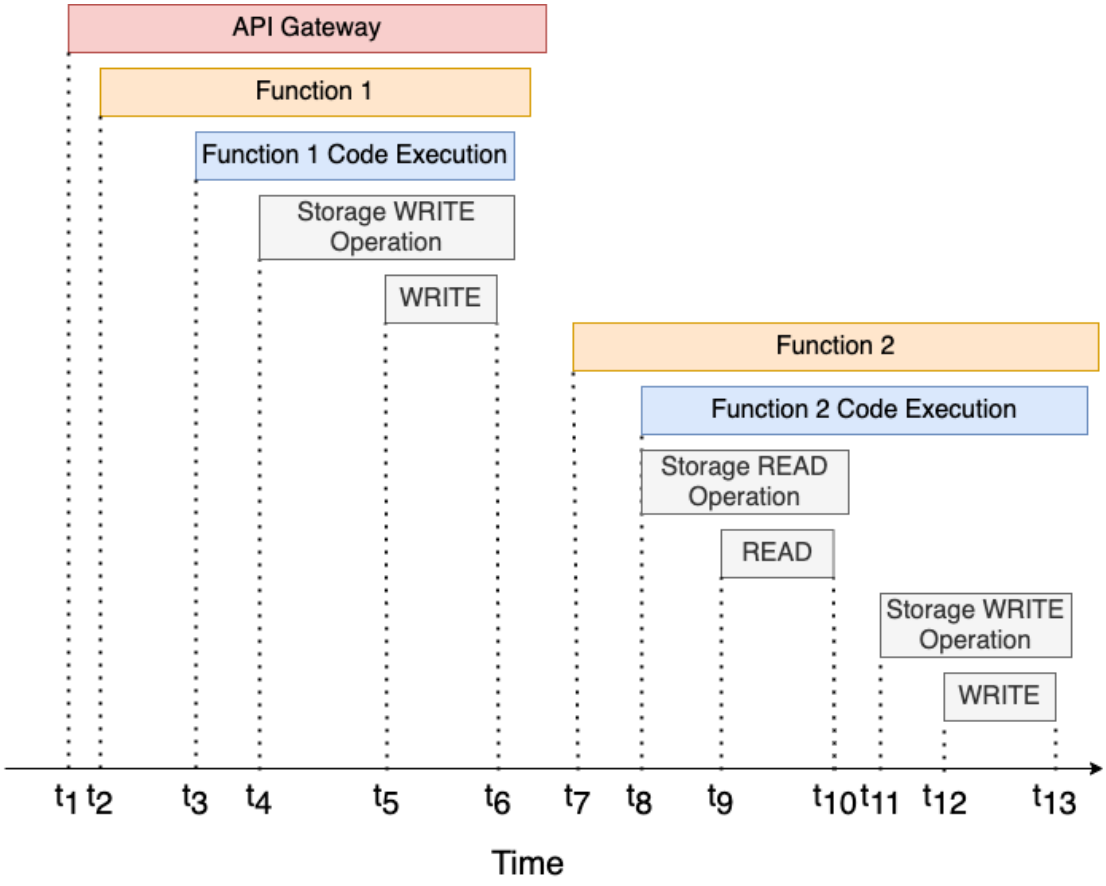
CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG



Benchmarking of Serverless Application Performance across Cloud Providers

An In-depth Understanding of Reasons for Differences

Master's thesis in Computer science and engineering

Rui Deng

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Benchmarking of Real-world Serverless Application Performance Across Cloud Providers

An In-depth Understanding of Reasons for Differences

Rui Deng

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Benchmarking of Real-world Serverless Application Performance Across Cloud Providers
An In-depth Understanding of Reasons for Differences
Rui Deng

Supervisor: Joel Scheuner, Department of Computer Science and Engineering
Examiner: Christian Berger, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A high-level trace conceptual design for serverless benchmark application

Typeset in LaTeX
Gothenburg, Sweden 2022

Benchmarking of Real-world Serverless Application Performance Across Cloud Providers
An In-depth Understanding of Reasons for Differences
Rui Deng
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Serverless computing has emerged as a new compelling cloud computing model for deploying modern applications, creating demand for benchmarking serverless platforms to help practitioners make a suitable choice. Existing works related to serverless benchmarking primarily focus on microbenchmarking to measure an individual aspect of function performance such as CPU speed and cold start. Some studies propose an application-centric benchmarking framework but lack an in-depth analysis of the application performance difference across cloud providers. Furthermore, none of the related studies provides details about addressing benchmark fairness.

In contrast, this thesis presents a methodology to design a serverless application benchmark for a fair comparison between two leading cloud providers: AWS and Azure. The benchmark execution generates detailed traces constituting the end-to-end execution duration which enables drill-down analysis on how the application performs differently across cloud platforms. The main finding shows that storage triggering can substantially impact the end-to-end latency, and the performance difference between cloud platforms.

Keywords: Cloud Computing, Serverless Computing, Benchmarking, Distributed Tracing

Acknowledgements

I want to express my deepest gratitude to my supervisor Joel for providing me the opportunity of working on such exciting and relevant research. His continuous support and inspiration have made this thesis project a tremendously fun and engaging research journey.

Rui Deng, Gothenburg, February 2022

Contents

List of Figures

List of Tables

1
Introduction

Cloud computing becomes the de-facto platform for both new and existing digital applications, and 40% of all enterprise applications will be deployed to the cloud by 2023 according to the prediction from Gartner [16]. When organizations shift their infrastructure from on-premise to the cloud, they typically start the cloud journey by leveraging virtual machines (VM) (e.g., AWS EC2[1]) or other VM-based services on the cloud to build their application environment and deploy applications. However, managing the scaling of VMs (especially scaling down to zero) can be challenging, which requires high expertise and a significant amount of time to achieve a good level of stability. Failing to properly handle scaling can cause over-provisioning in the sudden usage surge resulting in higher cost than expected while incurring unnecessary cost when there is little workload or even zero workloads since cloud customers still need to pay for the running VM instance [7].

As a new compelling cloud computing model for deploying modern applications, serverless or serverless computing has become increasingly accepted by industry during past years. This is because it promises that all underlying resources provisioning and scaling are fully managed by the cloud provider and thus offers advantages in reducing costs for irregular or bursty workloads, alleviating operational concerns, and supporting out-of-box scalability [10]. For example, a performance and cost comparison of cloud services for deep learning applications shows that serverless can deliver better performance and cheaper cost in bursty workload [8].

When architects and developers choose a cloud provider to build applications using serverless, they want to compare how applications would perform on the candidate cloud platforms. It is because application performance matters to the end-user experience that is crucial for business in today's highly digitized world. Operational cost is also directly connected to the application performance due to the pay-per-use pricing model based on execution duration and resource consumption (i.e., memory) [2, 34, 19].

1.1 Problem Description

Understanding serverless application performance across cloud providers can usually be achieved by conducting performance evaluation or performance benchmarking

[1] `https://aws.amazon.com/ec2/`

[22] to systematically analyze the application's performance on each target cloud platform. Prior works mostly use microbenchmarks to measure individual aspects that impact the serverless performance, such as CPU and memory-bound performance, network performance, and cold start [41]. Some existing works start to propose benchmark frameworks and tools that are more application-centric, aiming for evaluating the overall performance of an application and the interaction between its components [21, 45, 13]. However, there is a lack of in-depth analysis of the detailed breakdown of overall performance for a realistic application to understand the difference across cloud providers clearly.

Another gap in serverless benchmarking is the lack of a transparent approach to address the fairness of a benchmark. Creating a good benchmark has been, for a long time, considered to be challenging due to various details that can influence the adoption of benchmark and the results [15]. The difficulty has been further increased in serverless because the underlying infrastructure is abstracted away with vendor-specific implementation, which raises concern about the fairness of benchmark, especially when comparing cloud platforms. Prior studies such as BeFaaS [21] mention fairness as one of the general requirements for their benchmarks, but none discusses how it is specifically addressed.

1.2 Purpose

The purpose of this thesis is to close these two gaps by

- establishing a methodology for building real-world serverless application benchmark to fairly compare cloud platforms
- generating insights about why serverless application performance differs across cloud platforms

The method used in this thesis can provide a clear reference to other practitioners and researchers for addressing fairness in their serverless benchmarking study. For practitioners (developers/architects) who already decided to use serverless computing as their technical choice for the application, the insights generated in the thesis can greatly help them select the most suitable cloud provider for their serverless applications and improve their understanding of serverless application performance on the cloud.

1.3 Research Questions

The following research questions need to be addressed to achieve the goal of the thesis.

RQ1: How to design a real-world serverless benchmark application that is fair to compare performance across heterogeneous cloud platforms?

This question aims to explore a clear approach for constructing serverless application benchmarks as the foundation for fair performance comparison. Based on the different known aspects that can potentially influence serverless application performance, this question is further split into the following sub-questions:

- RQ1.1: How can the architecture and configuration of serverless applications be mapped closely across cloud providers?
- RQ1.2: How can the instrumentation be implemented to provide comparable insights across cloud providers?
- RQ1.3: How can the workload be designed and executed to minimize the divergence of load test?

RQ2: How does serverless application performance differ across different cloud providers?

This question targets the post-experiment analysis of the serverless benchmark application performance data across the cloud platforms to observe the difference.

RQ3: Why does serverless application performance differ across cloud providers?

As a natural next step of RQ2, this question addresses the in-depth analysis of the detailed performance data making up the overall performance to investigate the connection with the observation from RQ2. This observation will give us insights into what factors lead to the realization of the same serverless application performing differently on different cloud providers.

1.4 Scope

There are a large number of Cloud providers offering Serverless computing. It is impossible to evaluate all of them within the limited time of this thesis. The target cloud platforms used in this study are thus limited to the two biggest public cloud vendors in terms of market share [17]: Amazon Web Services (AWS)[2] and Microsoft Azure[3].

For benchmark applications, it is ideal to leverage a broader range of selected Serverless applications to increase the coverage of user cases and cloud services to generate potentially more insights. Due to the time limit of the thesis, this study will focus on one representative application to create a high-quality benchmark and perform drill-down analysis for generating insights. The method used in this thesis for building, operating and analyzing the benchmark can be applied to other benchmark applications.

[2] `https://aws.amazon.com/`

[3] `https://azure.microsoft.com/en-us/`

1.5 Contributions

This thesis intends to make two contributions. One is to provide the researcher community a detailed reference methodology for building serverless application benchmarks focusing on fairness and fine-grained analysis. The other one is the insight generated from the comparison of AWS and Azure which suggests that developers and architects should be careful about choosing triggering mechanisms, especially storage triggers, when building serverless applications since it can significantly increase the end-to-end execution time.

1.6 Limitations

Since the underlying infrastructure of serverless computing is abstracted away, users rely on cloud providers to generate the detailed performance data based on the instrumentation of the application using vendor-specific SDK[4]. However, the correctness and accuracy of data is thus out of control of this thesis.

Another main limitation of this thesis is the reproducibility of results due to the nature of ever-changing cloud services and providers' constant optimization on performance. In order to achieve long-term reproducibility, continuous improvement on benchmark and data analysis is required over time which is beyond the scope of this thesis.

1.7 Thesis Outline

The remainder of this thesis is structured as follows:

- Chapter 2 introduces the fundamental concepts related to this thesis including cloud computing, serverless computing, distributed tracing, performance benchmarking, and Serverless Benchmarker (SB).
- Chapter 3 introduces the methodology developed in this thesis for designing and operating the serverless benchmark for fair performance comparison across cloud platforms using the state-of-art benchmark framework, and Agile methodology.
- Chapter 4 presents serverless benchmarking experiments results, summarizes and discusses the observation, and describes the threats to validity.
- Chapter 5 presents the related work in serverless benchmarking and discusses the difference and contribution made by this thesis.
- Chapter 6 concludes the thesis by summarizing the findings that answer the research questions and the contribution to the community and suggests future work.

[4] `https://en.wikipedia.org/wiki/Software_development_kit`

2
Background

This chapter introduces the general knowledge and concepts needed to understand this thesis, including cloud computing, serverless computing, distributed tracing, and serverless benchmarking.

2.1 Cloud Computing

Cloud computing enables on-demand access to computing resources (e.g., networks, servers, storage, applications, and services) over the network [30]. Depending on the level of management effort needed from the users, cloud computing is typically categorized into three types of models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). From an end-user perspective, IaaS provides fundamental infrastructure resources such as networking, storage and virtual machines (VM) without managing hardware while still giving maximum control on infrastructure to end users. Compared to IaaS, PaaS alleviates the operational burden of managing fundamental cloud infrastructure, and offers various developers tools and services so users can focus on building customized environments and applications. SaaS, taken to an extreme, delivers complete software products over the internet directly to end-users without users managing any underlying infrastructure. Major cloud providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud[1] offer a broad range of cloud services based on all three types of models.

With cloud computing, organizations and individual developers do not need to invest significant capital beforehand to deploy their applications, especially in the experiment and early development phase. Instead, they have immediate access to an enormous pool of computing resources and services without up-front commitment and pay only for what has been used (pay-as-go), which can lead to increased speed to market and innovation as well as reduced cost [1].

2.2 Serverless Computing

Filling the gap between PaaS and SaaS, serverless computing intends to abstract away server (VMs) management entirely from developers (compared to PaaS) and

[1] `https://cloud.google.com/`

provide out-of-box scalability (including scaling down to zero), which usually requires high-level expertise and significant efforts to achieve. It minimizes the effort needed to manage infrastructure so developers can focus on business logic and applications. Moreover, Serverless Computing only charges users when their applications are being used to serve requests or events, which is closer to the original expectation for cloud computing treated as a utility [7].

2.2.1 Function as a Service

The primary type of serverless computing is Function as a Service (FaaS) (e.g., AWS Lambda[2] and Azure Functions[3]) which uses *function* as the unit of computation to execute the user code in response to triggers such as events or HTTP requests [7]. This provides an attractive alternative to implementing the microservice-based architecture, a trending style of building applications composed of small and self-contained components that are independently salable with the goal of improving development time and scalability.

Under the hood of serverless functions, cloud providers provision function instances on-demand and at scale [14]. A function starts with handing an event such as an HTTP request which arrives at a function routing service to be routed to an available function instance. If no function instance is available, the resource manager that is responsible for scheduling and managing all the function instances needs to prepare a new function instance based on the configuration specified by the user. This process includes creating a new function instance and initializing it with necessary container image and code artifacts, which incurs a so-called *cold start* in this case. Once the function instance is fully ready, it can finally process the invocation request to handle the business logic in the user code. If the function instance is already available (known as *warm instance*), this cold start phase can be bypassed, and the event is directly processed.

2.2.2 Serverless Application

With FaaS services and other serverless components provided by cloud providers [5, 37, 20], developers can build fully-fledged Serverless applications which typically uses FaaS as the computing layer to host and execute business logic code, combining with other fully managed or serverless services for data storage, messaging, streaming, and user authentication/authorization [10]. Based on a comprehensive study on 89 serverless applications from open-source projects, academic literature, industrial literature, and domain-specific feedback [11], a serverless application has the following main characteristics:

- AWS is the dominating platform for serverless applications, accounting for 80 % of all studied applications, followed by Azure (10%). It is mainly because AWS is the first major cloud provider to offer serverless computing, and AWS Lambda is the pioneer of FaaS.

