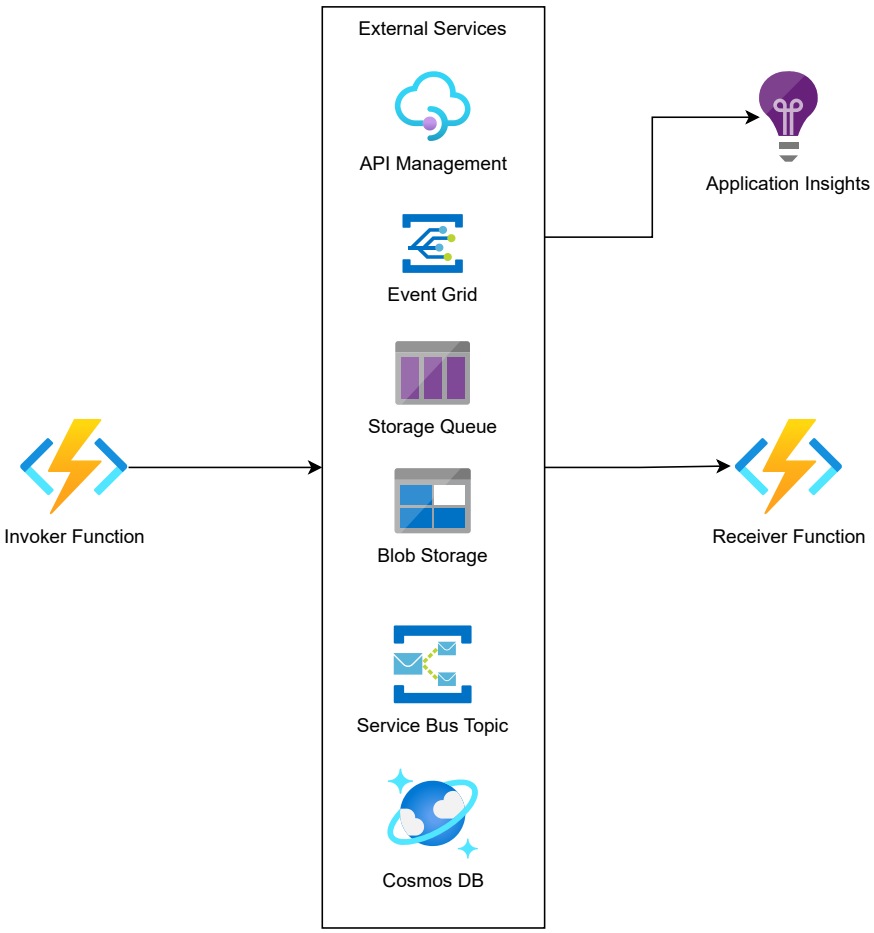
CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG



Serverless Function Triggers in Azure

An Analysis of Latency and Reliability

Master's thesis in Computer science and engineering

Henrik Lagergren & Henrik Tao

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

Master's thesis 2022

Serverless Function Triggers in Azure

An Analysis of Latency and Reliability

Henrik Lagergren
Henrik Tao

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

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

Serverless Function Triggers in Azure
An Analysis of Latency and Reliability
Henrik Lagergren
Henrik Tao

Supervisor: Joel Scheuner, Department of Computer Science and Engineering
Advisor: Christoffer Noring, Cloud Advocate Lead, Microsoft
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 visualization of the flow within Microsoft Azure of a function and its correlated function trigger.

Typeset in LaTeX
Gothenburg, Sweden 2022

Serverless Function Triggers in Azure
An Analysis of Latency and Reliability
Henrik Lagergren
Henrik Tao
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Serverless computing has seen a very rapid growth in popularity the recent years, where businesses are now able to completely outsource their IT infrastructure through cloud providers. However, some practitioners that are sensitive to latency and reliability might suffer from unwanted effects when deploying their IT infrastructure to the cloud. This thesis intends to investigate the latency and reliability of various Azure Function triggers offered by Microsoft Azure, one of the most popular cloud providers. In order to conduct such an investigation, a solid benchmark was designed and implemented to test the performance of seven different triggers on two runtimes. The findings, based on the results, from this thesis show that various trigger types have major differences in latency compared to each other. The choice of runtime does also have an impact on latency. However, the impact of the runtime is not as important compared to the choice of trigger type. For bursty workloads, increasing the size of invocation bursts tends to cause longer tail latency for the triggers. The HTTP and Event Hub triggers perform the best, where the shortest and most stable latency was observed for Event Hub on all different burst sizes, while HTTP had some latency increases at the heavier burst sizes. The undoubtedly worst performing trigger was Blob storage. For out-of-order event deliveries, an inter-arrival time of 250ms will lower the risk of high occurrence of out-of-order. To fully ensure ordering, higher invocation delays had to be tested to pinpoint the optimal delay. Missing event deliveries were most apparent for two trigger types, and the other types had too few missing deliveries to draw any conclusions. Duplicate event deliveries were absent for three out of seven trigger types. Results suggest that there might be a difference between runtimes for duplicate deliveries depending on trigger type.

Keywords: Serverless computing, function-as-a-service, function triggers, benchmark, distributed tracing, latency, reliability.

Acknowledgements

We want to thank our supervisor Joel Scheuner for all of the time and effort he has put into this thesis. Joel has an immense passion for what he is doing and his knowledge within the field is exceptional. The feedback and insights he has provided us with have been immensely valuable for us. The outcome of this thesis would not have been the same without his dedication and supervision.

