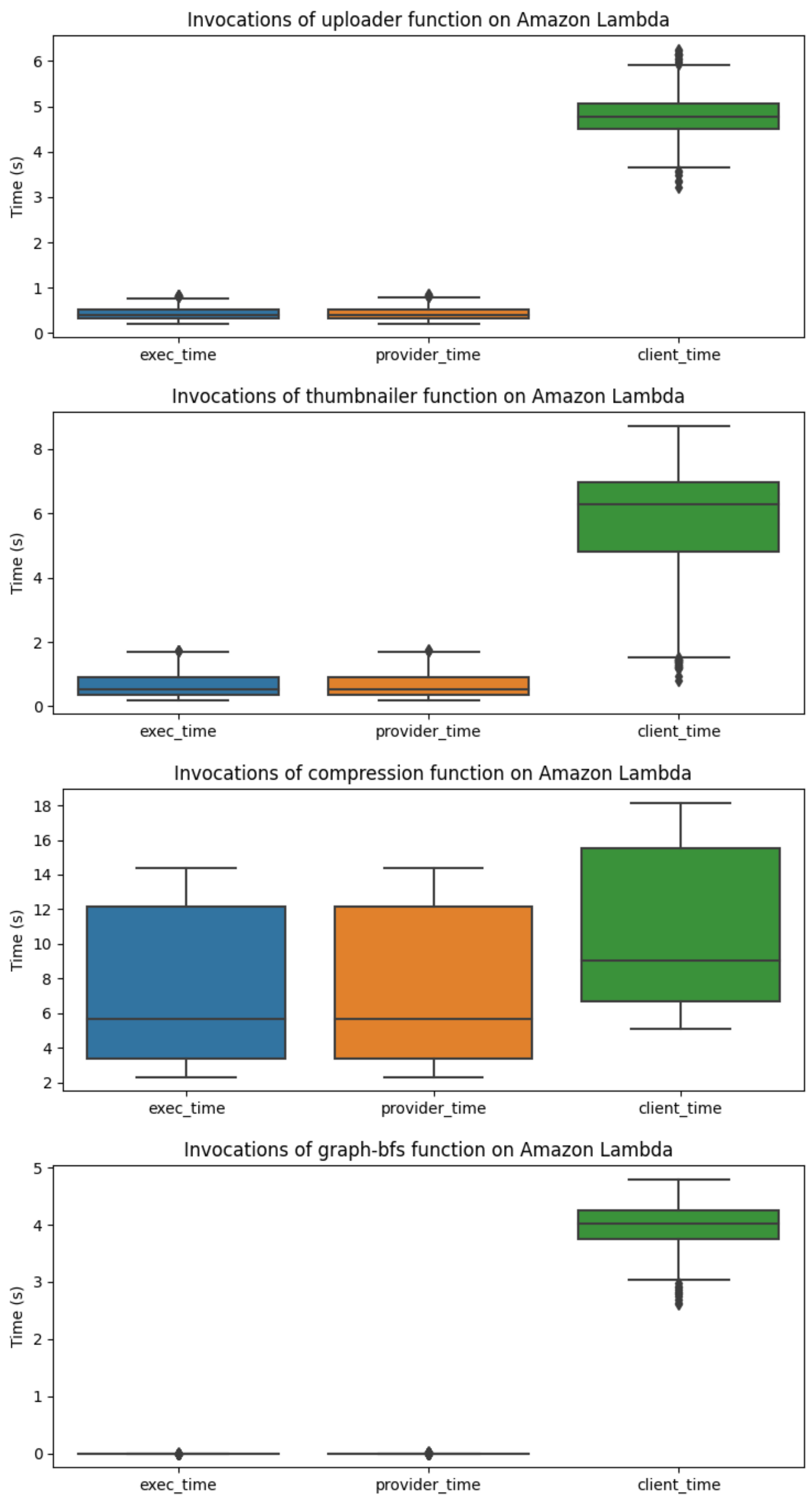


Figure 3.10: Benchmark execution time, provider time and client time for experiments executed on Amazon Lambda.

A differentiation is observed between execution time and client time across all service providers, as illustrated in Figure 3.10. This disparity is less pronounced for workloads characterized by longer execution times, such as compression. In this context, "benchmark time" refers to the execution time within the Cloud environment, excluding network and system latency. Conversely, "provider time" encompasses the execution time augmented by the additional overhead introduced by the language and the serverless sandbox of the provider. "Client time" denotes the measurement taken from the client's perspective. A substantial invocation overhead underscores the importance of providers offering comprehensive tools for sequential function invocation, ideally orchestrated entirely on the providers' side.

In summary, graphical representations of experiments performed using SeBS tool provide evidence that augmenting the allocated memory yields a reduction in execution time. Consequently, the presence of a linear relationship between system performance and allocated memory was validated. Remarkably, it was observed that Azure's performance surpasses the reported metrics from the 2021 study, reflecting an unexpected improvement in its operational efficiency. Moreover, Google platform exhibit significant variability in performance, while AWS platform yields the responses with the most predictable latency. The execution time records obtained from the Azure platform display a comparatively greater number of outliers when compared with the other two platforms.

In alignment with prior work [9], the analysis underscores the inherent variability in FaaS performance. The most significant variance is observed in the uploader and compression benchmarks, which rely heavily on I/O bandwidth. Long-running functions, exemplified by compression tasks, yield the highest number of performance outliers. In general, the experiments show a wide distribution of latencies, contributing to inconsistent and unpredictable performance of FaaS platforms.