[2] `https://cloud.google.com/`

[3] `https://azure.microsoft.com/en-us/services/functions/`

- Serverless applications are mainly used for short-running tasks with low data volume and bursty workload. It is due to providers limiting how long the function can run per execution. For example, AWS Lambda can run up to *15 minutes*[4] per execution while the timeout limit for Azure function in Consumption plan[5] is only *10 minutes*[6].
- Serverless applications typically use cloud storage, cloud databases, and cloud messaging.
- HTTP triggers and cloud events are the most commonly used triggers.
- Most serverless applications use few cloud functions, with 82% of them using five or fewer functions.
- The most popular programming languages for serverless applications are Python and NodeJS. It can be because interpreted languages (Python, Ruby, JavaScript) have significantly less cold-start delays as compared to compiled runtimes (Java, .NET, etc.) [39]. However, the major choice for serverless applications on Azure is C# [12]. The main reason can be that C# is the first and most supported language by Azure [31, 35].

2.3 Distributed Tracing

While benefiting from building a decentralized highly-specialized microservices-based distributed system, developers found themselves in a new situation where they lost the observability of the whole application. In a monolithic application, tracking application requests end-to-end used to be straightforward. However, now it becomes increasingly challenging due to the complex nature of the distributed system. The general solution to this problem is using *distributed tracing*, a method of profiling and monitoring microservices-based applications to pinpoint the causes of poor performance [38]. The idea of *tracing* is to instrument the code of a distributed service at selected points to produce data when executed, and the data from various points reached by request can be combined to generate an overall trace [40]. With such end-to-end tracing, the entire life cycle of the request can be understood, and thus, it makes it easy to pinpoint any failure and performance issue during the entire processing of the request.

For Serverless applications, tracing is more difficult since the underlying infrastructure is abstracted away and out of the user's control. Therefore, developers have to rely on distributed tracing tools offered by cloud platforms to collect the tracing data. The following subsection introduces the distributed tracing tool provided by AWS and Azure.

[4] `https://www.amazonaws.cn/en/new/2018/aws-lambda-enables-functions-that-can-run-up-to-15-minutes/`

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

[6] `https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale`

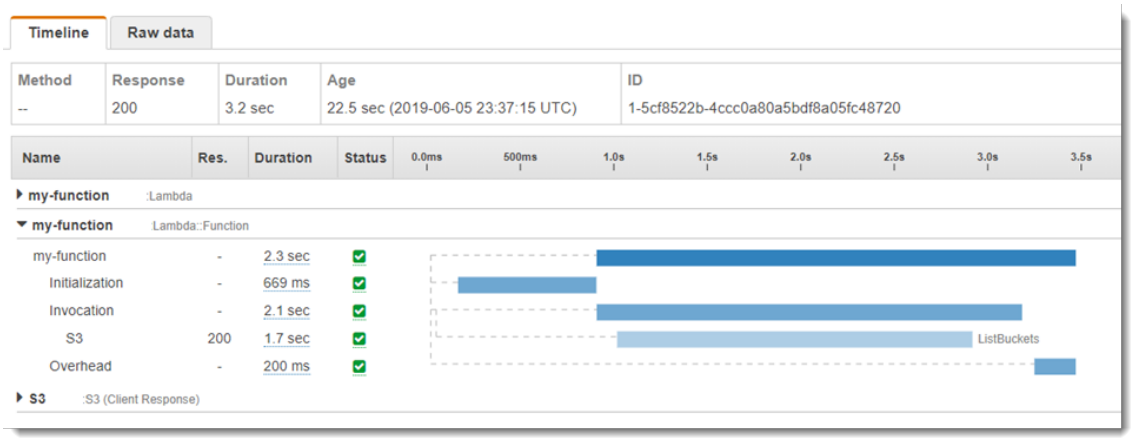


Figure 2.1: An AWS X-Ray timeline view of Lambda default trace

2.3.1 AWS X-Ray

The distributed tracing service provided by AWS is called X-Ray[7]. It is a fully managed service with both a Web user interface for visualizing tracing data and programming SDK for instrumentation and interacting with X-Ray Rest API. AWS X-Ray receives data from services and organizes them as *segments*. Instead of sending trace data directly to X-Ray, each client SDK sends JSON segment documents to a daemon process called *X-Ray daemon*[8] first and then uploads them in batches to X-Ray. With a *trace ID* tracking a request through the execution, all segments generated by a single request can be correlated by X-Ray to form an end-to-end trace for the request and a service graph that can visualize the services and resources making up the application. A segment can also be broken down into sub-segments which provide more granular timestamps and details about downstream calls. Figure 2.1 shows an AWS X-Ray timeline view of a simple Lambda execution default end-to-end trace[9].

Many AWS services provide various levels of integration with X-Ray, including instrumentation such as sampling, adding headers to incoming requests, and automatically sending trace data to X-Ray. These built-in integrations can provide the service's basic trace data once enabled. However, AWS X-Ray SDK for different programming languages is required to capture additional desirable data. The X-Ray SDK provides a simple mechanism to add instrumentation code into Lambda functions to implement the custom instrumentation.

[7] `https://aws.amazon.com/xray/`
[8] `https://docs.aws.amazon.com/xray/latest/devguide/xray-daemon.html`
[9] https://docs.aws.amazon.com/lambda/latest/dg/csharp-tracing.html

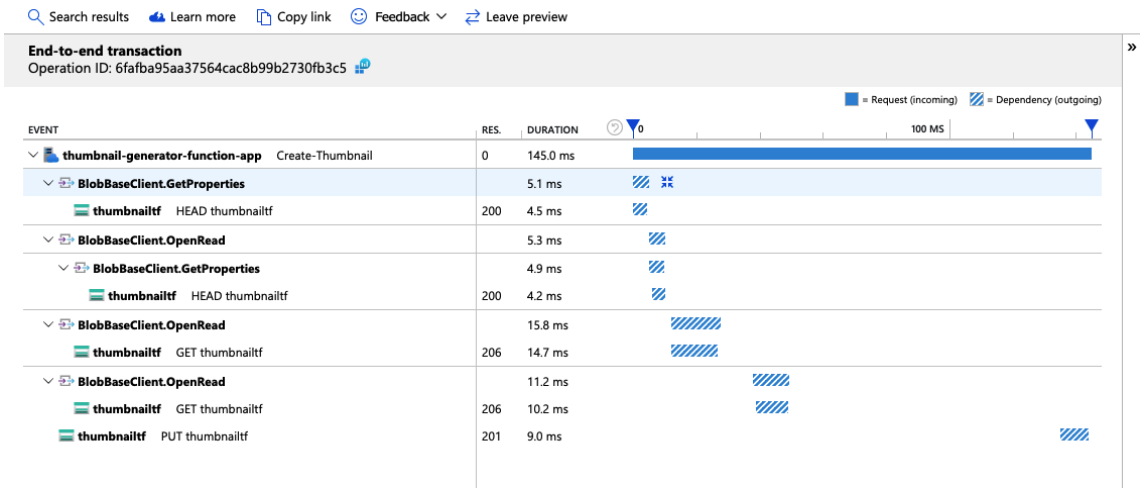


Figure 2.2: An Azure Application Insights transaction diagnostics view of an Azure Function

2.3.2 Azure Application Insights

Azure Application Insights[10], part of Azure Monitor[11], provides similar functionalities as AWS X-Ray for distributed tracing capability. It uses OpenTelemetry[12], a popular open-source observability framework, for instrumentation. Azure serverless services such as Azure Function and API Management Service offers integration support with Application Insights. Like AWS X-Ray, Application Insights starts collecting basic trace data once enabled on these services and Application SDK is required to perform custom instrumentation for capturing additional data needed. Depending on which programming languages are used, the maturity and usability of the SDK can vary. For example, Application Insights for .NET[13] receives the largest community and most frequent maintenance according to GitHub repository statistics. Figure 2.2 shows an Azure Application Insights view of function execution.

2.4 Performance Benchmarking

This section introduces the basics of performance benchmarking and its application in the serverless area (a.k.a, serverless benchmarking).

2.4.1 Benchmarking Basics

Benchmarking is an empirical and systematic approach used in computer science to compare the performance of computer systems, tools, techniques, etc., [22]. A benchmark often refers to a System Under Test (SUT), a collection of components

[10] `https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview`
[11] `https://docs.microsoft.com/en-us/azure/azure-monitor/overview`
[12] `https://opentelemetry.io/`
[13] `https://github.com/microsoft/ApplicationInsights-dotnet`

necessary to run the benchmark scenario, and includes the definition of complete application architecture with the interested components [15]. A benchmark should include the clearly defined motivation of the comparison, the representative tasks or workloads, and the measurements (quantitative or qualitative) [44].

The main concerns for building a benchmark are described as follows [24]. Successful benchmarking requires a balance between these characteristics and criteria:

- Relevance: how relevant and applicable the benchmark and results are to that area and relevant parties. For the consumer of the benchmark results, the relevance should also consider the specific context and user cases. One main challenge for relevance is scalability since it is expected to run on a broader system and simulate the behaviors of real applications. The general approach to improve benchmark relevance in a specific area is to focus on narrow applicability.
- Reproducibility: the benchmark should be able to reproduce similar results consistently. Due to the variability of modern software systems, achieving perfect reproducibility may not be possible. In reality, we can improve the reproducibility by running the benchmark for a long enough time to include representative samples of the variable behaviors. This can also mean running multiple times to improve consistency.
- Verifiability: other researchers and interesting parties can use the benchmark to verify the result to prove its trustfulness. One important way to improve verifiability is by providing as many details as possible about the benchmark and data.
- Fairness: ensure systems can be compared on their metrics without artificial constraints. The general approach to better fairness is to design benchmarks based on consensus from a panel of experts rather than individual parties and consider how the result may be used.

2.4.2 Serverless Benchmarking

There are generally two types of benchmarks in the serverless area: *micro-benchmarks* and *application benchmarks* [41]. Depending on which type of benchmarks to be used, serverless benchmarking can be categorized into *micro-benchmarking* and *application benchmarking*

Micro-benchmarking typically uses a single function to evaluate individual aspects of the serverless function, such as CPU and memory-bound performance, disk I/O performance, and network performance. For example, a single AWS Lambda function implements a function handler that obtains the parameter from its triggering invocation events and then uses a floating-point operation to calculate the latency for CPU-intensive operation [41]. Another micro-benchmark example from FunctionBench [23] uses a single function with cloud storage for downloading and uploading objects to measure network performance.

On the other hand, application benchmarking uses applications with multiple server-

less components and typically measures the end-to-end response time. For example, BeFaaS uses an e-commerce benchmark that implements a webshop application comprising 17 functions and a Redis instance as an external service to persist state [21]. Another application benchmark example can be an Image Processing application performing image transformation tasks using Python Pillow[14] library which fetches an input image from shared block storage and applies different effects to it before uploading it to another shared storage [23].

2.5 Serverless Benchmarker (SB)

To standardize the serverless benchmarking tests in a reproducible, automated way, there is a significant effort from the community to create frameworks and tools for addressing this need [23, 21, 28, 45, 49, 46, 13]. Such benchmark frameworks and tools are typically composed of the following core components:

- Built-in benchmarks: used for demonstrating the features of the framework and tool, and providing examples for how to create and integrate custom benchmarks.
- Deployment tool: standardizes the deployment of the benchmarks to target cloud providers.
- Automation of load generation: enables users to configure the load profile for orchestrating workload as needed.

One of such tools is Serverless Benchmarker (SB), created by the SPEC-RG CLOUD research group [13]. It is a meta-benchmarking tool to orchestrate reproducible serverless application benchmarking so that users can focus on implementing the target Serverless application benchmarks and designing the workload profile without worrying about the technical complexity of executing experiments.

Figure 2.3 shows a high-level architecture overview about SB comprising four major components: 1) the deployment component facilitates the deployment automation for benchmark apps to target cloud providers. SB leverages containerization technology Docker[15] to create reproducible deployment by abstracting away dependencies needed for deploying the application benchmark. It can also automatically mount application code and credentials into the right container directories which simplifies the deployment process. 2) The invocation component provides interfaces and templates to configure workload patterns for load generation. The load generation process is also automated through the integration with the open-source load testing tool K6[16] which is optimized for minimum resource consumption and good developer experience. 3) The trace downloading component provides template and integration for downloading tracing data from distributed tracing tools in the cloud providers using the corresponding SDKs and APIs. 4) the trace processing components can implement custom logic to pre-process the downloaded tracing data into curated

[14] `https://python-pillow.org/`
[15] `https://www.docker.com/`
[16] `https://k6.io/`

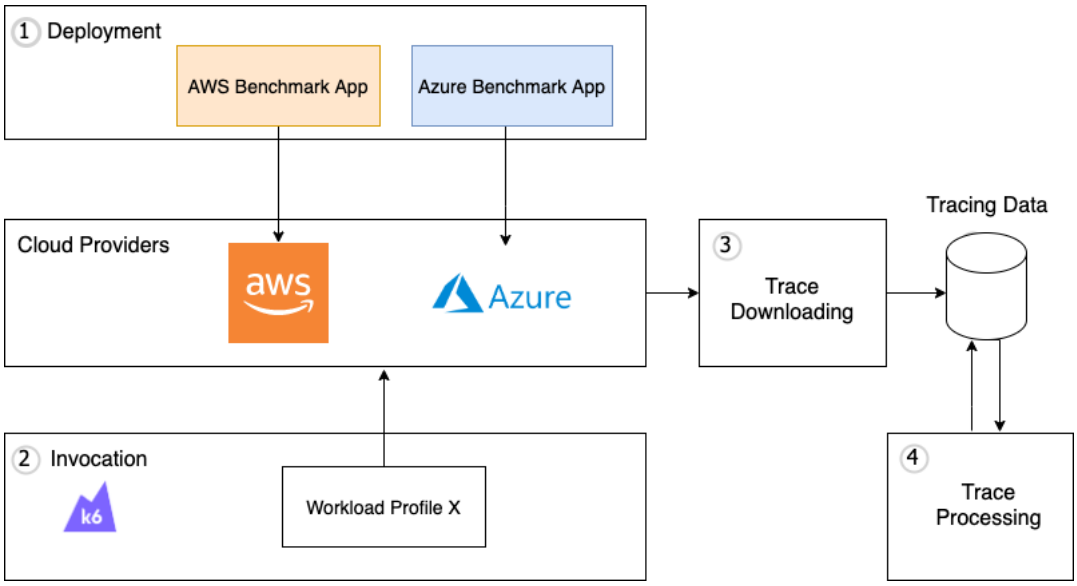


Figure 2.3: High-level overview of Serverless Benchmarker (SB) architecture

tracing data before drill-down analysis.

Figure 2.4 shows a step-by-step workflow for using SB and the Command Line Interface (CLI). In the Initialization step, SB prepares the container and installs the required packages. The Login step is about authenticating with the target cloud providers. The Deployment step then deploys the benchmark application to the cloud platform based on the specific configuration. After the benchmark application is deployed, it can be invoked in the Invocation step with a configured workload pattern defined using K6. After invocation is completed, tracing data from the cloud provider can be downloaded. Finally, the raw trace data fetched from the previous step is processed to generate final tracing data that meets the requirement. The rest of the section describes in detail how each step works:

Step 1: Initialization comes with pre-installed packages and default configuration for the underlying infrastructure. Depending on specific benchmarking needs, this step can be extended to install custom packages as needed. This step only needs to be executed once if there is no change on the dependent packages.