We would also like to thank Chris Noring at Microsoft for all of the valuable feedback and help he provided during the thesis.

Henrik Lagergren & Henrik Tao, Gothenburg, June 2022

Contents

List of Figures

List of Tables

1
Introduction

Serverless Computing has during recent years seen rapid growth in popularity, which has introduced a new paradigm shift. The traditional way for businesses was to set up and maintain their IT infrastructure on site, but the modern way instead leveraging responsibilities to cloud providers such as Microsoft Azure, Amazon Web Services (AWS), and Google Cloud. By leveraging the responsibilities, businesses benefit in terms of e.g. potential cost-savings, near-infinite scalability, and reduced operational efforts [1], [2]. Developers no longer need to struggle with time-consuming IT infrastructure tasks such as maintaining servers, allocating memory resources, and availability, Serverless Computing are covered by cloud providers. Instead, developers at IT businesses can focus more on the logic, where often their business value lies. The cloud providers each offer a range of cloud services, which their customers can use to develop and scale their applications. Further, a wide variety of popular language runtimes are often also supported within the cloud provider to make the startup or transition to serverless as effortless as possible [2].

Function as a Service (FaaS), which is a category within Serverless Computing, uses an event-driven model where uploaded code, called functions, is invoked by various trigger types, e.g. HTTP and queue trigger, on arriving events to their resources [3]. Microsoft Azure offers 12 different trigger types in total, where most of them also support input and output bindings [4]. The different trigger types do vary in their popularity, where some are far more used than others. By analyzing 89 serverless applications, Eismann et al. [1] found a consensus that the most common trigger types are HTTP and cloud events, which include queue, storage, and event triggers. However, one problem is that the underlying implementations of these popular triggers and the inherent latency drawback of cloud-based hosting applications, compared to servers on-premises, might make latency-critical applications unreliable when deployed to the cloud [5]. Further, in applications where the order of event delivery is an important factor, the choice of trigger types is crucial since some triggers do not guarantee event delivery, e.g. Event Grid and Queue storage in Azure [6], [7].

The focus on trigger types' latency in this thesis targets specific types of systems that have strict requirements in throughput, typically medium-sized to larger enterprise companies that e.g. sending and receiving millions of messages unobstructedly and quickly. Examples of such systems are social media platforms, and high-frequency trading systems [8]. Further, the focus on the reliability of event deliveries between triggers targets applications with emphasis on the reliability of the results from the

function execution. Examples of such systems are medical procedures or factories where medical logs of patients or building instructions must be registered correctly.

This thesis aims to provide insights into how the choice of triggers affects latency and the reliability of event deliveries for one of the major serverless provider, Microsoft Azure, assisting practitioners in decision-making for transitioning to serverless.

1.1 Purpose

The purpose of this study is to primarily investigate whether Azure Trigger types, in a certain runtime, have differing impacts on latency, to what extent the events are delivered out-of-order (OoO), and whether the deliveries are missed or delivered multiple times.

The results of this thesis should provide a foundation and deeper insights into the difference between various Azure trigger types regarding latency, reliability, and consistency of event deliveries, to help researchers and practitioners to decide whether certain trigger types are applicable for their intended purposes. Practitioners who mainly would benefit from this study are latency-critical enterprise companies that require the processing of massive throughput of data quickly and reliability-sensitive companies that are dependent on the outcome of event deliveries.

1.2 Research Questions

The following research questions are what this thesis intends to answer:

RQ1: How does the choice of trigger types affect the latency of invoking functions?

This research question intends to answer how different Azure trigger types, in a certain runtime, affect the latency of triggering. Latency data for all relevant trigger types will be collected and analyzed to get insights. To further investigate how different runtimes affect the latency, the following sub-questions will be answered:

- **RQ1.1**: How does the choice of runtime affect triggers' latency?
- **RQ1.2**: How does a bursty workload affect triggers' latency?
- **RQ1.3**: How does an inter-arrival time (IAT) controlled workload with a constant flow of trigger invocations affect triggers' latency?

Since various runtimes may vary in their language optimizations and implementations, sub-question (RQ1.1) intends to generate insights into how the choice of language runtime affects the latency using different trigger types. Further, the other two sub-questions (RQ1.2 and RQ1.3) address how latency is affected by trigger invocation patterns. A workload with short bursts of a various numbers of invocations, and another with an IAT controlled flow of invocations. An IAT-controlled workload

is where the delay between trigger invocations is manually configured to create a desired invocation pattern, more details in Section 4.2.4.

RQ2: How reliable and consistent are results of a function delivered?

This research question intends to answer how reliable and consistent invocations of various Azure Triggers are. The collected reliability data will be used to facilitate in-depth analysis and discussion post-experiment. Based on the different aspects of the question, it can be further divided into two sub-questions:

- **RQ2.1**: How frequent are function invocations OoO?
- **RQ2.2**: How frequent are function invocations missing or delivered multiple times?

These two sub-questions further specify the targeted aspects of reliability, namely, the order of event delivery, missed and duplicate event deliveries.