3.4.2 FaaSdom

Figures 3.11, 3.14, 3.16, and 3.18 show the results for the Throughput/ Latency experiment performed using *faas-matrix-function* function written in 4 different programming languages: JavaScript, Python, Go, and .NET.

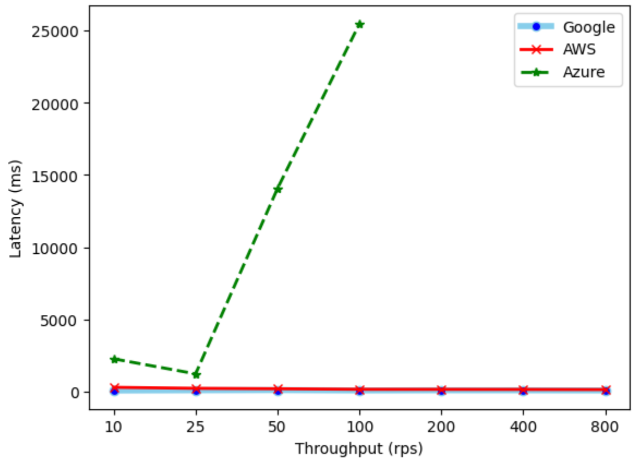


Figure 3.11: Throughput vs latency for Node.js *faasdom-matrix-mult* function.

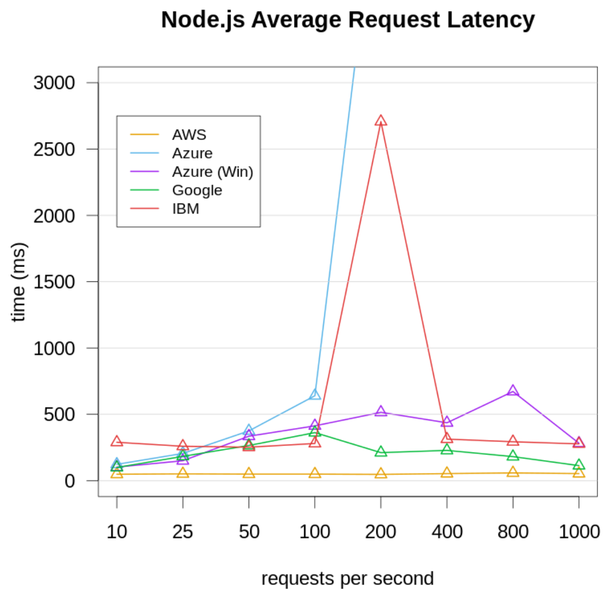


Figure 3.12: Throughput/latency graph for Node.js function based on the data from the original study.

In Figure 3.11 almost equal performance is achieved by a function running in Node.js runtime on Google Cloud Functions and AWS Lambda platform. A slight decrease in latency can be seen as the throughput increases, indicating that the performance increases with increased throughput. That may be caused by an effective instance allocation to handle the load, striking the balance between over-provisioning and under-provisioning of the virtual resources. Meanwhile, the latency on Azure Functions platform grows rapidly with the increasing throughput. The results for throughput higher than 100 requests per second were omitted from the graph, as the response latency quickly saturated. The result obtained during the recent experiment is slightly different from one presented in the original study. The latencies on Azure platform were reported to be much higher than was observed in 2020 (Figure 3.12).

Surprisingly, the Python function invocation on Azure platform has a somewhat steady performance (Figure 3.14) compared to the one observed in the original study (Figure 3.15). For Python language runtime, the latency of function executions on Azure and AWS platforms increases linearly, while the latency of function invocations on Google platform jumps unexpectedly at 100 requests per second. Furthermore, an optimal performance is reached at the low request rate, while increasing throughput leads to steady grows in latency until reaching a plateau at 800 requests per second. However, it was observed from the metrics dashboard on the Google Cloud Platform that the number of allocated instances did not remain constant, which otherwise could explain an increased delay at higher throughput rates (Figure 3.13).

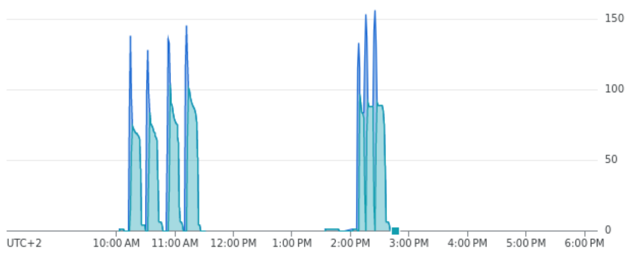


Figure 3.13: The number of instances on Google platform during Throughput/Latency experiment for Python language runtime.

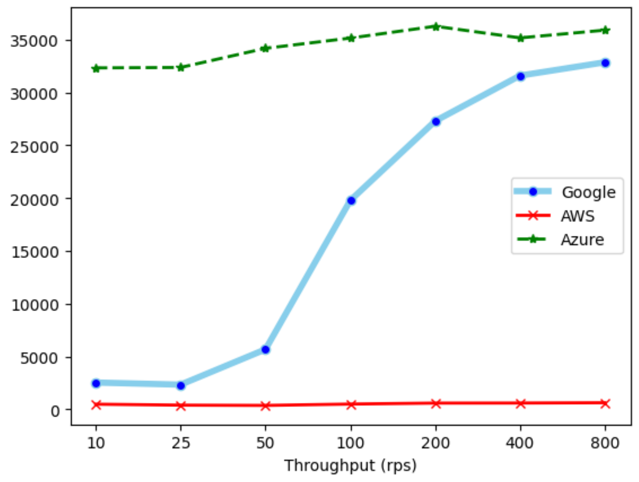


Figure 3.14: Throughput vs latency for Python *faasdom-matrix-mult* function.