Step 2: Login implements the authentication with cloud platforms and the temporary credentials are mounted into the container after being downloaded. By default, AWS and Azure login with SSO is supported out-of-box, which satisfies the need of this thesis. The default AWS credential session duration is 12 hours, while the default Azure CLI token session duration is 60 minutes. Therefore, the planning of the benchmark experiment should consider the login session and run ***sb login*** before the token expires.

Step 3: Development requires implementation of the deployment configuration for benchmark application. The configuration can differ depending on technologies used

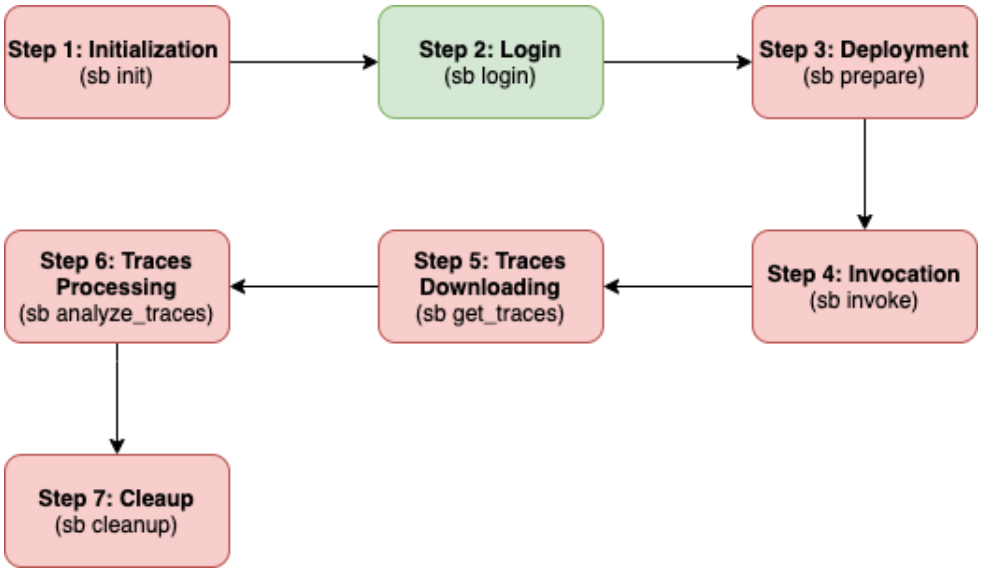


Figure 2.4: Step-by-step workflow using Serverless Benchmarker (SB)

for benchmark applications, such as the runtime and Infrastructure as Code tools. SB provides convenient functionalities for building Docker images, running CLI commands, and loading environment variables to the container.

Step 4: Invocation provides the integration point to specify how the benchmark application is invoked using K6 configuration.

Step 5: Traces Downloading aims for standardizing the download operation of traces from different cloud providers. Due to instrumentation and application difference, users may need to modify or re-implement the downloading logic instead of using the default implementation, which is required for this thesis. For AWS, all tracing data associated with an end-to-end request are unified as a single JSON file. Downloading X-Ray traces requires using AWS X-Ray SDK to fetch this monolithic JSON file based on trace ID. For Azure, tracing data is categorized into different telemetry types such as request, trace, and dependencies stored separately. Azure provides various ways to fetch the tracing data, such as Continuous Export (transferring data to other storage and then can be accessed through storage SDK) and REST API (use query expression as input to be executed through Log Analytics[17]). This thesis uses REST API via Python to download traces.

Step 5: Traces Processing pre-processes the data downloaded from the previous step and generates the final trace breakdown data by extracting and calculating the required breakdown data needed for post-experiments analysis.

Step 6: Cleanup destroys all the resources deployed in the target cloud platform.

[17] `https://docs.microsoft.com/en-us/azure/azure-monitor/logs/log-analytics-overview`

3

Research Method

This chapter introduces the methodology adopted in this thesis for conducting the application-centric serverless benchmarking across two leading cloud platforms (AWS and Azure). The overall research process is outlined firstly, and then the rest of the chapter presents the detailed design of the process including benchmark design, benchmark execution and data analysis.

3.1 Process Overview

This thesis adopts the empirical research method of benchmarking [22] to assess the performance of the serverless benchmark application on AWS and Azure. The research process includes benchmark design, benchmark execution and data analysis. In addition, an engineering research approach with controlled experiments is applied to study how to address the fairness of the benchmark. To efficiently conduct this study, Agile methodology[1] is also applied in the whole process to improve the benchmark and thesis artifacts continuously.

Figure 3.1 shows the whole research process with the input and output of each step. Based on the understanding of benchmarking theory and Serverless Benchmarker (SB) described in 2.4, the first step includes the design of the benchmark application, instrumentation and workloads, and the integration with SB, focusing on addressing the fairness for comparison to answer RQ1 in the Section 1.3. The second step is to operate the application benchmark built from the previous step and generate the tracing data. Thirdly, the data is analyzed to generate insights for answering the research questions RQ2 and RQ3 in Section 1.3. Last but not least, the benchmarking process is iterative for continuously improving the design and results.

3.2 Benchmark Design

This section describes the detailed design of the benchmark, including the design of application, instrumentation, workloads, and integration with SB.

[1] `https://agilemanifesto.org/`

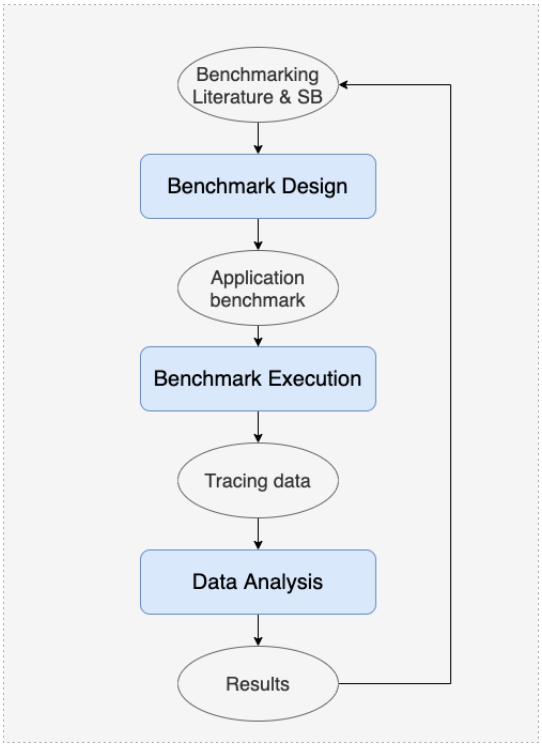


Figure 3.1: The flowchart of process overview

3.2.1 Application Design

When designing a serverless application benchmark, it is crucial to choose an application as realistic as possible to represent real-world Serverless usage. As described in Section 2.2.2, a serverless application typically contains 5 or fewer functions combining with cloud storage/database/messaging to handle short-running tasks with low data volume and bursty workload. Based on these characteristics, 3 of the supported applications in SB (Thumbnail Generator, Model Training and Video processing) are identified as candidates. Apart from representing different use cases and domains, the major difference is that Thumbnail Generator uses two functions while the other two applications use only 1 function. Considering the time constraint of this thesis, this study chooses Thumbnail Generator as the benchmark application because it requires the most straightforward domain knowledge while providing a slightly more complex scenario with two functions in the same chain execution. In addition, several related studies also use Thumbnail Generator as a target serverless application. [9, 50].

The Thumbnail Generator application is commonly used in web applications for creating resized images to fit with the User Interface (UI) on different devices and web pages. Figure 4.1 shows the high-level architecture of the Thumbnail Generator and the life cycle of the application execution is described as follows:

1. A user uploads the original image via an HTTP request to the API Gateway endpoint
2. The API Gateway triggers the serverless function Upload

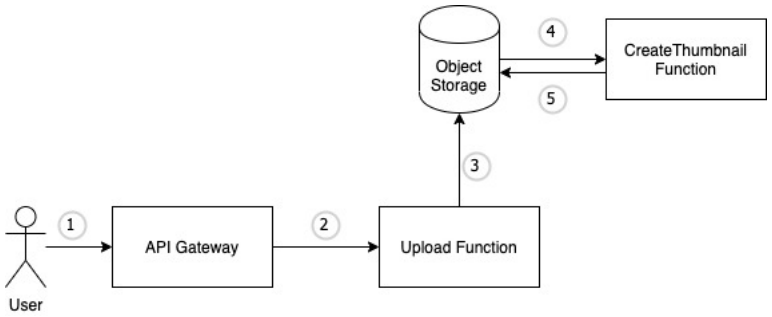


Figure 3.2: High-level design for Serverless thumbnail generation application

3. The Upload Function receives an image from API Gateway and uploads it to the Object storage
4. Once the image is saved/created in the storage, it triggers another Serverless function CreateThumbnail to process the image
5. The CreateThumbnail Function creates thumbnail images and saves them back to the storage.

To architect the application for the fair comparison across cloud platforms, it is important to use comparable components from different cloud providers. Based on the side-by-side comparison of product offerings among AWS and Azure [18], the mapping of components for the Thumbnail Generator is summarized in Table 3.1. This results in the high-level architecture on AWS and Azure as shown in Figure 3.3a and 3.3b respectively.

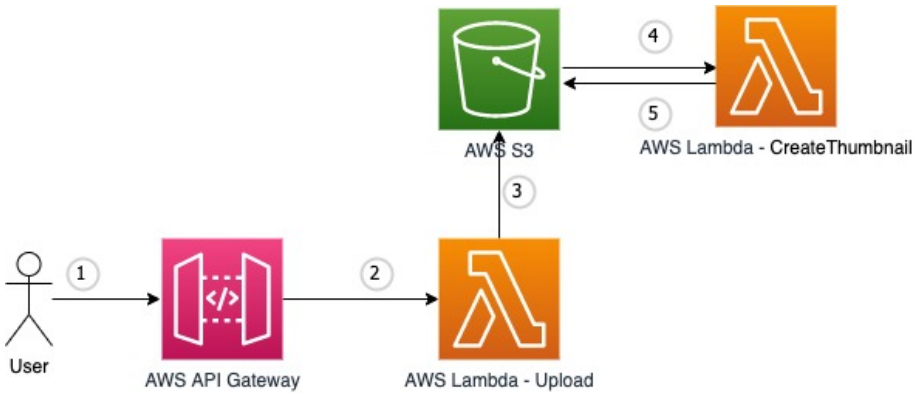
	AWS	**Azure**
API Gateway	AWS API Gateway	Azure API Management
Function	AWS Lambda	Azure Function
Object Storage	AWS S3	Azure Blob Storage

Table 3.1: Thumbnail Generator service mapping between AWS and Azure

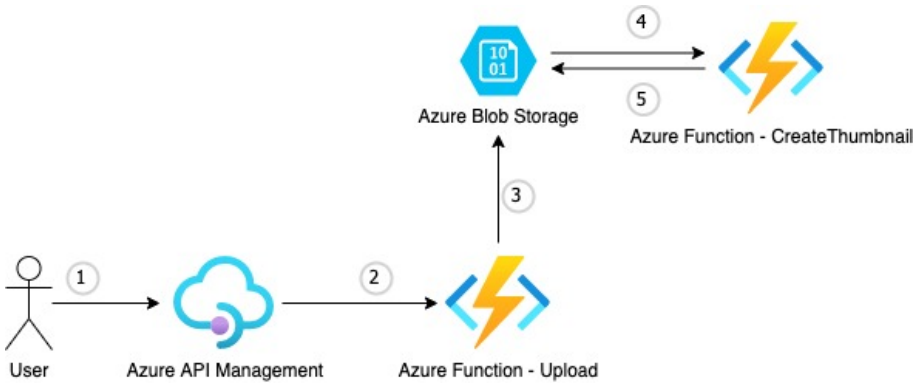
Using comparable components alone can not guarantee the fairness of comparison due to vendor-specific features and behaviors for cloud services. The following major aspects are identified during the benchmark design and need to be addressed properly with comparable configuration in order to improve fairness.

- *Function invocation method for cloud storage*: cloud providers such as AWS and Azure generally implement function invocation in two ways: polling-based and event-driven. The invocation method can be different depending on which cloud services are integrated with function. For example, AWS Lambda uses an event-driven method with S3[2] and API Gateway[3] but polling with DynamoDB[4] and Simple Queue Service (SQS)[5] [6]. The mechanism used by

[2] `https://aws.amazon.com/s3/`
[3] `https://aws.amazon.com/api-gateway/`
[4] `https://aws.amazon.com/dynamodb/`
[5] `https://aws.amazon.com/sqs/`



(a) AWS



(b) Azure

Figure 3.3: Detailed design for serverless thumbnail generation application on AWS and Azure

serverless functions on different cloud platforms can also be different for the same type of cloud service. For example, contrary to AWS Lambda with S3, the Azure Blob storage trigger used by the Azure function by default uses a "pulling" mechanism to periodically scan the storage containers and their logs for capturing events. This method not only can not guarantee all events can be captured because logs may be missed and blobs are scanned in a group of 10000 at a time but also can cause up to a 10-minute delay if the function application is on the Consumption plan and the function has gone idle [32]. To a have fair triggering mechanism for the second function (CreateThumbnail) triggered by cloud storage, the Azure function needs to leverage the Event-Grid trigger, which essentially uses EventGrid[6] service to send HTTP requests to notify the target about events that happen in the publisher (in this case, Azure Blob storage).

- *Cold start enhancement feature*: since the cold start is a well-known issue for serverless function, cloud providers started to offer solutions to resolve it during past years. AWS provides Provisioned Concurrency[7] for Lambda,

[6] `https://azure.microsoft.com/en-us/services/event-grid/`
[7] `https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html`

which keeps functions warm to respond, while Azure offers Premium plan[8] that uses pre-warmed workers to run applications with no delay after being idle. To have a fair comparison and keep the function's serverless nature, both the AWS and Azure applications in the benchmark should use their default configuration without these enhanced features.

- *Function Operating System (OS)*: For function hosting OS, AWS Lambda supports only Amazon Linux[9] which is AWS's own Linux distribution, while Azure supports both Windows and Linux. For a fair comparison, it is logical to consider using Linux for AWS Lambda and Azure Functions in the benchmark. However, this thesis argues that Amazon Linux 2 for AWS and Windows for Azure are more comparable due to their similar maturity on both platforms. Azure Functions were introduced first with support only for Windows [31] and Windows is the default OS for Azure Functions until today. Linux support was previewed one year later [36] with less optimization compared to Windows. One issue reported earlier regarding configured user limits on Linux [33] still occurs during the initial benchmark testing of this thesis using Linux for Windows Azure. Furthermore, many related works such as [27, 28, 21] also use default Windows for Azure in their benchmark.
- *Memory Size*: one of the major performance parameters for a serverless function is memory size which determines the amount of CPU power is allocated to the function instance. For Azure, the maximum memory size allocated to each instance is 1.5 GB and is not configurable for the default Consumption plan. In contrast, AWS allows granular control on memory size between 128 MB to 10248 MB. To avoid CPU throttling applied by the provider and have a similarly full vCPU core setup as Azure, AWS lambda functions in the benchmark are configured with 1769 MB [4, 46].
- *Programming language and runtime*: runtime and programming language also have a notable impact on fairness. The same runtime and programming language should be used for both AWS and Azure implementation to have a fair comparison. In this thesis, .NET Core 3.1 and C# are chosen due to mature software development kit (SDK) support for instrumentation, good documentation (especially in Azure), and large community on both platforms [3, 31].
- *Code reusability*: To minimize the impact and discrepancy caused by implementation code in the same programming language, the code should be reused as much as possible. In this serverless application, the main code that can be shared is the image resizing logic which is the primary business logic.