1.3 Scope

Due to the limited time frame of this thesis, it is not possible to address the available Azure Function Triggers within all five runtimes provided by Azure. This limits the scope of this thesis to at least execute the benchmark for all relevant available triggers in two different runtimes: Node.js and .NET. It is important to note that this thesis is based on an Azure student subscription and does not consider other offerings from Azure. The differences between subscriptions are mainly related to billing and management of Azure services. There is nowhere mentioned from Azure that the performance of services vary between subscriptions. Since it is not possible for the authors to fully guarantee, the findings and instructions from this thesis are limited to the student subscription.

1.4 Limitations and Delimitations

Even though the reproducibility of the conducted experiment is fully automated during the time this thesis is published, the benchmark can at any time be outdated due to changes in Microsoft Azure's infrastructure or their API. The findings from this thesis could therefore become obsolete due to future performance optimizations introduced by Microsoft Azure.

This master thesis will only investigate triggers within Microsoft Azure, which means that the result might not apply to other FaaS providers such as Amazon Web Services (AWS) or Google Cloud (GC). The reason for focusing on Azure is that it is the second most popular cloud provider, that similar studies have been done for AWS, and because of the limited time frame.

1.5 Outline

The structure of the thesis is as following:

- Chapter 2 introduces background knowledge and fundamental concepts that are essential to understanding the rest of this report. The chapter includes an introduction to cloud computing, serverless computing, function as a service, cloud services, infrastructure as code, distributed tracing, performance benchmarking, and load testing.
- Chapter 3 presents the related work towards this research area. The presented studies focus mostly on why latency and reliability are important factors within Serverless and how this thesis can learn from their methodologies.
- Chapter 4 presents the research methodology developed during the thesis for designing and executing the benchmark, data analysis, and unit tests.
- Chapter 5 presents the results from running the experiments, and observations from the results are summarized.
- Chapter 6 discusses the observations made from the results in the previous chapter, threats to validity of the findings, the implementation of the benchmark, and reproducibility.
- Chapter 7 concludes the thesis by summarizing the findings and answering the research questions. Furthermore, it suggests future work and improvements to the benchmark.

2
Background

This chapter introduces general knowledge and essential concepts to understand this thesis. It includes cloud computing, serverless computing, Function as a service triggers, infrastructure as code, distributed tracing, and load testing.

2.1 Cloud Computing

Cloud computing is the concept of hosting different services off-site on the cloud through the internet, which offers scalability in resources and costs [9]. The official definition of Cloud Computing by the National Institute of Standards and Technology (NIST) concisely summarizes the essence of Cloud Computing [10]:

> *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models [10].*

The five characteristics mentioned in the NIST definition consist of:

1. **On-demand self-service** that captures the consumer demands of unilateral provisioning of cloud computing resources.
2. **Broad Network Access** that enables the availability of cloud capabilities over the network on standardized client platforms.
3. **Resource Pooling** that encapsulates the essence of the internal infrastructural multi-tenant model of cloud providers' pooling mechanism, which allows the pooling of computing resources.
4. **Rapid Elasticity** that describes one of the main benefits of cloud computing, which is the availability and scaling of provisioned cloud resources to consumers' demands. The ability to rapidly (sometimes automatically) scale resources commensurate with demand to prevent under- and over-provisioning of resources is incredibly beneficial compared to traditional provisioning for which the consumer has to pay for unused and idle resources.
5. **Measured Service** that addresses the cloud systems' metering capabilities to automatically control and optimize resource use, but also provides both provider and consumer with transparency in resource usage of the utilized

service.

Furthermore, NIST refers to three service models that Cloud Computing usually is divided in: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). These models are essentially abstraction layers for what services and functionality cloud providers offer. The main difference between them is the distribution of management and responsibility of cloud resources among consumers and providers [10].

SaaS is the capability of cloud providers to provide out-of-the-box applications running on a cloud infrastructure. In this model, the consumer does not manage nor control any underlying resources, except for possible limited user-specific configuration settings, which can be seen to the right in Figure 2.1.

PaaS provides a platform that enables consumers to develop, deploy, run, and manage applications without the expertise and complexity of setting up and maintaining the underlying infrastructure. It allows the deployment of consumer-created applications using programming languages, tools, libraries, and services that are supported by the provider. In comparison to SaaS, PaaS gives the consumer control over the deployed application and data which can be seen in the middle of Figure 2.1.

IaaS is the lowest level and allows consumers to control e.g. operating systems, storage, and deployed applications, however, other parts of the underlying cloud infrastructure are still the responsibility of the provider, which can be seen to the left in Figure 2.1.