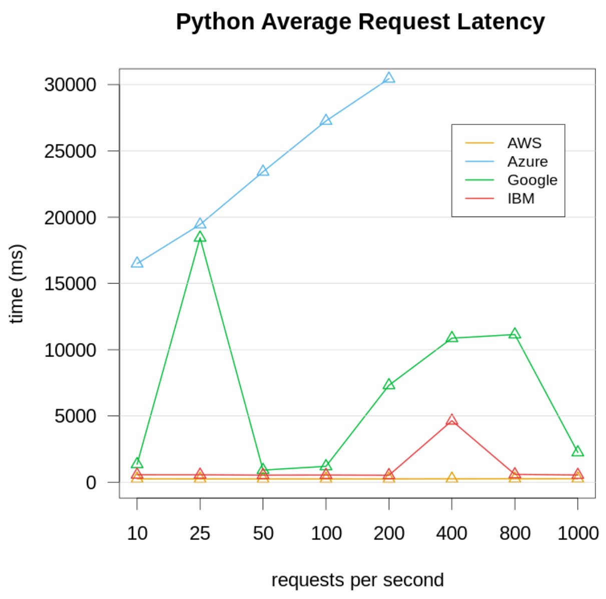


Figure 3.15: Throughput/latency graph for Python function based on the data from the original study.

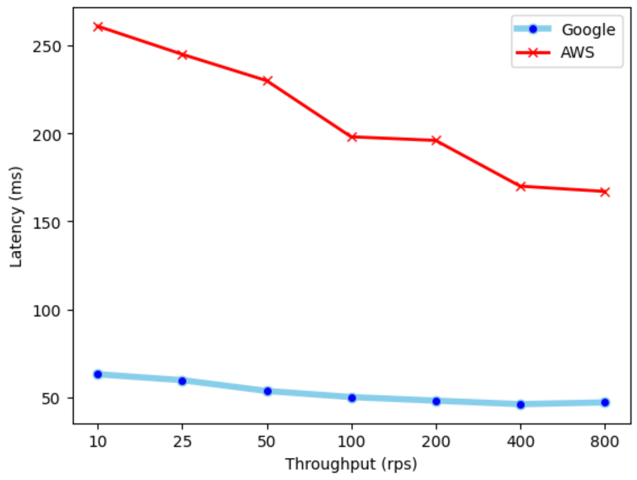


Figure 3.16: Throughput vs latency for Go *faasdom-matrix-mult* function.

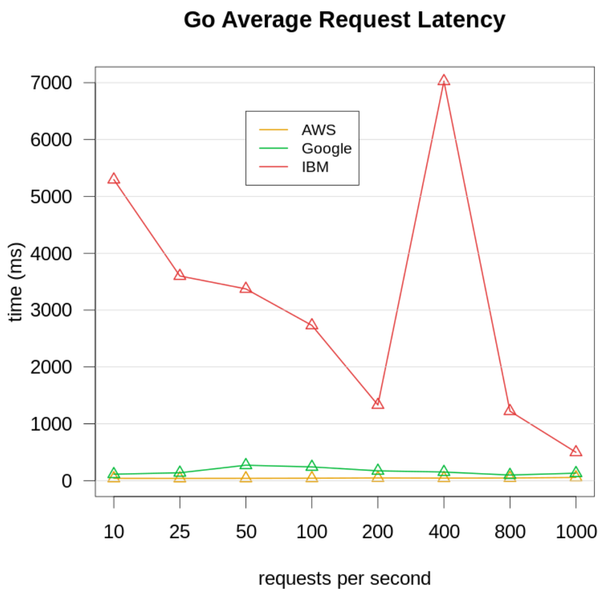


Figure 3.17: Throughput/latency graph for Go function based on the data from the original study.

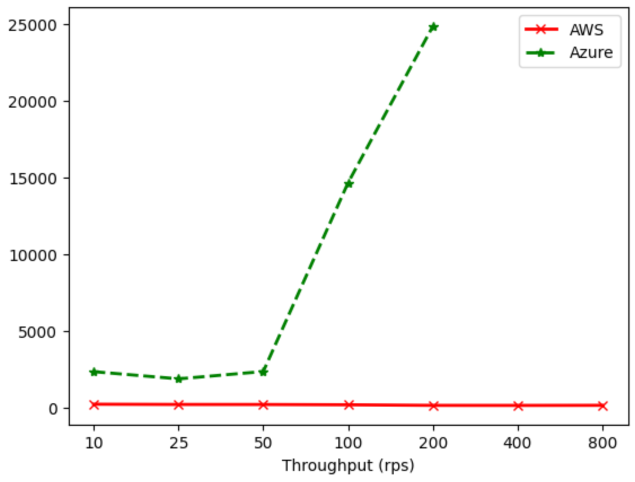


Figure 3.18: Throughput vs latency for .NET *faasdom-matrix-mult* function.