3.2.2 Instrumentation Design

Apart from the general requirements described in Section 2.4, the additional interest for this thesis is the drill-down analysis of performance to gain insights into the root cause of performance difference. Thus the benchmark should also support

[8] `https://docs.microsoft.com/en-us/azure/azure-functions/functions-premium-plan?tabs=portal`

[9] `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/amazon-linux-ami-basics.html`

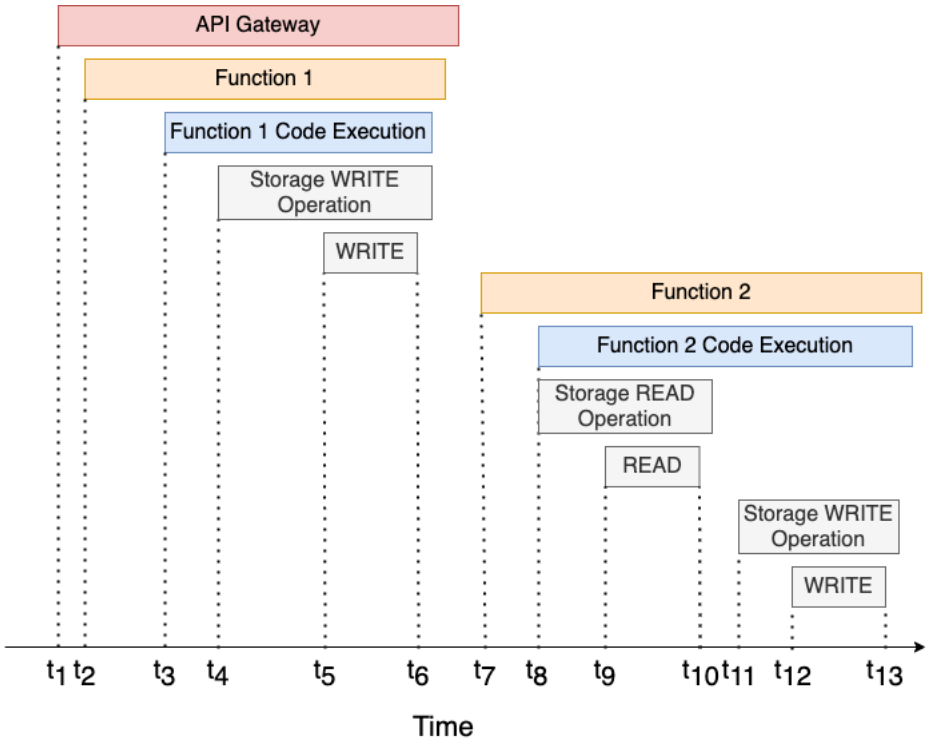


Figure 3.4: Trace design for serverless thumbnail generator application

the detailed breakdown measurements of the end-to-end performance. This section starts with the design for the end-to-end tracing to define the interesting timestamps that can be used for drill-down analysis. Beyond the trace design, the detailed instrumentation on AWS and Azure will also be presented to show how the challenge of generating comparable traces on both platforms has been addressed.

3.2.2.1 Trace Design

Based on the architecture of the application presented in the Section 3.2.1, there are 13 interesting timestamps identified, as shown in Figure 3.4, for drill-down analysis. The detailed description of these timestamps are presented as follows:

- t_1: The time in which the HTTP request is made to the API. This is defined as the starting time of the whole end-to-end tracing life cycle.
- t_2: The time in which the Upload Function is invoked. From this point, the life cycle of the Upload function starts.
- t_3: The time in which the Upload Function custom logic starts to be executed. This is also the time in which the first line of code of the function starts to be executed, which matters most for the user of this function.
- t_4: The time in which the Upload Function starts to interact with the storage service for uploading the image.
- t_5: The time in which WRITE operation starts for putting an image to storage.
- t_6: The time in which the WRITE operation ends which means the image is stored in the storage.
- t_7: The time in which the CreateThumbnail function is invoked. From this

point, the life cycle of the CreateThumbnail function starts.

- t_8: The time in which the CreateThumbnail Function custom logic starts to be executed. This is when the first line of code of this function's business logic starts to be executed, which matters most for the user of this function. At the same time, the CreateThumbnail function also starts to interact with the storage service for reading the image uploaded from the previous function.
- t_9: The time in which the READ operation starts for reading the image from the storage service
- t_{10}: The time in which the READ operation ends
- t_{11}: The time in which the CreateThumbnail Function starts to interact with the storage service for uploading the image.
- t_{12}: The time in which WRITE operation starts for putting the resized image object to the storage.
- t_{13}: The time in which the WRITE operation ends which means the resized image is stored in the storage. This signals the end of the process from the end user's perspective despite the additional time needed to shut down the function by the cloud provider.

With these timestamps, the whole life cycle of the application execution can be broken down into the following detailed, measurable segments:

- *HTTP Triggering Duration (t_1t_2)*: The duration between t_1 and t_{12} shows the time used for the HTTP API endpoint to trigger the Upload function.
- *Upload Function Startup Overhead (t_2t_3)*: The duration between t_2 and t_3 shows the actual time used for the Upload function to be started, including potential function cold start.
- *Upload Function Event Parsing Overhead (t_3t_4)*: The duration between t_3 and t_4 refers to overhead for reading and parsing HTTP trigger events for extracting useful information.
- *Upload Function Storage Connection Overhead (t_4t_5)*: The duration between t_4 and t_5 refers to overhead for connecting storage services.
- *Upload Function WRITE Operation Duration (t_5t_6)*: The duration between t_5 and t_6 refers to the time used for the actual image WRITE operation to storage.
- *Storage Triggering Duration (t_6t_7)*: The duration between t_6 and t_7 shows the time used for storage service to trigger the CreateThumbnail function.
- *CreateThumbnail Function Startup Overhead (t_7t_8)*: The duration between t_7 and t_8 shows the overhead for CreateThumbnail function to be started by function service, including potential function cold start.
- *CreateThumbnail Function Storage Connection Overhead (t_8t_9)*: The duration between t_8 and t_9 refers to overhead for connecting storage services for reading the image from the storage service.
- *CreateThumbnail Function READ Operation Duration (t_9t_{10})*: The duration between t_9 and t_{10} refers to the time used for the actual image READ operation to storage.
- *Image Resizing Duration ($t_{10}t_{11}$)*: The duration between t_{10} and t_{11} refers to the time used for resizing image computation.

- *CreateThumbnail Function Storage Connection Overhead (t11t12)*: The duration between t_{11} and t_{12} refers to overhead for connecting storage services for writing resized images to storage service.
- *CreateThumbnail Function WRITE Operation Duration ($t_{12}t_{13}$)*: The duration between t_{12} and t_{13} refers to the time used for the actual resized image WRITE operation to storage.
- *Total Duration (t_1t_{13})*: The duration between t_1 and t_{13} shows the end-to-end duration of trace.

3.2.2.2 Instrumentation for AWS

Basic tracing data is generated out-of-box once X-Ray is enabled on components such as API Gateway and Lambda. These basic tracing data from X-Ray covers 9 of the 13 timestamps (t_1, t_2, t_5, t_6, t_7, t_9, t_{10}, t_{12}, t_{13}) defined in Section 3.2.2.1. What is more, AWS X-Ray provides out-of-box correlation for all the traces generated in this application, especially between the two functions. This significantly eases the instrumentation of the AWS benchmark. To get the rest of the timestamps, AWS X-Ray .Net SDK[10] is used for instrumentation in Lambda function code.

3.2.2.3 Instrumentation for Azure

Azure Application Insights also has built-in integration with primary services used in the Azure application, such as API Management and Azure Function. The default tracing data generated by Application Insights also covers the same 9 of the 13 timestamps (t_1, t_2, t_5, t_6, t_7, t_9, t_{10}, t_{12}, t_{13}) as AWS X-Ray. The rest of the timestamps can be captured by instrumenting the Azure function code with Application Insights C# SDK[11].

However, Azure Application Insights do not provide an out-of-box correlation between the two functions in the Azure application. This can be addressed by leveraging metadata of the Blob storage object to store the trace ID. When the image is read in the following CreateThumbnail function, the trace ID from metadata can be read and add it to all traces associated with the CreateThumbnail function as a custom property. In consequence, this solution adds a small overhead to the total duration of the Azure application. However, considering such correlation may be implemented by AWS under the hood, this thesis argues that the overhead is necessary for Azure to achieve comparable instrumentation with AWS.

3.2.2.4 Cold Start Detection

Cold start only occurs during the first execution of a cloud function because the container must be started before the execution while the subsequent executions reuse the *warm* container for better performance [29]. Depending on cloud providers, the container can be kept *warm* for a different duration. For example, AWS Lambda

[10] `https://docs.aws.amazon.com/xray/latest/devguide/xray-sdk-dotnet.html`
[11] `https://github.com/microsoft/ApplicationInsights-dotnet`

can be terminated after being idle for 5 to 7 minutes while Azure Functions can be kept warm for 20 to 30 minutes [43]. To compare the performance in the situation when serverless functions are warm (a.k.a no cold start), the instrumentation should indicate whether there are cold starts for each function. This is straightforward in AWS since the default *invocation segment*[12] provides an indication of cold start out-of-box. Azure, however, doesn't provide such data by default, and additional configuration is needed to provide such instrumentation. The solution used in this thesis is to leverage Azure's *scale controller logs*[13] which can be enabled on function applications. This generates additional traces about the initialization process of the Azure function application host in which the two Azure functions are deployed and thus can be used for determining the cold start.

3.2.3 Workload Design

Since serverless applications do not start themselves, workload needs to be generated to invoke the benchmark application for analyzing its performance. To fairly compare the performance of the benchmark applications across cloud platforms, the workload generated in the benchmarking experiments should share the same workload model and size of data as input and be realistic. According to the characterization of production FaaS workload based on data collected across Microsoft Azure's entire infrastructure between July 15th and July 28th in 2019 [42], most of the serverless applications are invoked very infrequently (81% of them invoked at most once per minute on average) while less than 20% of the applications are responsible for 99.6% of all invocations (invoked on average at least once per minute). This can be due to the fact that serverless is more economically efficient for applications with a low invocation rate and bursty demand [39]. To simulate such two typical types of invocation patterns, this thesis creates a constant workload model as a baseline to evaluate the performance of serverless benchmark applications under infrequent invocation scenarios and a bursty workload model for heavy load scenarios. This section presents the detailed design for these two workload models.

3.2.3.1 Constant Workload

Constant workload contains only one phase using a constant load with low RPS (Requests per second). This thesis chooses to limit the total duration of the workload to a maximum 1 hour, which means the average request frequency is much higher than the infrequent workload scenario indicated by Microsoft serverless characterization data (at most once per minute on average). This simplification is motivated by 1) time constraint. Since using one invocation per minute, for instance, would take a long time for executing large numbers of invocations for data analysis. It also increases the time for repeating the experiments. 2) AWS Lambda can be kept warm for 5 to 7 minutes while Azure Functions can be kept warm for 20 to 30 minutes [43]. Invocations within a warm period do not change the main behavior of the serverless function. 3) this thesis primarily focuses on warm invocation scenarios

[12] `https://docs.aws.amazon.com/lambda/latest/dg/services-xray.html`

[13] `https://docs.microsoft.com/en-us/azure/azure-functions/configure-monitoring?tabs=v2#configure-scale-controller-logs`

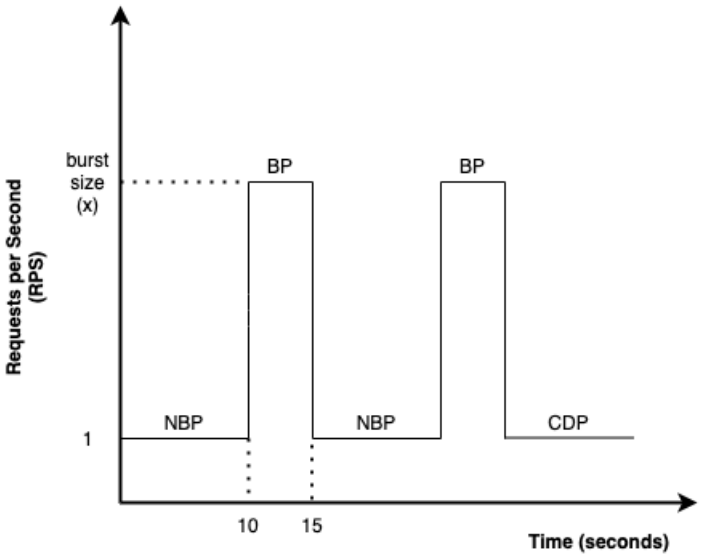


Figure 3.5: Bursty workload model visualization

which accounts for most of the invocations for serverless applications. Significantly infrequent scenarios such as once per 10 minutes for AWS (meaning cold start every time) are not the focus for this thesis.

Due to cost constraints, this thesis also chooses to limit the target number of total invocations to 300 (the actual number of invocations may not be the same as this number with the minor difference due to the limitation of load generation tool K6).

Therefore, three constant workloads are defined as follows:

- *Constant1*: Constant load with 1 request per 10 seconds running for 50 minutes
- *Constant2*: Constant load with 1 request per 3 seconds running for 15 minutes
- *Constant3*: Constant load with 1 request per second running for 5 minutes

3.2.3.2 Bursty Workload

Bursty workload design uses sequential phases with constant load to construct a bursty load pattern. It is composed of non-bursty phase (NBP), bursty phase (BP) and cooldown phase (CDP). The workload starts with non-bursty phases, alternates non-bursty and bursty phases, and ends with the cooldown phase following the last bursty phase. The non-bursty phase runs for 10 seconds with RPS 1, while the bursty phase runs for 5 seconds with a bursty size variable as its RPS value. For the cooldown phase, the RPS value is always 1, but the duration depends on how many invocations are still needed to meet the target number of total invocations. Figure 3.5 visualizes the bursty model with the three types of phases. With the same target number of total invocations as the constant workload model, three bursty workloads with different bursty size are defined as follow:

- *Bursty1*: burst size is 12 and the cooldown duration is 20 seconds.
- *Bursty2*: burst size is 25 and the cooldown duration is 30 seconds

- *Bursty3*: burst size is 50 and the cooldown duration is 40 seconds.

Table 3.2 summarizes all three bursty workloads Bursty1-Bursty3 with detailed invocation pattern for each phase.

	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**Total**
Bursty1	1x10	12x5	1x10	12x5	1x10	12x5	1x10	12x5	1x20	300
Bursty2	1x10	25x5	1x10	25x5	1x10	1x30	-	-	-	300
Bursty3	1x10	50x5	1x40	-	-	-	-	-	-	300

Table 3.2: Sequential phases (P1-Pn) of three bursty workload models. Each phase contains invocation pattern expressed in the form of **RPS x Duration** and the target total number of invocation is **300**

3.2.4 SB Integration

The integration with SB requires several main customizations based on the application and instrumentation design.