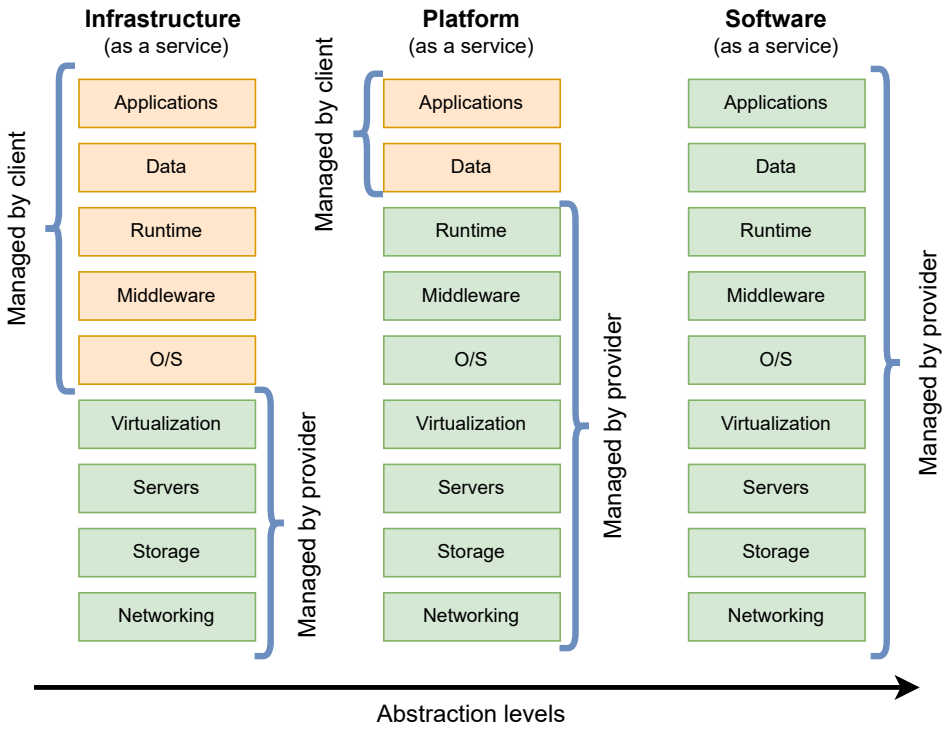


Figure 2.1: The client's and provider's responsibilities within the three cloud abstraction layers: SaaS, PaaS, and IaaS. The figure is reproduced with inspiration from similar illustrations of abstraction levels in cloud computing.

Lastly, NIST mentions the four deployment models, which are *private cloud*, *community cloud*, *public cloud*, and *hybrid cloud*, that addresses ownership and accessibility to the cloud infrastructure.

2.2 Serverless Computing

The fast emerging and compelling serverless paradigm for deployment and management of cloud applications are primarily due to the shift of enterprise application architectures to containers, virtual machines, microservices, the pay-as-you-go billing model, and elasticity of provisioning resources. Serverless Computing is according to Castro et al. [2] the natural path of progression for cloud computing considering recent advancements and adoption of virtual machines and containers.

Serverless computing is even closer to the original expectations of what cloud computing was supposed to be, namely, pay only for resources used, unlimited scalability, scaling down to zero, and abstracting from details of servers. However, serverless is complex since deploying applications to a serverless setting requires careful design

decisions in terms of e.g. quality-of-service monitoring, scaling, fault-tolerance properties, etc, where settings and availability might differ between cloud providers.

The challenges and limitations that are mentioned by Castro et al. concern "programming models" because of the orders of magnitude that for example, a simple video-streaming service might produce since it could run more than 150 serverless functions. Further, traditional tools for debugging and identification of bottlenecks are not applicable for serverless applications.

2.3 Function as a Service

Function as a Service (FaaS) is considered the most natural way to use serverless computing, and it focuses on providing small pieces of code represented as functions that are executed in the cloud. These functions are expected to only run for a short amount of time (at most minutes) and the execution of the function is initiated by some incoming FaaS trigger, such as HTTP request, storage upload, and database insertion/deletion/update. In addition, the functions are not allowed to keep a persistent state, which in combination with the execution time constraint facilitates maintainability and scalability by service providers [2].

The two major primitives of FaaS programming model are *Action* and *Trigger*. Action is a stateless function that executes the code deployed to the cloud. The action could either be invoked synchronously or asynchronously, where the former's invoker function (request) expects a response as a result of the action executed, and the invoker function of the latter does not. The actions can be invoked by REST API, or executed based on a trigger, which is a class of events from various sources, e.g. upload to storage, and insertion to a database. Further, a parallel invocation is when an event triggers multiple functions at the same time, and a sequential invocation is when a result of action triggers another function. The biggest cloud providers tend to provide the ability to mix and match different and multiple services to create complex serverless applications. Figure 2.2 shows an example of how an Azure application could be constructed by combining different services and Azure Functions to create a serverless application for a car toll booth.
The process of triggering begins with a trigger event being registered, which causes the cloud provider to automatically allocate the required amount of computing resources to perform the workload, and the process ends with executing the function. The allocated computer resources are then ready for further executions, which is referred to as the function is warm. However, if a function has not been used for a while the latency could increase substantially during the first invocation due to the time it takes to allocate capacity and for the function runtime to start up on the allocated server. This phenomenon is called cold start [12]. Figure 2.3 illustrates the difference when a function is executed with a warm or a cold start. As can be seen, the cold start has multiple tasks before it can execute the code, while the warm start executes the code directly. The duration of how long a function stays warm varies between providers, for Microsoft Azure, this deallocation of resources happens after roughly 20 minutes of inactivity [12].