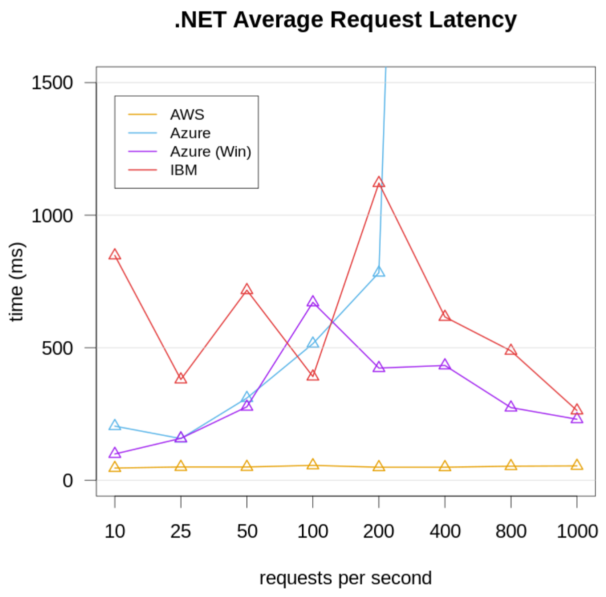


Figure 3.19: Throughput/latency graph for .NET function based on the data from the original study.

Notably, Go runtime is not supported on Azure platform, so we compared performance of Go function running on AWS and Google platform only. The obtained graph (Figure 3.16) represents a close-up look of the original graph without the latencies observed on IBM Cloud platform (Figure 3.17). Although both functions run on Google and AWS platform show increasing performance as the throughput increases, the former shows a much lower latency value for every value of throughput. However, the low latency may be caused by optimizations of Go runtime on Google platform, since Go language was originally developed by Google.

.NET runtime is not supported on Google platform, therefore the comparison was made between performance of .NET function on Azure and AWS platform (Figure 3.18). Azure platform demonstrated competitive response latency at throughput values below 50 requests per second. As the number of requests sent every second grew, the response latency grew linearly, reaching 25 seconds of delay at the rate of 200 requests per second.

3.5 Evaluation

3.5.1 SeBS

As reproduction of the experiments continued, the failure of automatic generation of Azure login credentials was discovered. As a result, a contribution to the open-source project repository was made by raising the issue on GitHub platform [65]. After discussion with the maintainer of the official repository, the conclusion was reached that the error occurred due to the updated Azure CLI interface. The main program flow was restored as the correct commands for interaction with Azure platform were inserted. Moreover, the Azure Functions configuration was updated so that it was possible to invoke the deployed function without authorization on the Azure platform. Two more bugs were found regarding integration with Azure platform, both of them were raised as issues on the GitHub platform [67, 66]. As of the time of writing, the issue with running regression tests on Azure platform was successfully resolved, while the issue of unsupported version of Python runtime was labelled as hardly reproducible.

Insights obtained regarding the performance of FaaS platforms, which include AWS Lambda, Google Cloud Functions, and Azure Functions were as followed:

- **Memory allocation impact on the function execution time**: The experiments confirm hat increasing the allocated memory results in a reduction in execution time. This validates the presence of a linear relationship between system performance and memory allocation.
- **Distinctive platform performance patterns**: It was observed that AWS Lambda consistently had the smallest median execution time across different memory configurations. In contrast, Google Cloud Functions and Azure Functions exhibited significant variability in performance.
- **Cold start impact on function execution time**: Cold start invocations showed latencies up to 1.7 times greater than warm invocations. The impact of cold starts was found to be negligible for functions with longer average runtimes.

Software component	**Version in source repository**	**Release date**	**Updated version**	**Release date**
Google Cloud SDK docker image	274.0.1-alpine	28 Dec 2019 [81]	400.0.0	30 Aug 2022 [82]
Node.js docker image	10.16.2-alpine	9 Aug 2019 [83]	18-alpine	11 Aug 2023 [84]
Azure CLI docker image	2.0.78	17 Dec 2019 [85]	2.52.0	5 Sep 2023 [85]
.NET runtime	dotnetcore2.1	30 May 2018 [86]	dotnet6	12 Sep 2023 [87]
Go runtime	go111	13 Aug 2019 [81]	go115	05 Aug 2021 [82]

Table 3.1: Software components and version migration performed to load FaaSdom tool.

- **Execution time differentiation**: There was a noticeable difference between execution time, provider time, and client time across all service providers. This disparity was less pronounced for workloads characterized by longer execution times.
- **Performance variability**: The experiments demonstrated a wide distribution of latencies across FaaS platforms, contributing to inconsistent and unpredictable performance. Azure Functions platform had a comparatively greater number of outliers in execution time records than Google Cloud Functions and Amazon Lambda platforms.

Performed experiments depict noteworthy disparities in warm invocations among service providers. Google platform ranks second after AWS, displaying notably higher performance levels, particularly in tasks related to image processing. Surprisingly, Azure platform performance increased compared to the one measured in 2021, sometimes exhibiting the most favourable performance, with execution times less than a second.

Furthermore, the ratio between cold start execution time and warm start execution time measured on client's device has reduced compared to the study conducted two years ago. This result underlines the significant impact of technological advancements on the obtained results and motivates the need for reproduction of benchmarking studies.

3.5.2 FaaSdom

A considerable presence of software components employing outdated versions was discovered. Table 3.1 provides an inventory of these software components along with the requisite version updates necessary to ensure the FaaSdom tool's functionality. The updated version of FaaSdom benchmarking tool is available at [71].

In general, due to its web-based user interface, utilizing the FaaSdom tool proved to be more user-friendly compared to SeBS. Nevertheless, several challenges were encountered during results collection and experiment execution. For instance, conducting cold start experiments necessitated the manual removal of deployed functions, as well as manual monitoring to confirm cold start and invocation of functions. Additionally, the web interface lacked clarity regarding the runtimes supported by cloud service providers, often indicating runtime incompatibility only during the actual experiment runs.