- Deployment: since both of AWS and Azure benchmark applications in this thesis are implemented using the open-source Infrastructure as Code tool Terraform[14] and Microsoft .Net Core (C#), Deployment (*sb prepare*) step (described in Figure 2.4) requires implementation such as building the function code assets using *dotnet publish* and running Terraform commands to deploy the application.
- Traces Downloading: for downloading tracing data via ***sb get_traces*** from Azure, this thesis leverages Azure's Log Analytics feature for Application Insights To efficiently fetch all relevant telemetry data associated with an end-to-end trace. It provides a friendly UI to write and execute custom queries for experimenting and tweaking the query to get the minimum needed data. After the query is finalized, it can then be used as input for calling Application Insights REST API in the trace downloading implementation to query the data. The JSON result of the Rest API is a list of telemetry data records that can be easily converted into Python Pandas[15] *dataframe* for further processing. Compared to AWS X-Ray trace downloading, this provides flexibility to optimize data size.
- Traces Processing: implements custom logic to pre-processes the tracing data and generate the final trace breakdown data (t_1 to t_{13}, defined in Section 3.4). To facilitate the study of the trace in the warm function invocation case, two additional measurements are added into the final trace breakdown: *f1_cold_start* and *f2_cold_start*, which respectively indicate if Upload function (function 1) and CreateThumbnail function (function2) have cold starts nor not. The value of the two columns is either 0 or 1 (1 means cold start while 0 means no cold start).

[14] `https://www.terraform.io/`
[15] `https://pandas.pydata.org/`

3.3 Benchmark Execution

All benchmark experiments are executed from a local computer. This simulates real users' interaction with benchmark applications and has a limited impact on the benchmarking results since the tracing is performed from the server-side starting from API being triggered. However, to minimize the network transmission discrepancy caused by cloud provider location, both AWS and Azure are deployed to the closest region possible for a fair comparison. This thesis deploys the AWS application in *us-east-1* region and Azure application in *eastus* region.

This thesis also leverages the SB SDK to create Python scripts to automate the experiment execution based on SB workflow. Listing 3.1 shows the experiment plan script using Constant3 workload. It starts to specify the location of the SB benchmark configuration files (*thumbnail_benchmark.py*) for both AWS and Azure apps so SB knows how the application should be deployed and invoked. Then the workload model is specified using *K6 options*[16]. Lastly, the function *run_test* defines detailed steps for executing the SB workflow described in Section 2.5.

```
1
2  """Constant workload
3  Runs an experiment for the thumbnail_generator app with constant
       workload that keeps 1 interaction per second for 5 minutes
4  """
5
6  import logging
7  from pathlib import Path
8  from sb.sb import Sb
9
10 apps_dir = Path('.')
11 apps = [
12     'AWS/thumbnail_benchmark.py',
13     'Azure/thumbnail_benchmark.py'
14 ]
15 app_paths = [(apps_dir / a).resolve() for a in apps]
16
17 sb_clis = [Sb(p, log_level='DEBUG', debug=True) for p in app_paths]
18
19 options = {
20     "scenarios": {
21         "constant": {
22             "executor": "constant-arrival-rate",
23             "rate": 1,
24             "timeUnit": "1s",
25             "duration": "5m",
26             "preAllocatedVUs": 20,
27             "maxVUs": 50
28         }
29     }
30 }
31
32
```

[16] https://k6.io/docs/using-k6/options/

```
33 def run_test(sb):
34
35     try:
36         sb.prepare()
37         sb.config.set('label', 'experiment_non_bursty_3.py')
38         sb.invoke('custom', workload_options=options)
39         sb.wait(5 * 60)
40         sb.get_traces()
41         sb.analyze_traces()
42     except:
43         logging.error('Error during execution of benchmark. Cleaning up
       ...')
44     finally:
45         sb.cleanup()
46
47 for sb in sb_clis:
48     run_test(sb)
```

Listing 3.1: An experiment plan in Python shows the automation of SB execution workflow

3.4 Data Analysis

To visualize the data in different ways for analysis, additional Python scripts are created for generating the plots such as CDF (Cumulative distribution function)[17] plot and Violin plot[18]. In CDF, the y axis shows the cumulative probability or percentile of the data distribution, and the main advantage of the CDF is giving readers a direct understanding of the information such as minimum, maximum, median, and percentiles. Violin plot is another method of plotting the numeric data, combining box plot and kernel density plot. The main advantage of the Violin plot is that it shows the entire distribution of the data apart from the median, interquartile range, minimum, maximum. The width of the Violin plot shows how frequently that value occurs in the data set. This thesis uses CDF to visualize the total duration of benchmark application invocation and shows breakdown segments of the total duration (described in the Section 3.2.2.1) with a Violin plot. To easily compare plots for different workloads, the X and Y axis for the CDF and Violin plots are adjusted to use the same scale based on the actual results.

[17] https://en.wikipedia.org/wiki/Cumulative_distribution_function
[18] https://en.wikipedia.org/wiki/Violin_plot

4
Results and Discussion

The benchmarking experiments results are grouped based on the two workload models (Constant and bursty workload) described in Section 3.2.3. Results for each workload model are divided into two groups: all invocations and warm invocations. The former includes all invocations and studies the overall performance results. The latter includes only warm invocations to focus on the application performance in the warm invocation scenario which represents the majority of invocations. The observation results are based on drill-down analysis of the CDF plot and statistics summary of total duration as well as the Volin plot of all segments making up the total duration described in Section 3.2.2.1 (except for t_3t_4 due to its minor impact on performance and irrelevance to serverless). The results and findings are presented at the end of each workload result.

4.1 Constant Workload

This section presents the results of the experiment for all three constant workloads Constant1-Constant3 described in Section 3.2.3.1.

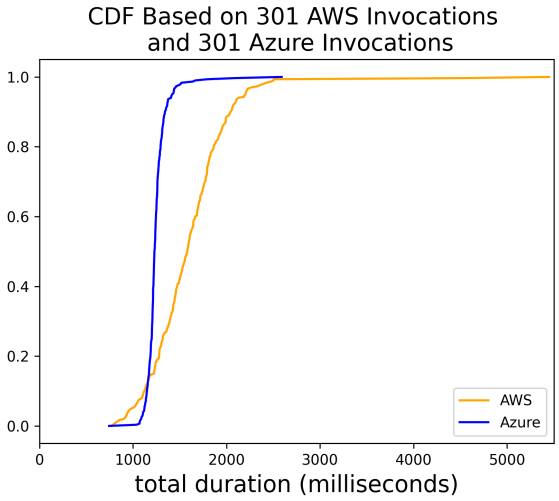
4.1.1 All Invocations

Figure 4.1 presents the CDF of total duration for all invocations based on three constant workloads while detailed statistics summary for total duration in milliseconds (ms) are shown in Table 4.1, 4.2, 4.3.

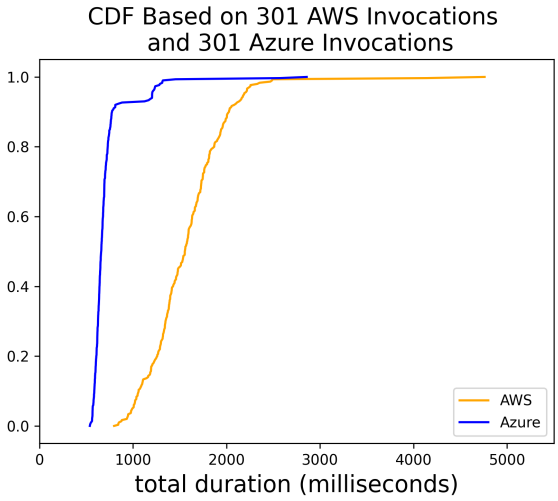
- All results indicate that end-to-end total duration is generally shorter in Azure than AWS except for some outliers.
- With increasing RPS, AWS shows lower variability for 99th percentile and bigger improvement for the maximum duration while Azure shows both bigger variability in 99th percentile and bigger increase for maximum duration.

To help understand what specifically contributes to these distinct performance difference and variability, Figure 4.2 presents the Violin plot of the main segments making up the total duration for all invocations. As clearly shown in all Violin plots:

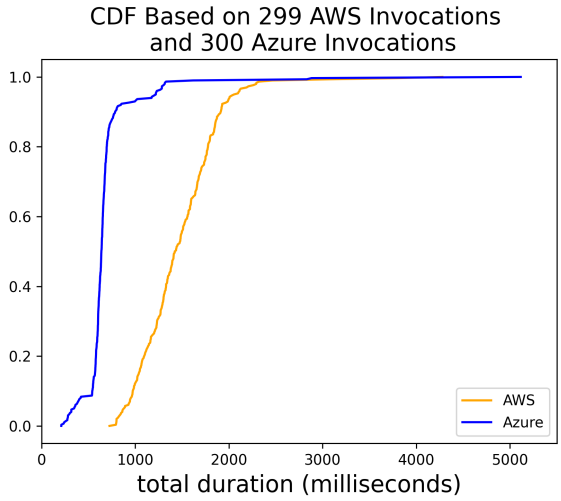
- AWS yields significantly longer latency when triggering the second function via storage (S3) events (t_6t_7) while Azure, in contrast, shows a much lower median and standard deviation despite some outlier in Constant2.
- With increasing RPS, AWS keeps relatively stable for most segments and even



(a) Constant1 workload



(b) Constant2 workload



(c) Constant3 workload

Figure 4.1: CDF plot of total duration based on three **constant** workloads with **all invocations**

bigger improvement on maximum for t_6t_7 while Azure shows significant variability for t_2t_3 (startup overhead of the first function) and increase in maximum for t_1t_2, t_2t_3 and t_7t_8

4.1.2 Warm Invocations

Figure 4.3 presents the CDF of total duration for all warm invocations based on three constant workloads. Compared to all invocations results, the observation is similar except that AWS and Azure show decreased high-percentile latency (99th percentile and maximum) or tail latency. As data shown in Table 4.1, 4.2, 4.3, AWS has significantly lower 99th percentile and maximum values for Constant2 and Con-

	All Invocations			*Warm Invocations*		
	AWS	Azure	Difference	AWS	Azure	Difference
Median	1581	1230	351	1580	1228	352
95th percentile	2204	1419	785	2191	1413	778
99th percentile	2504	1668	836	2477	1639	838
Maximum	5451	2591	2860	5451	2090	3361

Table 4.1: Statistics summary of total duration in milliseconds for **Constant1 workload**

	All Invocations			*Warm Invocations*		
	AWS	Azure	Difference	AWS	Azure	Difference
Median	1556	661	895	1554	660	894
95th percentile	2188	1205	983	2166	1205	961
99th percentile	2487	1321	1166	2330	1315	1015
Maximum	4760	2858	1902	2492	2858	-366

Table 4.2: Statistics summary of total duration in milliseconds for **Constant2 workload**

	All Invocations			*Warm Invocations*		
	AWS	Azure	Difference	AWS	Azure	Difference
Median	1422	639	783	1417	638	779
95th percentile	2058	1220	838	2007	1012	995
99th percentile	2476	1632	844	2291	1304	987
Maximum	4278	5115	-837	2464	1329	1135

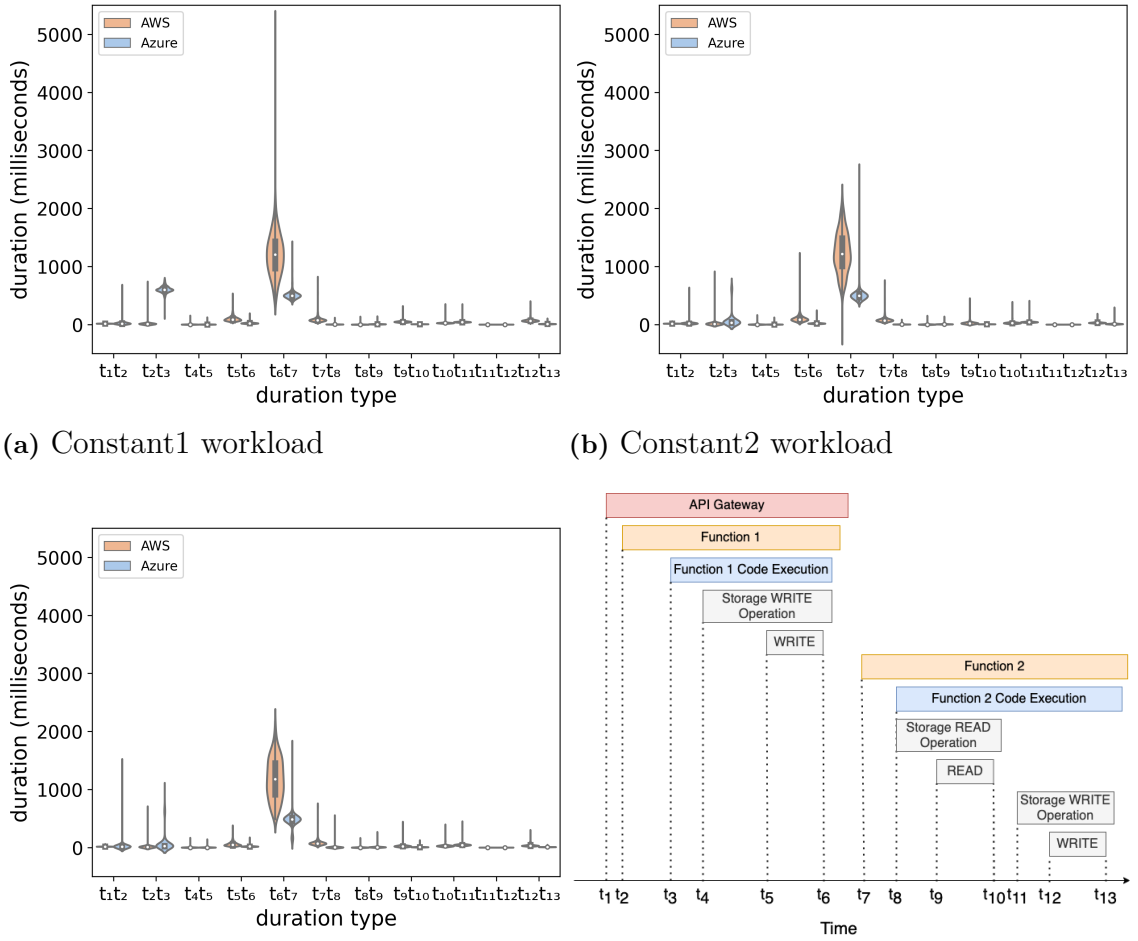
Table 4.3: Statistics summary of total duration in milliseconds for **Constant3 workload**

stant3 while maximum for Constant3 in Azure decreases from 5115 ms to 1329 ms.

For detailed breakdown, the violin plots in Figure 4.4 show that outliers for most of the segments decrease compared to all invocations results. Compared to all invocations results, the observation is similar except that t_2t_3 becomes the clear driver for the variability for Azure when increasing RPS.

4.1.3 Discussion

As results analysis shown in previous two sections 4.1.1 and 4.1.2, *Azure outperforms AWS in overall performance regardless of considering all invocations or only warm invocations*. This observation is surprising, and in contrast to many prior findings in relevant research such as [28, 46, 9] whose results show that AWS, as a pioneer in FaaS, is better performing cloud provider in serverless than Azure in constant workload.