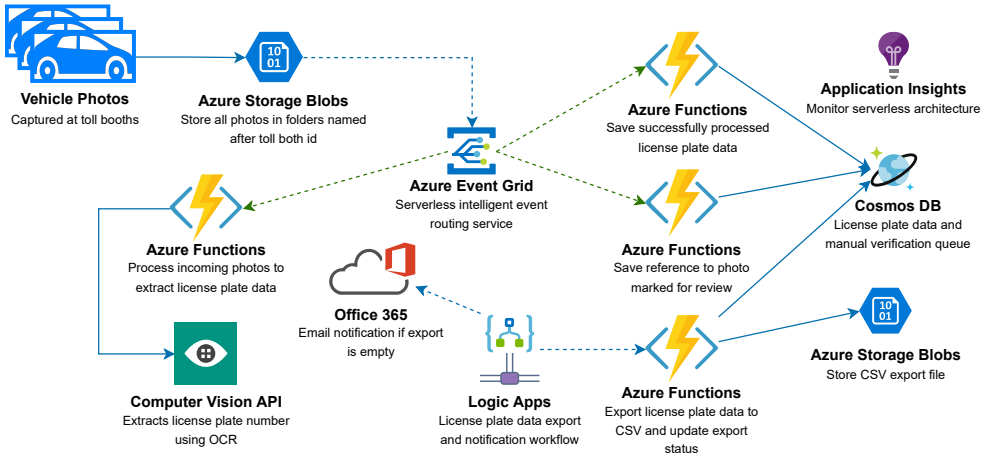


Figure 2.2: An example of how an Azure Application could be applicable to a toll booth [11].

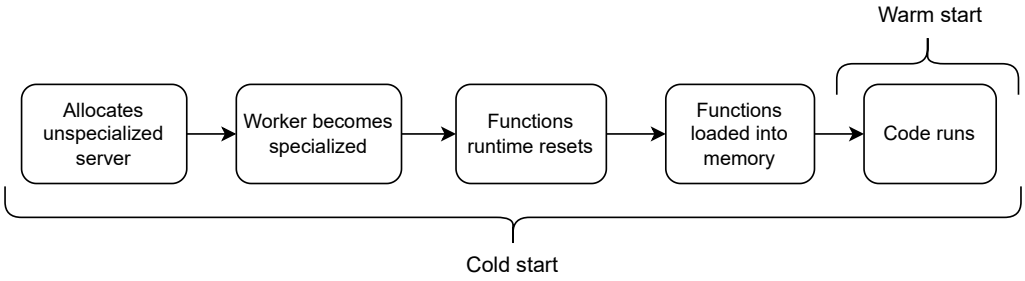


Figure 2.3: A visualization of what happens during a start of a trigger invocation, with focus on the differences between a cold and a warm start.

Cloud providers provide various triggers for different purposes and can sometimes be used interchangeably with trade-offs, which depend on e.g. application context and other constraints. The Azure triggers studied during this thesis are HTTP, Blob Storage, Queue Storage, Cosmos DB, Service Bus Topic, Event Hub, and Event Grid. This selection of triggers is based on popularity [1] and to cover a wide variety of Azure triggers. Triggers that were excluded from investigation were third-party triggers, which have other requirements, and timer trigger which is typically not used for latency-sensitive applications.

2.4 Cloud Services

In Section 2.3, services were mentioned as building blocks for creating serverless applications, and Figure 2.2 only shows a small fraction of the services that Azure offers. The available services cover various areas such as AI, machine learning, networking, management, storage, security, etc, and the offerings are similar between cloud providers but the underlying implementations of the services might be different. The following is a non-exhaustive list with basic descriptions of the services that are used in this project:

- **Azure Functions:** A serverless solution that provides "compute on-demand" and scales according to demand, which is the essence of serverless computing. This allows developers to focus on the logic and coding without worrying about the maintenance of underlying infrastructure.[1]
- **Azure Storage:** Microsoft's cloud storage solution for modern data storage. It offers scalable, highly available, durable, and secure storage in the cloud for various data objects.[2]
  - **Blob Storage:** Storage optimized for storing large amounts of unstructured data.[3]
  - **Queue Storage:** A messaging store for storing tons of messages and for reliable messaging between components.[4]
- **Cosmos DB:** Fully managed SQL database which enables response times in milliseconds and automatic and instant scalability. No database management nor the administration is required by the developer.[5]
- **Event Grid:** Allows for efficient development of event-based architecture applications, through subscriptions to Azure resources where events are retrieved and then sent to event handlers, e.g. Azure Functions or Service Bus, for further processing.[6]
- **Event Hub:** A big data streaming platform and event ingestion service that is capable of retrieving and processing millions of events per second.[7]
- **API Management:** A service that provides an API gateway, which enables communication between Azure's services and the client applications by forwarding requests from the clients to respective backend services.[8]
- **Service Bus:** Enterprise message broker with messaging queues and publish-subscribe topics.[9]
- **Azure Monitor (Application Insights):** A service that enables collection, analysis, diagnosis, and acting on telemetry from the cloud to give clients insights and assist in assessing the availability and performance of clients applications and services.[10]

2.5 Infrastructure as Code

Infrastructure as Code (IaC) is essential in the context of cloud resource provisioning and deployment because it enables organizations to automate the provisioning of infrastructure using *code* instead of e.g. the cloud providers' portals and CLI, and

[1] https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview
[2] https://docs.microsoft.com/en-us/azure/storage/common/storage-introduction
[3] https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction
[4] https://docs.microsoft.com/en-us/azure/storage/queues/storage-queues-introduction
[5] https://docs.microsoft.com/en-us/azure/cosmos-db/introduction
[6] https://docs.microsoft.com/en-us/azure/event-grid/overview
[7] https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-about
[8] https://docs.microsoft.com/en-us/azure/api-management/api-management-key-concepts
[9] https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview
[10] https://docs.microsoft.com/en-us/azure/azure-monitor/overview

allows for fast development, deployment, and scaling of cloud applications with reduced risks and costs [13]. In an enterprise setting, it is not uncommon to deploy applications to production many times each day, where the infrastructure is constantly being deployed, destroyed, and scaled up and down. It is critical for an automated provisioning and deployment solution to control cost, reduce risks, and quickly respond to market opportunities and threats [13]. IaC utilizes a high-level descriptive coding language to automate the provisioning of cloud resources, which removes the tedious tasks of manually provisioning and managing cloud resources such as storage, databases, and servers, for the developer during the development, testing, and deployment of an application. There does not exist a standard syntax for declarative IaC, and different providers support different file formats, e.g. XML, JSON, and YAML [14].