Similar to the results observed in the original study, stable response latencies are evident for all tested languages on AWS. Azure demonstrates the highest latency values for all language runtimes. When conducting a comparative analysis with the graphs presented in the original paper, a comparable behaviour is noted for Amazon Functions when operating on Node.js runtime. However, a distinct behaviour is documented for Python function invocations on the Google Functions platform, with latency increasing more rapidly. Nevertheless, the observed trend of superior performance at low request rates aligns with the findings highlighted in the reproduced study.

The summary of the key findings for each programming language and platform combination is as follows:

- **Node.js (JavaScript)**: Node.js functions on Google Cloud Functions and AWS Lambda demonstrated almost equal performance. An increase in throughput led to a slight decrease in latency of JavaScript functions, suggesting effective instance allocation. Azure Functions showed significantly higher latencies than in the original study.

- **Python**: Azure Functions had somewhat steady performance for Python runtime, unlike in the original study. Latency for Azure and AWS platforms increased linearly with throughput, while Google's latency jumped unexpectedly at 100 requests per second. Google Cloud Functions experienced fluctuations in the number of allocated instances, potentially explaining increased latency at higher throughput rates.

- **Go**: Go runtime was not supported on Azure. On Google Cloud Functions and AWS Lambda, Go functions showed increasing performance with higher throughput. Google Cloud Functions exhibited significantly lower latency for every throughput value.

- **.NET**: .NET runtime was not supported on Google Cloud Functions. Azure Functions showed competitive response latency at low throughputs, but latency grew linearly with increasing requests per second, reaching 25 seconds of delay at 200 requests per second.

Chapter 4

Conclusions and Outlook

4.1 Key Findings

In retrospect, it is crucial to revisit the fundamental questions posed in this study and evaluate whether they were successfully addressed. This study was conducted aiming to answer two questions:

- **Is it possible to reproduce the results of previous Cloud benchmarking studies?** While the installation of the benchmarking tools was initially hindered by the abundance of outdated dependencies, once those dependencies were updated and fine-tuned with accordance to the most recent usage guidelines, reproducing benchmarking experiments became considerably more straightforward. After analysing the outputs of the benchmarking experiments, all the statements outlined in reproduced studies regarding memory allocation impact on the function execution time, prominent platform performance patterns, cold start impact on function execution time, and performance variability were validated.

  In general, the experiments confirmed hat increasing the allocated memory results in a reduction in execution time. AWS Lambda platform consistently yielded the smallest median execution time across different memory configurations for function executions, while Google Cloud Functions and Azure Functions exhibited significant variability. Cold start invocations showed latencies up to 1.7 times greater than warm invocations. The workloads characterized by longer execution times were less impacted by cold start invocations and showed less difference between execution time, provider time, and client time across all service providers. Azure Functions platform was observed to have a greater number of outliers in execution time records compared to Google Cloud Functions and Amazon Lambda platforms.

- **What are the main factors contributing to the reproducibility of Cloud benchmarking results?** Since the number of encountered obstacles varied significantly depending on the current maintenance status of the tools used in the original studies, it was concluded that the pivotal point in reproducing benchmarking studies was the community support and original researcher's interest in discussing their publications. In the experiments, two FaaS benchmarking tools were used, namely SeBS and FaaSdom. Both tools are open-source, however SeBS tool needed fewer interventions during the experiment execution process, which speeded up the process of reproduction. The time left was used for discussion with the co-author of the scientific paper and extension of the tool by contributing to the open-source code.

When comparing the present state of FaaSdom application to the time of the last contribution by the original authors, it became evident that the tool needs supervision and maintenance to adapt to the ever-changing landscape of technological advancements. Thus, the supervision and the community's interest were established as the primary factors that contribute to the reproducibility of Cloud benchmarking studies.

Finally, the research proposes practical solutions and guidelines that could be implemented to strengthen the reproducibility of Cloud benchmarking research. Reproducible studies tend to have the following characteristics:

- **Open-source codebase**: Making the source code openly available empowers other researchers and developers to contribute to the project. This can be useful in several ways. First, contributors will be able to identify the errors and areas for improvement. Second, the tool will grow in terms of its abilities as developers and researchers will find and elaborate on new use cases, thus naturally extending the tool.

- **Extensive documentation**: At the state of development, when a few collaborators are involved, the implementation details can be shared via personal communication. As the project attracts new collaborators and contributors, comprehensive documentation of usage and installation procedures becomes crucial, serving as a serve as starting point and a guide. The tool will benefit from comprehensive documentation of its usage and installation procedures, simplifying the maintenance for the original authors.

- **Code structure that isolates dependencies**: By isolating dependencies, one can make sure that the monitoring and maintenance of dependencies and their versions will be efficient. One of the main sources of frustration for developers can be dispersed and hardcoded version specification for various dependencies occurring in numerous code components.

4.2 Future Work

This thesis focused on evaluating and comparing the performance of serverless computing platforms, specifically Amazon Lambda, Google Cloud Functions, and Azure Functions, and successfully validated the key finding of two benchmarking studies: "FaaSdom: A benchmark suite for serverless computing" by Maissen et al. [42] and "SeBS: A serverless benchmark suite for function-as-a-service computing" by Copik et al. [9]. This study can be further extended, by focusing on the alternative aspects of FaaS benchmarking.