(a) Constant1 workload

(b) Constant2 workload

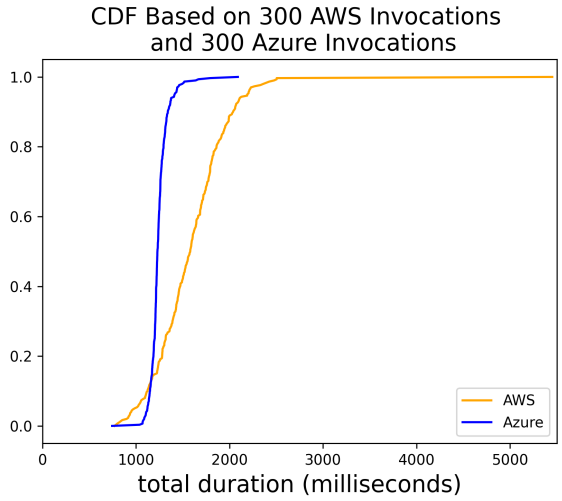
(c) Constant3 workload

(d) Trace design reference

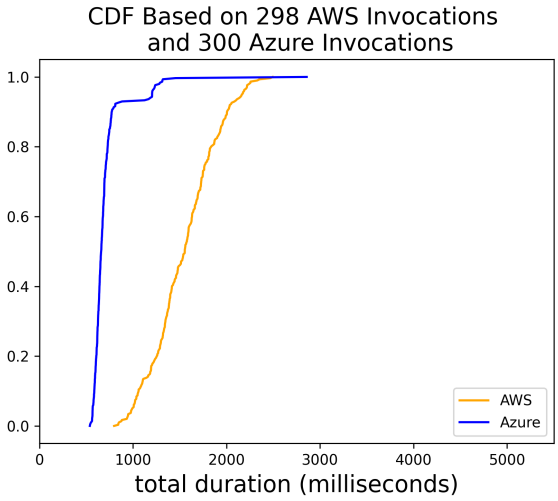
Figure 4.2: Violin plot of total duration breakdown based on three constant workloads with **all invocations**

Another observation is that *the storage triggering overhead (t6t7) is the most significant contributor to the overall performance (total duration) for benchmark applications on both platforms as well as the difference between them.* This contrasts the fact that traditional performance characteristics such as CPU/memory and network are primarily studied in academic literature while platform overhead, mainly cold starts, gains most attention in grey literature and industry [41]. This result shows that storage triggering overhead can substantially impact the overall performance if used in a serverless application, which explains why prior findings have different insights due to the lack of storage triggering in their benchmarks.

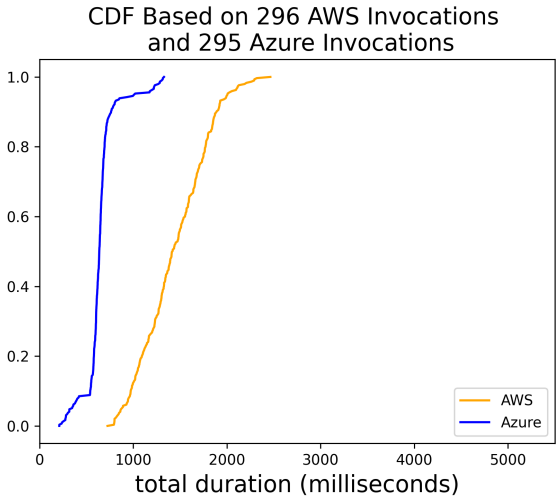
Furthermore, Azure shows higher variability than AWS in the face of increasing RPS. The variability of Azure performance is mainly caused by function startup overhead (t_2t_3).



(a) Constant1 workload



(b) Constant2 workload



(c) Constant3 workload

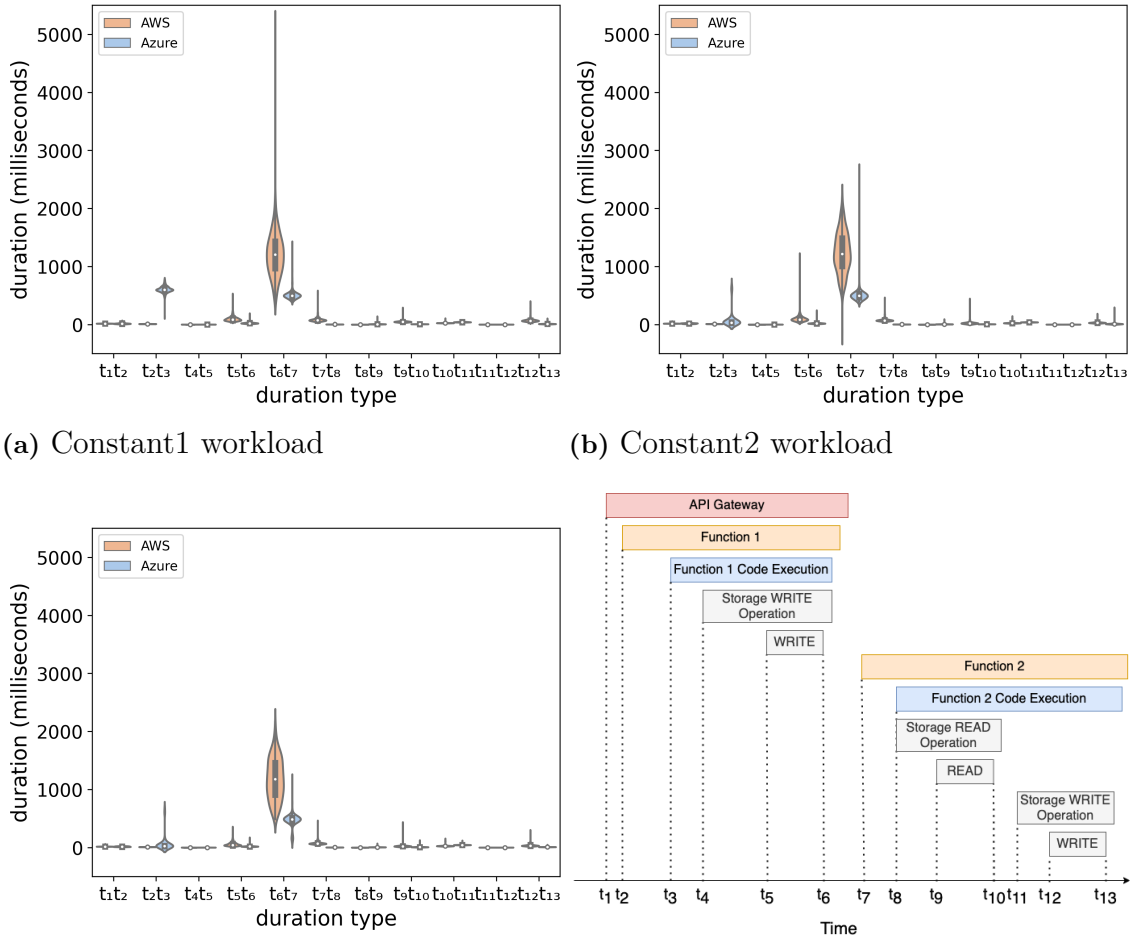
Figure 4.3: CDF plot of total duration based on three **constant** workloads with **only warm invocations**

4.2 Bursty Workload

This section presents the results of the experiment for all three bursty workloads BWL1-BWL3 described in Section 3.2.3.2.

4.2.1 All Invocations

Figure 4.5 presents the CDF of total duration for the three bursty workloads while detailed statistics summary for total duration in milliseconds (ms) are shown in Table 4.4, 4.5, 4.6. Similar as the constant workload scenario, Azure still outperforms AWS for overall performance in general except for some extreme outliers in BWL2 and BWL3. However, both AWS and Azure show significant performance degrada-



(a) Constant1 workload

(b) Constant2 workload

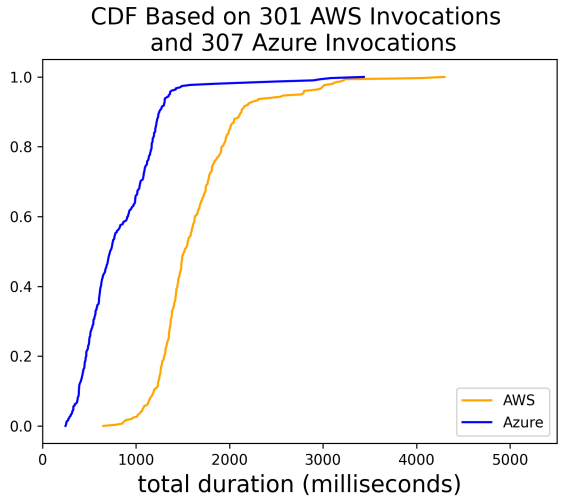
(c) Constant3 workload

(d) Trace design reference

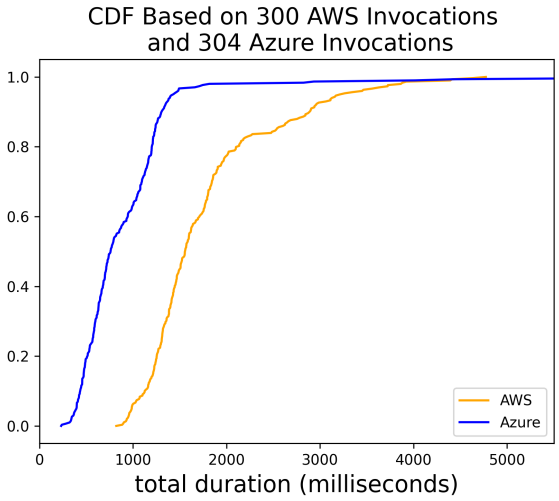
Figure 4.4: Violin plot of total duration breakdown based on three **constant** workloads with **only warm invocations**

tion, which is different from the constant workload case. Take 99th percentile for example, the highest values in constant workload is 2504 ms (NBWWL1 for AWS), which is even lower than the lowest value in bursty workloads (2873 ms). This is the same when RPS increases and both platforms show significant increase in total duration. However, Azure increases much faster in 99th and maximum for total duration, which causes Azure to lose advantage over AWS in BWL2 and BWL3. Take 99th percentile for example, AWS increases from 3220 ms to 4128 ms while Azure increases from 2873 ms to 4191 ms. The difference between them decreases from 862 ms to -63 which means AWS starts to outperform Azure.

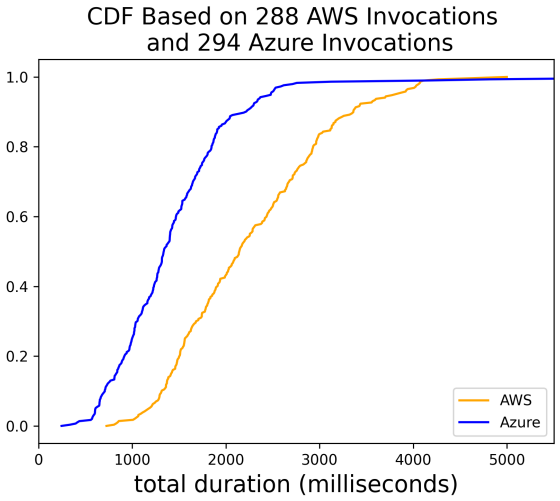
For the detailed breakdown of total duration, Figure 4.6 presents the Volin plot for the three bursty workloads with all invocations. Similar to the constant workload, storage triggering overhead t_6t_7 is still the biggest contributor to total duration and the difference between AWS and Azure. However, function startup overhead (t_2t_3 and t_7t_8) shows an increased impact on the overall performance. Azure performance for t_2t_3 drops sharply in bursty workload, and the same applies to AWS in t7t8. With



(a) BWL1 workload



(b) BWL2 workload



(c) BWL3 workload

Figure 4.5: CDF plot of total duration based on three **bursty** workloads with **all invocations**

increasing RPS, storage operations (t_8t_9, t_9t_{10}, $t_{12}t_{13}$) and resizing computation ($t_{10}t_{11}$) for AWS and Azure also increase with higher median and tail latency.

4.2.2 Warm Invocations

Considering only warm invocations, both AWS and Azure, as shown in Figure 4.7, decrease significantly in total duration compared to all invocations cases, which is the same as the constant workload scenario. However, with increasing RPS, the difference between AWS and Azure decreases accordingly. Take 99th percentile for example, the difference decreases from 834 to -90 which means AWS starts to outperform Azure.

	All Invocations			*Warm Invocations*		
	AWS	Azure	Difference	AWS	Azure	Difference
Median	1518	723	795	1487	700	787
95th percentile	2768	1360	1408	2108	1305	803
99th percentile	3220	2873	347	2266	1432	834
Maximum	4299	3437	862	2979	1582	1397

Table 4.4: Statistics summary of total duration in milliseconds for **BWL1 workload**

	All Invocations			*Warm Invocations*		
	AWS	Azure	Difference	AWS	Azure	Difference
Median	1556	761	795	1470	739	731
95th percentile	3214	1429	1785	2261	1356	805
99th percentile	4393	3986	407	2654	2185	469
Maximum	4773	6396	-1623	3112	4019	-907

Table 4.5: Statistics summary of total duration in milliseconds for **BWL2 workload**

	All Invocations			*Warm Invocations*		
	AWS	Azure	Difference	AWS	Azure	Difference
Median	2143	1337	806	1678	1326	352
95th percentile	3803	2480	1323	2388	2342	46
99th percentile	4128	4191	-63	2566	2656	-90
Maximum	4998	10179	-5181	2793	3122	-329

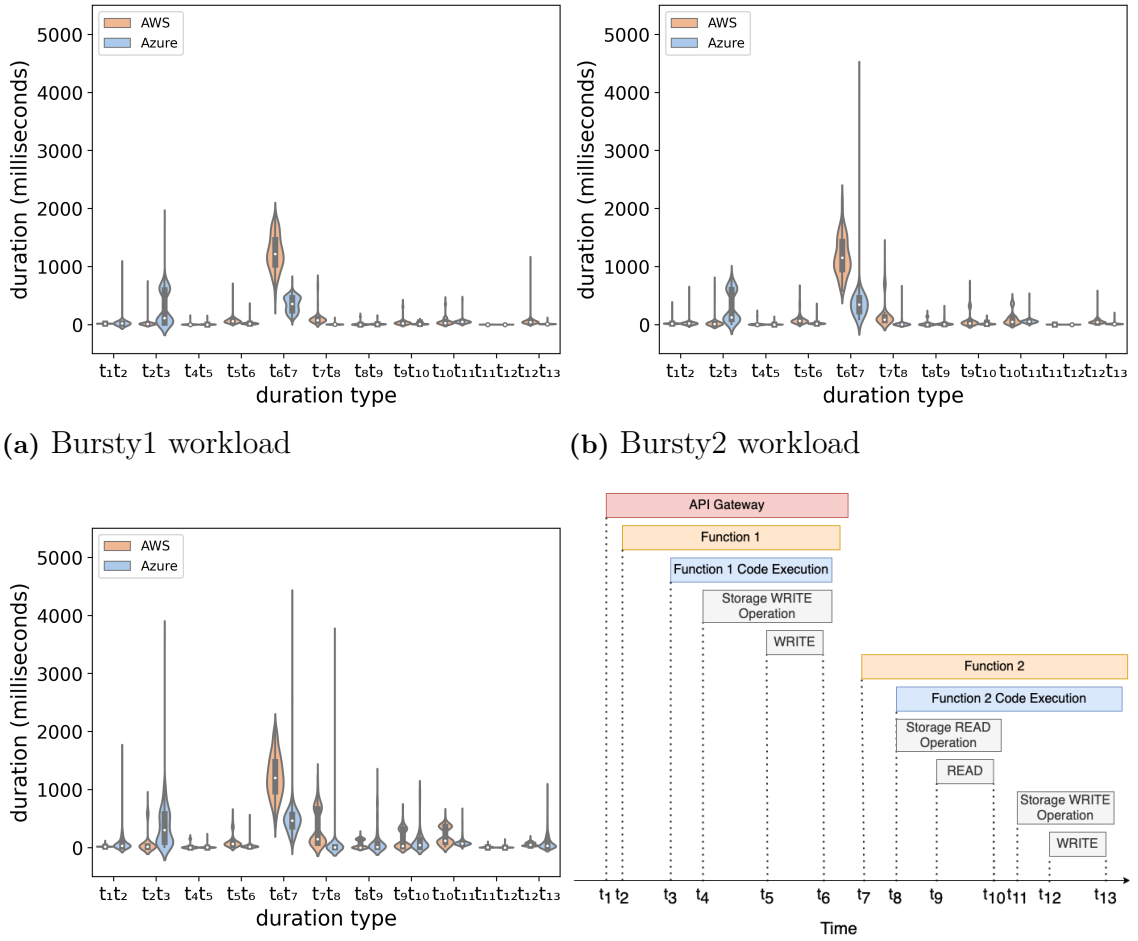
Table 4.6: Statistics summary of total duration in milliseconds for **BWL3 workload**

For detailed breakdown segments of total duration shown in Figure 4.8 with Volin plot, Azure performance for t_2t_3, t_8t_9, t_9t_{10}, and $t_{12}t_{13}$ decreases with increasing RPS while AWS keeps stable performance for most of the segments.