To implement IaC in this thesis, Pulumi is used. Pulumi is an open-source tool that mitigates the drawback of provider-specific configuration languages and enables developers to create, deploy, and manage cloud resources using popular programming languages such as Node.js, Python, Go, and .NET Core. This further allows developers to utilize engineering practices, integrate infrastructure with CI/CD workflows, automate deployments with code at runtime, and complement Pulumi with other preferred tools and libraries to facilitate development while also reducing boilerplate code [15].

2.6 Distributed Tracing

Distributed tracing is a method of observing requests, dependencies, and traces, which traverse across numerous distributed inter-dependent components in cloud environments. It is a critical component of observability and allows for pinpointing where failures occur and causes of poor performance [16]. In a traditional monolithic setting within a single application, this would correspond to logging, and tracking the application end-to-end would be fairly straightforward, which is not the case in a decentralized microservice-based distributed system such as Azure. Performing tracing in cloud environments is essential to instrument the code at specific points of interest, where tracing data is produced and later aggregated to generate a complete end-to-end trace. This is, however, a great challenge due to the complexity and nature of distributed systems [17]. In a serverless setting, where the underlying infrastructure is abstracted away from the user, tracing is even more complicated, and consumers have to rely on tools provided by the cloud providers for monitoring and collecting tracing data.

Microsoft Azure Application Insights is the service that is provided by Azure for distributed system tracing. It is part of Azure Monitor, which is a collection of multiple services that help consumers with collecting, analyzing, and acting on telemetry from the cloud. The Azure Insights service can be managed through Azure's Web portal, which e.g. enables visualization of traces. Figure 2.4 shows an end-to-end transaction of traces containing information, data, and statistics, among

many other features of traces. It also provides a software development toolkit[11] (SDK) to enable consumers to programmatically interact with the service for reasons such as instrumentation and configuration. The service is created as a shared resource between all of the other components and is used to group traces. However, the results from most of the triggers are disconnected traces, for which there is no reliable way provided by Azure to connect/correlate them, except for the HTTP trigger. A Python script is implemented to manually correlate the traces, more details in Chapter 4.

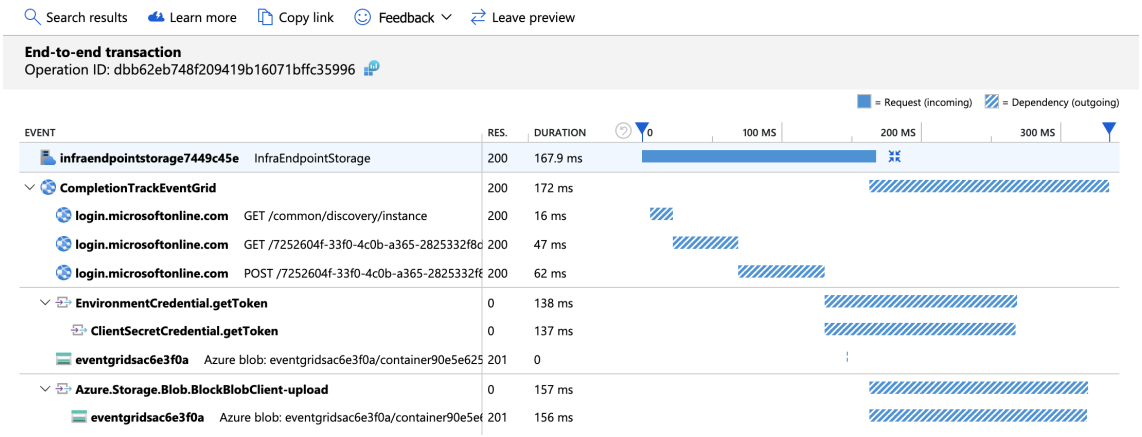


Figure 2.4: An example of end-to-end trace visualization in Azure's Web Portal.

2.7 Performance Benchmarking

Benchmarking is a process of using benchmarks to collect measurements for assessment and comparisons to conclude the best option for a given scenario concerning certain objectives [18]. For example, the SPECCpu benchmark is used to compare the performance of CPUs to find out which CPU is the best performing in terms of the run time of several programs [19]. Following is a definition of benchmark for a more precise description:

> *"Standard tool for the competitive evaluation and comparison of competing systems or components according to specific characteristics, such as performance, dependability, or security" [20]*

Some desirable characteristics when building a benchmark are discussed by Kisowski et al. [20] and are critical to consider when designing and developing a benchmark:

- **Relevance** and usefulness of a benchmark in a specific setting to different consumers. It is important for the consumer of benchmark results that the assessment of a benchmark's relevance is based on the context for which the benchmark was originally planned. From the perspective of the benchmark

[11] https://www.nuget.org/packages/Microsoft.ApplicationInsights/2.21.0-beta1

designer, relevance involves determining the intended use of a benchmark and developing the benchmark with consideration to the identified intended use cases. Generally, relevance is assessed by the breadth of applicability, and if the workload is relevant for the specific scenario. Achieving scalability, which is an important aspect of relevance, is a great challenge since developing benchmarks that are expected to run on a wide variety of different systems is difficult.