In the performed experiments, the evaluation of Go runtime on Azure platform and .NET runtime on GCP was omitted, since Azure platform doesn't possess a built-in a Go runtime, and .NET runtime is absent on GCP. However, some of prominent Cloud platforms recently introduced a support for custom language runtimes [88]. Therefore, a new study could be conducted by extending the available array of language runtimes with the custom ones to obtain more data related to the performance of various language runtimes on mentioned Cloud platforms.

Another strategy would be to assess more FaaS benchmarking tools, such as BeFaaS [60] and FunctionBench [59]. By reproducing more benchmarking studies and making a comparative analysis of the obtained results, it may be possible to identify common patters among benchmarking tools that were produced as a part of scientific research endeavours, thereby contributing to the enhancement of FaaS benchmarking studies' reproducibility and utility.

The subsequent phase of the study may broaden by researching expressivity of the benchmarks that are currently available. As a part of the study, new use cases may be found, resulting in the creation of new benchmarks. Particularly interesting fields of application of FaaS is IoT, so designing new benchmarks to represent Smart Home or health monitoring application workloads could be a promising area of research. That study would also allow extending existing benchmark suites, such as SeBS.

Including more cloud providers, including open-source Cloud platform such as OpenWhisk is regarded as a possible continuation trajectory of the study. Benchmarks may also be used for reverse engineering a better Cloud architectures for a specific use cases represented by the respective workloads.

Bibliography

[1] eurostat. (2021) File: Use of cloud computing services, 2020 and 2021. Accessed on September 10, 2023. [Online]. Available: https://ec.europa.eu/eurostat/statistics-explained/images/6/65/Use_of_cloud_computing_services%2C_2020_and_2021_%28%25_of_enterprises%29_v2.png

[2] J. Allen. (2021) 7 Strategies for Migrating Applications to the Cloud, introducing AWS Mainframe Modernization and AWS Migration Hub Refactor Spaces. Accessed on September 11, 2023. [Online]. Available: https://aws.amazon.com/blogs/enterprise-strategy/cloud-native-or-lift-and-shift/

[3] The Johns Hopkins Data Science Lab. (2022) Online course: Intro to reproducibility in cancer informatics. chapter 2: Defining reproducibility. Accessed on September 12, 2023. [Online]. Available: https://jhudatascience.org/Reproducibility_in_Cancer_Informatics/defining-reproducibility.html

[4] F. Richter. (2023) Amazon maintains lead in the cloud market. Accessed on September 11, 2023. [Online]. Available: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/

[5] G. Gireesh. (2021) Saas vs paas vs iaas: What's the difference and how do you choose? Accessed on September 5, 2023. [Online]. Available: https://www.liquidweb.com/kb/saas-paas-iaas/

[6] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Isahagian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, *Serverless Computing: Current Trends and Open Problems*, 12 2017, pp. 1–20.

[7] IBM Technology. (2021) Youtube lecture: What is faas (functions as a service)? Accessed September 8, 2023. [Online]. Available: https://youtu.be/EOIja7yFScs?si=dUPfJ1_88LwPJNpY

[8] faas-benchmarking. (2020, Apr.) The FAASDOM benchmark suite. [Online]. Available: https://github.com/faas-benchmarking/faasdom

[9] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 64–78.

[10] H. Li. (2022) How digitalocean's new droplet console works. Accessed on September 12, 2023. [Online]. Available: https://www.digitalocean.com/blog/how-digitaloceans-new-droplet-console-works

[11] C. Fisher, "Cloud versus on-premise computing," *American Journal of Industrial and Business Management*, vol. 08, pp. 1991–2006, 01 2018.

[12] TutorialsPoint. (2023) Cloud computing overview. Accessed on September 1, 2023. [Online]. Available: https://www.tutorialspoint.com/cloud_computing/cloud_computing_overview.htm

[13] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[14] Gartner. (2023) Gartner peer insights: Cloud infrastructure and platform services. Accessed on September 1, 2023. [Online]. Available: https://www.gartner.com/reviews/market/cloud-infrastructure-and-platform-services

[15] J. Manner, "Towards performance and cost simulation in function as a service," 2019.

[16] C# Corner. (2023) Top 10 cloud service providers. Accessed on September 1, 2023. [Online]. Available: https://www.c-sharpcorner.com/article/top-10-cloud-service-providers/

[17] Pluralsight. (2023) Compute compared: Aws vs. azure vs. gcp. Accessed on September September 2, 2023. [Online]. Available: https://www.pluralsight.com/resources/blog/cloud/compute-compared-aws-vs-azure-vs-gcp

[18] N2WS. Aws vs. azure vs. google cloud comparison. Accessed on September 2, 2023. [Online]. Available: https://n2ws.com/blog/aws-vs-azure-vs-google-cloud

[19] Cockroach Labs. (2020) Aws, azure, and gcp respond to the 2020 cloud report. Accessed on September 2, 2023. [Online]. Available: https://www.cockroachlabs.com/blog/aws-azure-gcp-respond-to-the-2020-cloud-report/

[20] ——. (2021) 2021 Cloud Report. Accessed on September 2, 2023. [Online]. Available: https://www.cockroachlabs.com/blog/2021-cloud-report/

[21] ——. (2022) 2022 cloud report. Accessed on September 2, 2023. [Online]. Available: https://www.cockroachlabs.com/blog/2022-cloud-report/

[22] C. Vazquez, R. Krishnan, and E. John, "Cloud computing benchmarking: a survey," in *Proceedings of the international conference on grid, cloud, and cluster computing (GCC)*. The Steering Committee of The World Congress in Computer Science, 2014, p. 1.