4.2.3 Discussion

As the analysis of the results shown in the previous two sections 4.2.1 and 4.2.2, Azure still outperforms AWS in general for overall performance regardless of considering all invocations or only warm invocations. The storage triggering overhead (t_6t_7) is still the biggest contributor to the overall performance (total duration) for benchmark applications on both platforms and the difference between them.

However, AWS and Azure show degraded performance in bursty workload compared to constant workload. This also applies when RPS increases. However, Azure performance decreases significantly faster than AWS, which causes the difference between



(a) Bursty1 workload

(b) Bursty2 workload

(c) Bursty3 workload

(d) Trace design reference

Figure 4.6: Violin plot of total duration breakdown based on three **bursty** workloads with **all invocations**

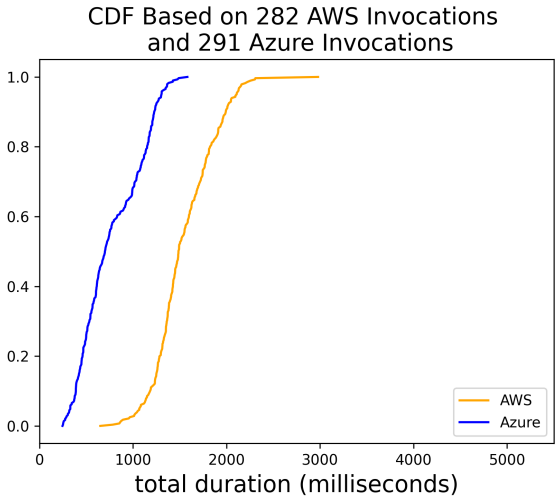
them to decrease, and AWS even outperforms Azure for tail latency in higher RPS (Bursty2 and Bursty3). This observation aligns with the finding from STeLLAR [46], which indicates Azure displays higher sensitivity to the burst size than AWS. Kuhlenkamp et al.[25] in their elasticity benchmarking of FaaS also find out that AWS can maintain stable performance in high bursty workload while Azure can not.

4.3 Threats to Validity

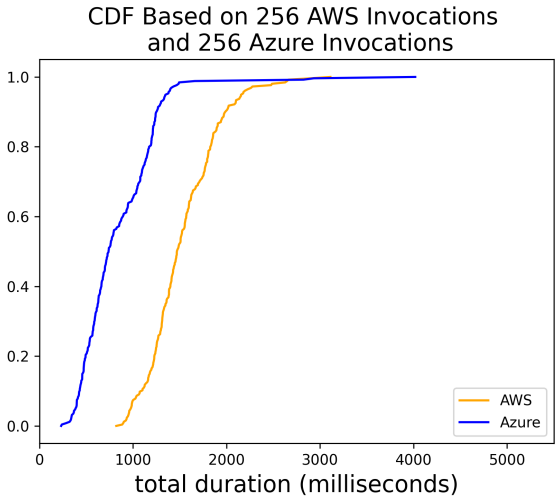
This section discusses the threats to construct, internal, and external validity that need to be considered in the context of this study.

4.3.1 Construct Validity

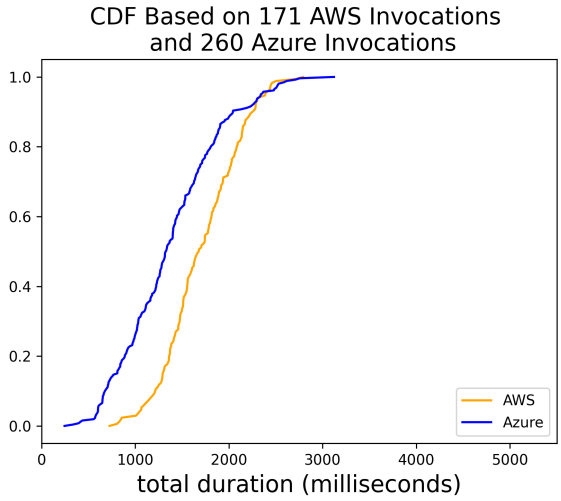
Construct validity refers to the extent to which the methodology and measurements adopted are relevant to the research questions. In the context of this thesis, main



(a) BWL1 workload



(b) Bursty2 workload

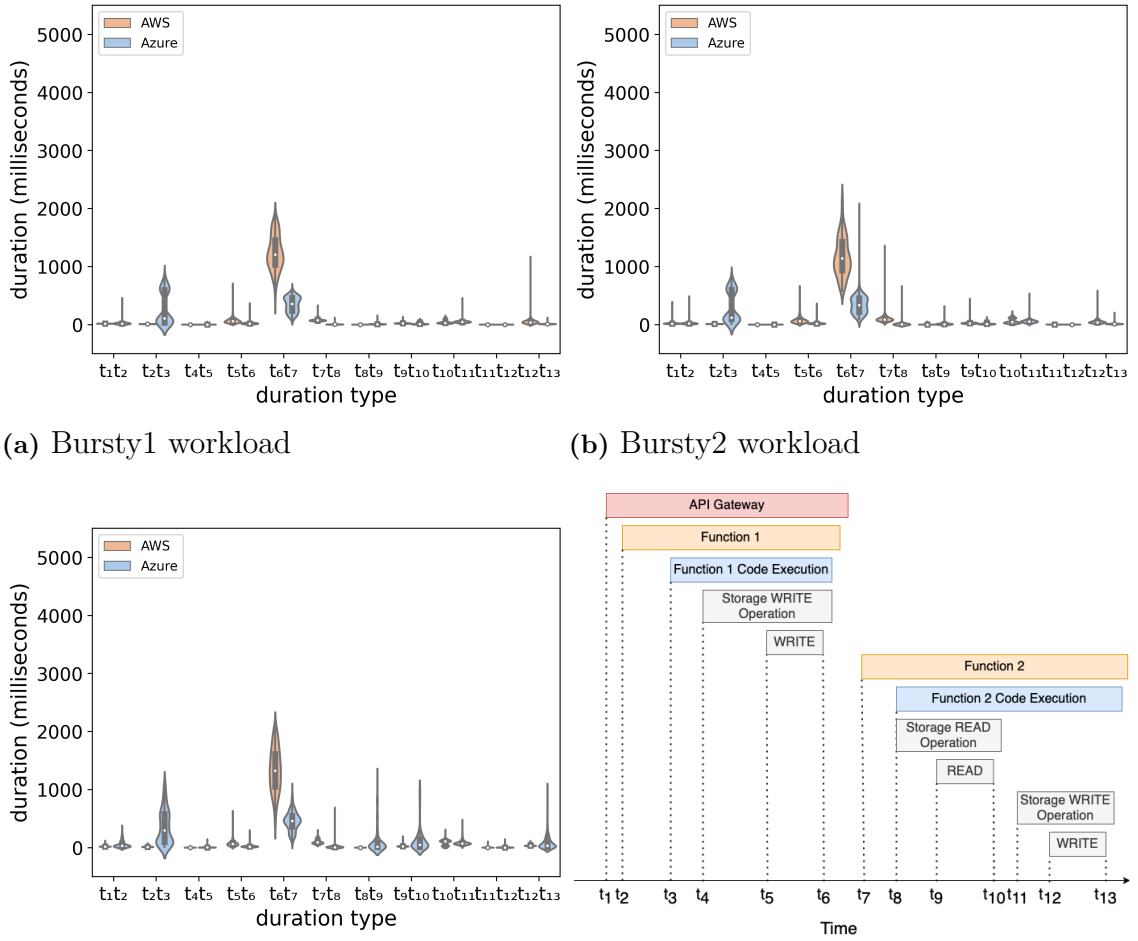


(c) Bursty3 workload

Figure 4.7: CDF plot of total duration based on three **bursty** workloads with **only warm invocations**

construct validity is about whether serverless application performance can be fairly compared.

In the benchmark design at Section 3.2, the major configurations that impact the performance are studied and addressed with clear solutions and motivation for improving the fairness. For example, the choice of runtime may affect the fairness of the comparison due to provider-specific features and support maturity for the runtime. In this thesis, .NET Core 3.1 with C# may give Azure an advantage in reduced code start for the second function invocation because of its default in-process mechanism. However, the result from experiments shows that its impact on the overall performance and comparison is limited since cold start latency accounts for a small part of end-to-end duration in the case of storage triggers being used.



(a) Bursty1 workload

(b) Bursty2 workload

(c) Bursty3 workload

(d) Trace design reference

Figure 4.8: Violin plot of total duration breakdown based on three **bursty** workloads with **only warm invocations**

4.3.2 Internal Validity

Internal validity refers to the extent to which the cause-and-effect relation is trustworthy and can not be explained by other factors or variables. This threat is generally the primary concern in Cloud benchmarking or performance testing and especially challenging for serverless benchmarking since underlying infrastructure is abstracted away and can almost be treated as a black box for end users [26].

The primary threat to validity is the unknown factors that may impact the performance. In this thesis, primary academic literature related to serverless benchmark is reviewed together with grey literature such as blogs and documentation to understand as many known factors/parameters as possible and address them for AWS and Azure. However, minimizing such threat to validity needs community effort to address it continuously with collective findings [44].

Another threat to internal validity is that cloud providers change their services

and infrastructure over time. That results in the insights we gained about the cause of performance difference between AWS and Azure may change over a certain period. This requires that the benchmark execution in the thesis must be executed at the same time against the cloud providers. To mitigate this, these experiments are performed repetitively during the study to update the datasets and review the analysis. The benchmark should be maintained and updated constantly with follow-up experiments and data analysis in future work.

4.3.3 External Validity

External validity refers to the extent to which the results from the study can be generalized, and the conclusion can hold throughout the study domain [48]. The two major threats to external validity in the context of this study are how the insights generated from comparing serverless application performance in AWS and Azure can be applied to other Cloud platforms and serverless applications.

This study is limited to AWS and Azure due to the time limit. However, the two cloud providers are the two biggest in the market for general cloud computing. In terms of serverless adoption, they represent even 90% of serverless applications in an extensive study on the status of serverless [10]. However, the results and conclusion need to be validated for other cloud providers, which is feasible with manageable effort using the methodology presented in this thesis.

Another limitation is that this thesis uses a single serverless application as the benchmark. This limits the coverage of serverless components and architecture patterns and thus generates a limited amount of insights. Additional benchmarks, including other triggering mechanisms (e.g., queue trigger), storage type (e.g., serverless database), serverless orchestration, are needed to be studied since their interaction with other services may also impact the overall performance as well as the difference between cloud providers.

In general, using a broad range of diverse serverless applications to perform benchmarking across more cloud providers can significantly improve the generalizability of the insights developed from the benchmarking.

5

Related Work

This chapter presents the related work in serverless benchmarking mainly from the academic community. These works present well-structured benchmarking design and refined insights for serverless computing which provide references and domain knowledge for this thesis. The first section reviews the earlier works in general serverless benchmarking focusing on microbenchmarking while the second section focuses on the main works in application-centric benchmark published in recent years, which are most related to this thesis. The difference and contribution made by this thesis are also discussed together with each group of related studies.

5.1 Serverless Benchmarking

Prior serverless benchmarking works both in academic and grey literature mainly evaluate the performance of Function as a Service (FaaS) (i.e., AWS Lambda, Azure function, Google Cloud Functions) which is the major computing component of a serverless application. According to the comprehensive literature review performed by Scheuner and Leitner [41], most studies use FaaS micro-benchmarks composed by single-function meanwhile, CPU performance and platform overhead (i.e., cold start) are the most studied characteristics. Wang et al. [47] conducted one of the first comprehensive serverless measurement studies on the three leading cloud providers (AWS, Azure, and Google) by integrating all necessary instrumentation into a single function called measurement function to collect metrics such as invocation time and runtime information. Their results show that AWS Lambda has the lowest cold start latency (the time to provision a new function instance). A recent serverless benchmarking study, FaaSdom [28], evaluates metrics such as call latency (round-trip), cold start, and throughput of FaaS platforms from AWS, Azure, Google, and IBM. It reveals that AWS Lambda is the best-performing cloud provider with the lowest call latency, average cold start latency, and most stable response.

Real-world serverless applications can consist of multiple functions integrating with other types of serverless components (storage, queue, API, event bus, messaging service, workflow orchestrator, etc.). To provide relevant reference and guidance to serverless practitioners, this thesis uses a realistic real-world serverless application as a benchmark to study the overall application performance and the detailed break-down segments making up it. The results of this thesis reveal that AWS can perform significantly worse than Azure with other components involved in the application despite the performance advantage from individual AWS Lambda functions.

5.2 Serverless Application Benchmarking

Existing application-oriented benchmarking works propose comprehensive benchmarking frameworks along with various applications as benchmarks to measure different aspects of application performance. They also present experiments results with these benchmarks to demonstrate the frameworks.

PanOpticon [45] looks into the performance impact from function chaining and the choice of triggers. To demonstrate the support of various chaining mechanisms in AWS and Google Cloud, applications with chain length 2 are deployed to AWS and Google, respectively, to measure the latency of each chain mechanism. However, the results only compare a different triggering mechanism within the cloud provider. To demonstrate the effectiveness of comparing platforms, a sample serverless chat server application, covering HTTP trigger, database access, pub/sub messaging, and storage trigger, is used for benchmarking against AWS and Google Cloud. Nevertheless, it only compares execution time and memory usage at the individual function level for the application with average latency. In contrast, this thesis compares the end-to-end execution time with detailed breakdown segments, and the results are based on an analysis of total distribution.

ServerlessBench [49] identifies a set of critical metrics in serverless computing, such as communication latency, startup latency, and stateless execution. It evaluates AWS Lambda, two open-source serverless platforms (Open Whisk and Fn), and one private cloud (Ant Financial) against these metrics using micro-benchmark and application benchmark. However, ServelessBench focuses on mainly analyzing the underlying implication of serverless computing while this thesis focuses more on performance differences between cloud platforms.