- **Reproducibility** is the capability of a benchmark to consistently produce the same results each time the benchmark is executed for a particular environment. An ideal reproducible benchmark would enable users to produce the same results when executing the benchmark independent of the environment the benchmark is executed in. However, this is typically not the case due to the inherent challenges of modern computer systems which further introduce variability in performance measurements. Factors that affect the variability include thread scheduling, dynamic compilation, physical disk layout, and network connection.
- **Fairness** ensures that systems can be compared reasonably and compete on their merits without artificial constraints. Fairness has to be considered in different parts of a benchmark. An example is during benchmark development, where benchmarks developed by a consensus of experts are generally perceived fairer than by a single company. Another example is the requirements of hardware and software to run a benchmark. It is often necessary to put constraints on what components are allowed to be used since different configurations of hardware and software might impact the results.
- **Verifiability** is essential because the results should be verifiable by practitioners to deem the trustworthiness of the results produced by the benchmark. Good benchmarks typically perform self-validation to ensure that the workload is running as expected, and also functional tests to verify whether the output of the benchmark is correct. One way of improving verifiability is to include more details than necessary when outputting the results from the benchmark. These details could potentially raise questions in case of inconsistencies.

In the case of FaaS Performance Benchmarking, two benchmark types are usually considered, namely, micro-benchmarks and application benchmarks [21]. The differentiation between the two types lies in the objective, where micro-benchmarks target specific performance aspects with artificial workloads while application-benchmarks target the overall performance of real-world application workloads.

2.8 Load Testing

The reliability part of this thesis requires simulation of workloads to determine the behavior of the Azure Function Triggers under different levels of load, the amount, and the pattern of requests. Load testing is a testing method to evaluate an application under certain load conditions by sending requests and measuring the responses [22]. These load conditions are typically specified by developers and reflect realistic application scenarios. The main benefit of this testing method is the identification of the maximum operating capacity and cause of performance degradation e.g. mem-

ory leaks, and thread contentions, of an application, to improve performance and facilitate mitigation processes.

K6 is an open-source load testing tool that will be used for load testing the benchmark [23]. It allows developers to programmatically define the load tests using the CLI tool with developer-friendly APIs, and scripting using JavaScript. There are various configurations for defining a load test and K6 provides a wide variety of options, e.g. number of virtual users (VU) to run concurrently, tags, scenarios, iterations, etc.

3
Related Work

In this chapter, other studies that relate to the proposed work of this thesis are presented to understand the contribution of the thesis given the state of existing research and provide a further context of the domain.

3.1 Latency of Serverless Function Invocation

There is a common belief that serverless is not suitable for latency-critical systems, however, Eismann et al. [24] found that this belief is not true where a large percent of serverless applications experience high traffic intensity. With this in mind, one of the purposes of this thesis is to empathize and evaluate the latency within serverless to, for example, ease latency-sensitive applications within serverless. There are already other studies that have researched latency in the domain of serverless, but not with the same focus. However, it is important to evaluate those studies to learn and be aware of previous research. Table 3.1 by Spoltis et al. [25] shows services within various industries where low-latency is critical. Not all of the services mentioned in the table can be adopted to serverless functions, however, those that are, the results of this thesis will help in deciding what function trigger types to use to reduce the latency when using Azure Functions.

In a recent study by Bertilsson and Grönqvist [26] the performance, specifically the latency, of different triggers between Azure Functions and Lambda Functions was being analyzed. The purpose of the study is to increase the understanding of different FaaS service triggers and their benefits and drawbacks with respect to performance. Even though their study was pursuing a performance comparison between providers, the research methodology has a lot of similarities to this thesis, as also how the Azure measurements proceed. Except for the latency result from Azure triggers, there are some other valuable insights in their study that could be beneficial for this thesis, for example, how to design the benchmark and ensure that it is comparable between different triggers.

Industry	Applications and services
Education	• Video conferencing • Live-streaming • Rich learning content • Dynamic e-learning platforms • Presentation applications • Dynamic administration tools • Cloud-based applications
Healthcare	• Picture Archiving Communications Systems (PACS) • Telemedicine, telehealth applications • Diagnostic imaging • Electronic Medical Records (EMR) • Patient portals • Mobile healthcare applications and equipment
Media and Entertainment	• Live-streaming breaking news • Television shows • Videoconferencing • Movies over Internet • Transfer large files, images, and videos • Real-time gaming
Finance	• High-Frequency Trading (HFT) and high speed information exchange • Financial transactions • Connections to brokers, dealers, exchanges and hedge funds

Table 3.1: Latency sensitive services [25].

Pelle et al. [27] is another study that evaluates performance in serverless towards latency-sensitive applications. They also tried to adjust a drone control application and investigate the performance. This thesis intends to do a similar analysis, but for Microsoft Azure, however, their study focused more on pure latency within AWS and not specific trigger types as this thesis intends to do. The key parts that can be used from the study of Pelle et al. [27] are however research methods and discussion about latency, which is highly interesting.