[23] Hewlett Packard. (2023) Netprf GitHub Repository. Accessed on September 2, 2023. [Online]. Available: https://github.com/HewlettPackard/netperf

[24] Intel. (2023) HiBench GitHub Repository. Accessed on September 2, 2023. [Online]. Available: https://github.com/Intel-bigdata/HiBench

[25] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," 09 2010, pp. 143–154.

[26] EPFL. (2023) CloudSuite GitHub Repository. Accessed on September 2, 2023. [Online]. Available: https://github.com/parsa-epfl/cloudsuite/tree/main

[27] Google. (2023) PerfKit Benchmarker GitHub Repository. Accessed on September 2, 2023. [Online]. Available: https://github.com/GoogleCloudPlatform/PerfKitBenchmarker

[28] M. Ficco, M. Rak, S. Venticinque, L. Tasquier, and G. Aversano, *Cloud Evaluation: Benchmarking and Monitoring*, 04 2015, pp. 175–200.

[29] Yahoo. (2010) YCSB GitHub Repository. Accessed on September 2, 2023. [Online]. Available: https://github.com/brianfrankcooper/YCSB

[30] A. E. Carroll, "Publication bias: The threat to science," *The New York Times*, 2018, accessed on September 2, 2023. [Online]. Available: https://www.nytimes.com/2018/09/24/upshot/publication-bias-threat-to-science.html

[31] M. Baker, "1,500 scientists lift the lid on reproducibility," *Nature*, vol. 533, no. 7604, 2016.

[32] G. J. Lithgow, M. Driscoll, and P. Phillips, "A long journey to reproducible results," *Nature*, vol. 548, no. 7668, pp. 387–388, 2017.

[33] C. G. Begley and J. P. Ioannidis, "Reproducibility in science: improving the standard for basic and preclinical research," *Circulation research*, vol. 116, no. 1, pp. 116–126, 2015.

[34] D. Randall and C. Welser, *The Irreproducibility Crisis of Modern Science: Causes, Consequences, and the Road to Reform.* ERIC, 2018.

[35] M. Serra-Garcia and U. Gneezy, "Nonreplicable publications are cited more than replicable ones," *Science Advances*, vol. 7, no. 21, p. eabd1705, 2021. [Online]. Available: https://www.science.org/doi/abs/10.1126/sciadv.abd1705

[36] T. Miyakawa, "No raw data, no science: another possible source of the reproducibility crisis," pp. 1–6, 2020.

[37] Papers With Code. (2022) Ml reproducibility challenge 2022. Accessed on September 4, 2023. [Online]. Available: https://paperswithcode.com/rc2022

[38] EDS book. (2023) Reproducibility challenge 2023. Accessed on September 4, 2023. [Online]. Available: https://eds-book.github.io/reproducibility-challenge-2023/intro.html

[39] International Conference on Learning Representations. (2019) Iclr 2019 reproducibility challenge. Accessed on September 4, 2023. [Online]. Available: https://www.cs.mcgill.ca/~jpineau/ICLR2019-ReproducibilityChallenge.html

[40] Kaggle. (2022) Kaggle awards for ml reproducibility challenge 2022. Accessed on September 4, 2023. [Online]. Available: https://www.kaggle.com/reproducibility-challenge-2022

[41] The International Conference for High Performance Computing, Networking, Storage, and Analysis. (2022) Reproducibility initiative. Accessed on September 4, 2023. [Online]. Available: https://sc22.supercomputing.org/submit/reproducibility-initiative/

[42] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, "Faasdom: A benchmark suite for serverless computing," in *Proceedings of the 14th ACM international conference on distributed and event-based systems*, 2020, pp. 73–84.

[43] P. Mell and T. Grance, "The nist definition of cloud computing," Sep 2011.

[44] B. King. (2022) What is faas? function as a service explained. Accessed on September 12, 2023. [Online]. Available: https://www.digitalocean.com/blog/what-is-faas-function-as-a-service-explained

[45] IBM. (2020) What is serverless? Accessed on September 12, 2023. [Online]. Available: https://www.ibm.com/topics/serverless

[46] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of The ACM*, Nov 2019.

[47] AWS, Amazon Web Services. Serverless on aws. Accessed September 1, 2023. [Online]. Available: https://aws.amazon.com/serverless/

[48] Google. Serverless computing. Accessed September 1, 2023. [Online]. Available: https://cloud.google.com/serverless

[49] Microsoft Azure. Azure serverless. Accessed September 1, 2023. [Online]. Available: https://azure.microsoft.com/en-us/solutions/serverless/#solutions.

[50] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?" *IEEE Software*, vol. 38, no. 1, pp. 32–39, 2021.

[51] S. Eismann, J. Scheuner, E. v. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4152–4166, 2022.

[52] Microsoft. Azure functions http trigger. Accessed September 8, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook-trigger?tabs=python-v2%2Cin-process%2Cfunctionsv2&pivots=programming-language-python

[53] Amazon. Supported Languages at AWS Lambda. Accessed September 8, 2023. [Online]. Available: https://blog.awsfundamentals.com/supported-languages-at-aws-lambda

[54] ——. 11 most in-demand programming languages. Accessed September 8, 2023. [Online]. Available: https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages/

[55] Microsoft. Supported languages in azure functions. Accessed September 1, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/supported-languages#languages-by-runtime-version/

[56] ——. Announcing general availability of azure functions. Accessed September 1, 2023. [Online]. Available: https://azure.microsoft.com/en-us/blog/announcing-general-availability-of-azure-functions/

[57] E. v. Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, "A spec rg cloud group's vision on the performance challenges of faas cloud architectures," *International Conference on Performance Engineering*, Apr 2018.