STeLLAR [46] supports detailed measurements for per-component performance, such as data communication delays for chained functions. It reveals that storage accesses (retrieval of function images during the function instance startup and inter-function data communication that happens via a storage service) and bursty workload are among the two largest contributors to tail latency. Although aligned with the impact of storage access and bursty workload, this thesis also finds that the storage triggering has an even more significant impact on both the overall application performance and the difference across cloud platforms. In addition, STeLLAR analyzes data transfer delays against only AWS and Google due to the missing support for Go runtime at the time of its writing. In contrast, this thesis focuses on AWS and Azure, which are the two largest cloud providers, and the finding related to storage thus represents broader serverless usage.

BeFaaS [21] provides a built-in e-commerce application benchmark and supports drill-down analysis with fine-grinded measurements. The evaluation of the BeFaaS framework using the e-commerce benchmark investigates computing, networking

transmission, and database query latency (involving two functions and database operations). The results show that network transmission is the most time-consuming factor. However, other common characteristics such as triggering and cold starts are not evaluated.Furthermore, BeFaaS is the only study that mentions fairness as a benchmark framework requirement. Still, it does not discuss how the fairness of performance comparison is addressed in its benchmark. In contrast, this thesis addresses fairness throughout the benchmark design. The benchmark used in this thesis provides detailed traces about the two most common triggers (HTTP and Storage), cold starts overhead, storage access as well as computing.

6
Conclusion

This thesis designed and implemented an application benchmark targeting AWS and Azure with a continuous focus on fairness (to answer RQ1). The benchmark has been integrated with the state-of-art serverless benchmarking tool, Serverless Benchmarker (SB), for standardizing and automating the benchmarking tests. The resulting benchmark has been operated with two representative workload patterns (constant and bursty) and generated detailed measurements making up the end-to-end duration of the benchmark application invocations. The drill-down analysis has been performed to develop insights for answering RQ2 and RQ3.

RQ1: How to design a real-world serverless benchmark application that is fair to compare performance across heterogeneous cloud platforms?

Fairness is a highly subjective matter which can always lead to divergent opinions. The challenge is also compounded with the provider-specific implementation on Cloud platforms. Therefore, building a good benchmark with a trustful fairness level is not a one-time activity. As described in Section 3.1, this thesis stresses that *building a "fair" benchmark is a continuous process and journey* and proposes a methodology with an Agile approach to address this challenge continuously. Furthermore, the motivation of different design choices and configurations in the benchmark must be transparent and informative to create trust with the community.

More specifically, this thesis breaks down the fairness challenge into following the three sub-questions and manages to address them with clear solutions.

RQ1.1 How can the architecture and configuration of serverless applications be mapped closely across cloud providers?

Benchmark application architecture should adopt comparable serverless components and configurations with caution about default settings. For example, by default, the Azure blob storage trigger uses a "pulling" mechanism causing high latency and should be replaced with an Event trigger to match AWS. Azure does not allow granular configuration on function memory size and allocates up to 1.5 GB for each instance if choosing a Consumption plan. To match this, AWS Lambda needs to be configured with 1769 MB memory, equivalent to one full vCPU.

In addition, benchmark application architecture should also consider realistic usage and maturity. For example, by default, Azure uses Windows as the underlying op-

erating system to switch to Linux. Choosing Linux for the Azure function appears fairer since AWS Lambda only uses Linux. However, Linux support for the Azure function has less maturity and adoption than Windows, while AWS creates its own optimized Linux distribution (Amazon Linux 2) for Lambda. Therefore, using Windows for the Azure function is a more fair choice.

RQ1.2: How can the instrumentation be implemented to provide comparable information across cloud providers?

Instrumentation design needs to identify interesting timestamps that can be collected across cloud providers to serve the goal of benchmarking. This requires understanding distributed tracing tools provided by cloud providers to know what data can be collected for each platform and what instrumentation can be performed. For example, AWS provides the underlying function cold start time through the default Initialization segment while this information is unavailable in Azure. To have a comparable measurement for function startup overhead, this thesis uses the time between function triggering time (out-of-box from distributed tracing tools from both cloud providers) and the execution time of the first line code in function code through instrumentation.

RQ1.3: How can the workload be designed and executed to minimize the divergence of load test?

The workloads need to be realistic and represent real-world serverless usage. Based on the finding of production usage statistics from one of the major cloud providers, this thesis designs two workload models: constant and bursty. Each has three variations with different RPS patterns.

The workloads can be defined and specified using a load testing tool (e.g., K6, supported by Serverless Benchmarker used in this thesis) and shared across benchmark applications. Such load testing tools generate reproducible load traffic that meets the specification.

RQ2: How does Serverless application performance differ across different Cloud providers?

Serverless applications generally perform better in Azure than AWS. However, AWS is better at handling increasing load and bursty workload while Azure shows higher variability in this case.

RQ3: Why does serverless application performance differ across Cloud providers?

Storage triggering overhead accounts for the largest part of the end-to-end duration for both AWS and Azure. It is also the biggest contributor to the performance difference between AWS and Azure since AWS has significantly longer latency than Azure.

However, Azure shows variable performance for function startup overhead and storage access in the face of RPS increasing and bursty workload while AWS keeps relatively stable performance for this segment, which explains the variability of the performance difference.

6.1 Future Work

The future work can focus on continuing mitigating the threats to validity described in Section 4.3.

More improvements on fairness can be investigated for mitigating construct validity. For example, due to the maturity of support on distributed tracing in Azure at the time of writing, this thesis uses .Net Core 3.1 with C# as a runtime to write the functions, and this gives Azure a slight advantage in reducing the number of cold starts for the second function. This can be improved by using Python or NodeJS in the future, which has comparable runtime behaviors on both platforms as well as representing a larger audience in serverless communities [11].

For mitigating internal validity, the benchmark application and results should be continuously updated over time in the future to keep up with provider changes.

For mitigating external validity, future work can first extend the support for more major cloud providers such as Google Cloud and Alibaba Cloud[1] in order to verify if the insights can be generalized across a broad range of major cloud providers. To address the single benchmark limitation of this study, more benchmark applications covering both more use cases and architecture patterns can be created using the methodology in the thesis to generate potential new insights.

[1] `https://us.alibabacloud.com/en`

Bibliography

[1] Michael Armbrust et al. “A view of cloud computing”. In: *Commun. ACM* 53.4 (2010), pp. 50–58.

[2] AWS. *AWS Lambda Pricing*. Last accessed 16 November, 2021. 2021. URL: https://aws.amazon.com/lambda/pricing/.

[3] AWS. *AWS Lambda Supports C#*. Last accessed 15 December, 2021. 2021. URL: https://aws.amazon.com/about-aws/whats-new/2016/12/aws-lambda-supports-c-sharp/.

[4] AWS. *Configuring Lambda function options*. Last accessed 15 January, 2022. 2021. URL: https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html.

[5] AWS. *Serverless on AWS*. Last accessed 24 November, 2021. 2021. URL: https://aws.amazon.com/serverless/.

[6] AWS. *Using AWS Lambda with other services*. Last accessed 20 December, 2021. 2021. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html.

[7] Paul C. Castro et al. “The rise of serverless computing”. In: *Commun. ACM* 62.12 (2019), pp. 44–54.

[8] Dheeraj Chahal et al. “Performance and Cost Comparison of Cloud Services for Deep Learning Workload”. In: *ICPE (Companion)*. ACM, 2021, pp. 49–55.

[9] Marcin Copik et al. “SeBS: a serverless benchmark suite for function-as-a-service computing”. In: *Middleware*. ACM, 2021, pp. 64–78.

[10] Simon Eismann et al. “Serverless Applications: Why, When, and How?” In: *IEEE Softw.* 38.1 (2021), pp. 32–39.

[11] Simon Eismann et al. “The State of Serverless Applications: Collection, Characterization, and Community Consensus”. In: *IEEE Transactions on Software Engineering* (2021), pp. 1–1. DOI: 10.1109/TSE.2021.3113940.

[12] Simon Eismann et al. *The State of Serverless Applications: Collection, Characterization, and Community Consensus - Replication Package*. Zenodo, 2021. DOI: 10.5281/zenodo.5185055. URL: https://doi.org/10.5281/zenodo.5185055.

[13] Erwin Van Eyk et al. “Beyond Microbenchmarks: The SPEC-RG Vision for a Comprehensive Serverless Benchmark”. In: *ICPE Companion*. ACM, 2020, pp. 26–31.

[14] Erwin Van Eyk et al. “The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms”. In: *IEEE Internet Comput.* 23.6 (2019), pp. 7–18.

[15] Enno Folkerts et al. “Benchmarking in the Cloud: What It Should, Can, and Cannot Be”. In: *TPCTC*. Vol. 7755. Lecture Notes in Computer Science. Springer, 2012, pp. 173–188.

[16] Gartner. *Gartner Predicts the Future of Cloud and Edge Infrastructure*. Last accessed 24 November, 2021. 2021. URL: gartner.com/smarterwithgartner/gartner-predicts-the-future-of-cloud-and-edge-infrastructure/.

[17] Gartner. *Gartner Says Worldwide IaaS Public Cloud Services Market Grew 40.7% in 2020*. Last accessed 28 January, 2022. 2021. URL: https://www.gartner.com/en/newsroom/press-releases/2021-06-28-gartner-says-worldwide-iaas-public-cloud-services-market-grew-40-7-percent-in-2020/.

[18] Google. *Compare AWS and Azure services to Google Cloud*. Last accessed 20 December, 2021. 2021. URL: https://cloud.google.com/free/docs/aws-azure-gcp-service-comparison?utm_source=google&utm_medium=blog&utm_campaign=FY21-Q2-Product-Mapping-Blog&utm_content=documentation.

[19] Google. *Google Cloud Functions Pricing*. Last accessed 16 November, 2021. 2021. URL: https://cloud.google.com/functions/pricing.

[20] Google. *Serverless Computing*. Last accessed 24 November, 2021. 2021. URL: https://cloud.google.com/serverless.

[21] Martin Grambow et al. “BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms”. In: *CoRR* abs/2102.12770 (2021).

[22] Wilhelm Hasselbring. “Benchmarking as Empirical Standard in Software Engineering Research”. In: *EASE*. ACM, 2021, pp. 365–372.

[23] Jeongchul Kim and Kyungyong Lee. “FunctionBench: A Suite of Workloads for Serverless Cloud Function Service”. In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019, pp. 502–504. DOI: 10.1109/CLOUD.2019.00091.

[24] Jóakim von Kistowski et al. “How to Build a Benchmark”. In: *ICPE*. ACM, 2015, pp. 333–336.

[25] Jörn Kuhlenkamp et al. “Benchmarking elasticity of FaaS platforms as a foundation for objective-driven design of serverless applications”. In: *SAC*. ACM, 2020, pp. 1576–1585.

[26] Christoph Laaber, Joel Scheuner, and Philipp Leitner. “Performance testing in the cloud. How bad is it really?” In: *PeerJ Prepr.* 6 (2018), e3507.

[27] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. “Evaluation of Production Serverless Computing Environments”. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 442–450. DOI: 10.1109/CLOUD.2018.00062.

[28] Pascal Maissen et al. “FaaSdom: a benchmark suite for serverless computing”. In: *DEBS*. ACM, 2020, pp. 73–84.

[29] Johannes Manner et al. “Cold Start Influencing Factors in Function as a Service”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018, pp. 181–188. DOI: 10.1109/UCC-Companion.2018.00054.

[30] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. en. 2011. DOI: https://doi.org/10.6028/NIST.SP.800-145.

[31] Microsoft. *Announcing general availability of Azure Functions*. Last accessed 20 December, 2021. 2021. URL: https://azure.microsoft.com/en-us/blog/announcing-general-availability-of-azure-functions/.

[32] Microsoft. *Azure Blob storage trigger for Azure Functions*. Last accessed 20 December, 2021. 2021. URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob-trigger?tabs=csharp.

[33] Microsoft. *Function apps running on Linux App Service Plans hitting open file descriptors limit*. Last accessed 28 January, 2022. 2020. URL: https://github.com/Azure/azure-functions-host/issues/6466.

[34] Microsoft. *Microsoft Azure Functions Pricing*. Last accessed 16 November, 2021. 2021. URL: https://azure.microsoft.com/en-us/pricing/details/functions/.

[35] Microsoft. *Supported languages in Azure Functions*. Last accessed 20 December, 2021. 2021. URL: https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages#languages-by-runtime-version/.

[36] Microsoft. *The Azure Functions on Linux Preview*. Last accessed 28 January, 2022. 2021. URL: https://azure.github.io/AppService/2017/11/15/The-Azure-Functions-on-Linux-Preview.html/.

[37] Mirosoft. *Azure Serverless*. Last accessed 24 November, 2021. 2021. URL: https://azure.microsoft.com/en-us/solutions/serverless/#solutions..

[38] Haoran Qiu et al. "FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices". In: *OSDI*. USENIX Association, 2020, pp. 805–825.

[39] Ali Raza et al. "SoK: Function-as-a-Service: From An Application Developer's Perspective". In: *Journal of Systems Research - Mar 2021*. 2021. URL: https://openreview.net/forum?id=VdWaMgaTKtX.

[40] Raja R. Sambasivan et al. "So , you want to trace your distributed system ? Key design insights from years of practical experience". In: 2014.

[41] Joel Scheuner and Philipp Leitner. "Function-as-a-Service performance evaluation: A multivocal literature review". In: *J. Syst. Softw.* 170 (2020), p. 110708.

[42] Mohammad Shahrad et al. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 205–218. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

[43] Mikhail Shilkov. *Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP*. Last accessed 6 February, 2022. 2021. URL: https://mikhail.io/serverless/coldstarts/big3/.

[44] S.E. Sim, S. Easterbrook, and R.C. Holt. "Using benchmarking to advance research: a challenge to software engineering". In: *25th International Conference on Software Engineering, 2003. Proceedings.* 2003, pp. 74–83. DOI: 10.1109/ICSE.2003.1201189.

[45] Nikhila Somu et al. “PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications”. In: *COMSNETS*. IEEE, 2020, pp. 144–151.

[46] Dmitrii Ustiugov, Theodor Amariucai, and Boris Grot. “Analyzing Tail Latency in Serverless Clouds with STeLLAR”. In: *2021 IEEE International Symposium on Workload Characterization (IISWC'21)*. Institute of Electrical and Electronics Engineers (IEEE), 2021.

[47] Liang Wang et al. “Peeking Behind the Curtains of Serverless Platforms”. In: *USENIX Annual Technical Conference*. USENIX Association, 2018, pp. 133–146.

[48] Hyrum K. Wright, Miryung Kim, and Dewayne E. Perry. “Validity concerns in software engineering research”. In: *FoSER*. ACM, 2010, pp. 411–414.

[49] Tianyi Yu et al. “Characterizing serverless platforms with serverlessbench”. In: *SoCC*. ACM, 2020, pp. 30–44.

[50] Vladimir Yussupov et al. “Facing the Unplanned Migration of Serverless Applications: A Study on Portability Problems, Solutions, and Dead Ends”. In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2019)*. ACM, Dec. 2019, pp. 273–283. DOI: 10.1145/3344341.3368813.