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

[59] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 502–504.

[60] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, and D. Bermbach, "Befaas: An application-centric benchmarking framework for faas platforms," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2021, pp. 1–8.

[61] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," *Journal of Systems and Software*, vol. 170, p. 110708, 2020.

[62] Jeffrey A. Clark. Python pillow library. Accessed September 8, 2023. [Online]. Available: https://pillow.readthedocs.io/en/stable/

[63] G. Tene. (2015) wrk2 github repository. Accessed on September 12, 2023. [Online]. Available: https://github.com/giltene/wrk2

[64] W. Glozer. (2015) wrk github repository. Accessed on September 12, 2023. [Online]. Available: https://github.com/wg/wrk

[65] S. Nursultan. (2023) Issue discussion on github: Azure credential creation script doesn't work. Accessed on September 7, 2023. [Online]. Available: https://github.com/spcl/serverless-benchmarks/issues/176

[66] ——. (2023) Issue discussion on github: Azure supports only python 3.7. Accessed on September 7, 2023. [Online]. Available: https://github.com/spcl/serverless-benchmarks/issues/177

[67] ——. (2023) Issue discussion on github: Regression tests are not passing for azure. Accessed on September 7, 2023. [Online]. Available: https://github.com/spcl/serverless-benchmarks/issues/178

[68] ——. (2023) Issue discussion on github: Documentation for outputted results. Accessed on September 13, 2023. [Online]. Available: https://github.com/spcl/serverless-benchmarks/issues/180

[69] Google. (2023) Welcome to colaboratory. Accessed on September 12, 2023. [Online]. Available: https://colab.research.google.com/?utm_source=scs-index

[70] S. Nursultan. (2023) Fork of SeBS GitHub repository. Accessed on September 7, 2023. [Online]. Available: https://github.com/nurSaadat/sebs

[71] ——. (2023) Fork of faasdom project. Accessed on September 9, 2023. [Online]. Available: https://github.com/nurSaadat/faasdom

[72] DigitalOcean. (2023) Digitalocean official website. Accessed on September 7, 2023. [Online]. Available: https://www.digitalocean.com/

[73] DigitalOcean. (2021) How to connect to droplets with the droplet console. Accessed on September 12, 2023. [Online]. Available: https://docs.digitalocean.com/products/droplets/how-to/connect-with-console/

[74] M. Copik. (2021, Jul.) SeBS: serverless benchmarks suite. [Online]. Available: https://github.com/spcl/serverless-benchmarks

[75] Wired. (2015) The problem with putting all the world's code in github. Accessed on September 7, 2023. [Online]. Available: https://www.wired.com/2015/06/problem-putting-worlds-code-github/

[76] M. Copik. (2023) Sebs installation. Accessed on September 6, 2023. [Online]. Available: https://github.com/spcl/serverless-benchmarks#installation

[77] C. Maharjan. (2022) Evaluating serverless computers. Accessed on September 6, 2023. [Online]. Available: https://repository.lsu.edu/gradschool_theses/5648/

[78] M. Copik. (2023) Sebs cloud platforms configuration. Accessed on September 6, 2023. [Online]. Available: https://github.com/spcl/serverless-benchmarks/blob/master/docs/platforms.md

[79] V. Schiavoni. (2021) Faasdom installation. Accessed on September 6, 2023. [Online]. Available: https://github.com/faas-benchmarking/faasdom#install

[80] SciPy. Official scipy website. Accessed September 9, 2023. [Online]. Available: https://scipy.org/

[81] Docker Hub. (2019) Image layer details: google/cloud-sdk:274.0.1-alpine. Accessed on September 13, 2023. [Online]. Available: https://hub.docker.com/layers/google/cloud-sdk/274.0.1-alpine/images/sha256-cc824eeb6355cdc59cd3dad705fdd6899d0a137154a68025df5598739f8c422f?context=explore

[82] ——. (2022) Image layer details: google/cloud-sdk:400.0.0. Accessed on September 13, 2023. [Online]. Available: https://hub.docker.com/layers/google/cloud-sdk/400.0.0/images/sha256-d7d89de58ed3a72a623736676a5106488052a720f073ba281bb20bfb9d9d7aee?context=explore

[83] ——. (2019) Image layer details: node:10.16.2-alpine. Accessed on September 13, 2023. [Online]. Available: https://hub.docker.com/layers/library/node/10.16.2-alpine/images/sha256-f39c6470aa6d6468484fd38a76066632ed514c2234afa91c144b6511d3c52f22?context=explore

[84] ——. (2023) Image layer details: node:18-alpine. Accessed on September 13, 2023. [Online]. Available: https://hub.docker.com/layers/library/node/18-alpine/images/sha256-982b5b6f07cd9241c9ebb163829067deac8eaefc57cfa8f31927f4b18943d971?context=explore

[85] Microsoft. (2023) Azure cli release notes. Accessed on September 13, 2023. [Online]. Available: https://learn.microsoft.com/en-us/cli/azure/release-notes-azure-cli

[86] ——. (2021) Download .net core 2.1. Accessed on September 13, 2023. [Online]. Available: https://dotnet.microsoft.com/en-us/download/dotnet/2.1

[87] Google. (2023) .net github repository. Accessed on September 13, 2023. [Online]. Available: https://github.com/dotnet/core/blob/main/release-notes/6.0/6.0.22/6.0.22.md

[88] Amazon. (2023) Custom lambda runtimes. Accessed on September 14, 2023. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/runtimes-custom.html