3.2 Benchmark in Serverless Computing

Serverless is rapidly evolving, and the popularity of adopting Serverless is rising each year [28]. It is important that the development of experimental methodology advances concurrently with the evolving cloud domain to address new methodological challenges related to cloud computing and Serverless such as dynamic environments and on-demand resources and services [29], [30]. However, the lack of standardized performance benchmarking suites in the cloud complicates meta-analysis and comparison of research solutions [29]. Papadopoulos et al. [30] combine best practices from similar fields to propose eight methodological principles that could be adopted by the cloud community to improve and standardize the way performance evaluation is conducted. Further, their survey study showed that most of the proposed

principles were not or only partially addressed in the papers that were analyzed, which signifies the importance of adopting standardized performance evaluation in the field. The following are the eight reproducibility principles by Papadopuslos et al.:

P1: *Repeated experiments (statistical). Decide how many repetitions with the same configuration of the experiment should be run, and then quantify the confidence in the final result.*
P2: *Workload and configuration coverage. Should cover a representative sample space.*
P3: *Experimental setup description. Hardware and software setup should be described and the objective should be stated for each experiment.*
P4: *Open access artifact. At least a representative subset of the results should be made publicly available.*
P5: *Probabilistic result description of measured performance. Report a characterization of the empirical distribution of the measured performance.*
P6: *Statistical evaluation. Provide a statistical evaluation of the significance of the obtained results.*
P7: *Measurement units. For all the reported quantities, report the corresponding unit of measurement.*
P8: *Cost. The cost of running the experiment should be included.*

Copik et al. [29], in line with Papadopoulos et al. [30], emphasize the importance of standardized benchmarking suites in the cloud domain and propose the Serverless Benchmark Suite (SeBS), the first systematic FaaS computing benchmark for serverless computing. The benchmark suite measures metrics such as execution time, CPU utilization, memory, and I/O across different cloud providers. However, this thesis has a lower abstraction level and will focus on the performance of different trigger types from a single cloud provider.

3.3 Reliability of Event Deliveries

To the authors' knowledge and based on searches on IEEExplore and Google Scholar, not much research and papers have been devoted to the reliability of event deliveries in the domain of serverless. Documentation of Azure Triggers, e.g. Event grid [6], states whether the trigger type guarantees the order of event delivery, and some explanations and solutions of Azure Function event processing are addressed [31]. However, even if it is stated that there is no guarantee for order of event delivery for some trigger types, the frequency is not mentioned, which is highly relevant for applications that might tolerate unreliability to some degree. This unknown degree is a part of what this thesis intends to observe. A similar phenomenon of OoO, missed and multiple deliveries are addressed in the domain of Event Stream Processing, and the terms used by Finta et al. [32] are OoO arrival, data loss, and duplicate deliveries. Li et al. [33] state that if the order of events received by an event stream processing system is the same as the recorded timestamp order, it satisfies the total order assumption. Further, it is mentioned that it is common for event sequences to arrive OoO in distributed computing environments due to network traffic and node

failure. Even though the solution and context of the paper are different from this study, it provides information, inspiration, and ideas about event deliveries which is highly relevant.

3.4 Impact of Different Language Runtimes

Serverless providers often offer different types of runtimes for their triggers to satisfy customers. However, due to different language optimizations, the choice of the runtime can have an impact on the performance and cost. Jackson and Clynch [34] is one of the multiple studies that evaluate the impact of runtime. They chose to investigate the aspect of cold-starts for both Lambda Functions and Azure Functions. In their conclusion, they found that there were differentials on both platforms, where some runtimes performed significantly better compared to the rest. Jackson and Clynch concluded that C# .NET was the best performer and most economical option for Azure Functions and that NodeJS should be carefully used to avoid potential slow and costly start-up times.
The experimental setup that Jackson and Clynch used is different from the one of Bertilsson and Grönqvist [26], which this thesis will be building upon. The main difference is that Jackson and Clynch investigate cold-starts which require a systematic approach to produce. As mentioned in 2.3, cold starts occur due to either inactivity or newly deployed Azure Functions. Therefore, the setup from Jackson and Clynch uses timer services to invoke the functions at specific times to produce cold-starts. Another difference is the cost aspect of Jackson and Clynch study which required additional components to handle calculations. Both studies by Jackson and Clynch, and Bertilsson and Grönqvist considered two cloud providers (AWS and Azure). However, in this thesis the focus is only on one provider and therefore the setup will diverge from the initial setup by Bertilsson and Grönqvist.
The difference between Jackson and Clynch and this thesis is the research areas within serverless. While Jackson and Clynch investigate the impact of language runtimes on cold-starts, this study aims to look at how language runtimes affect the latency of invoking Azure Functions with different trigger types.

4
Research Method

This chapter introduces the methodology adopted in this thesis to analyze and evaluate the the performance of Azure Function Triggers based on latency and reliability of event delivery.

4.1 Method Overview

The empirical research method of benchmarking is adopted in this thesis to create a relevant benchmark for performance assessment of a software system. This empirical research method has essential attributes that are addressed throughout the thesis report, which are the following [35]:

- The quality to be benchmarked.
- The metrics to quantify the quality.
- The measurement method(s) for the metric.
- The workload and justification of the design.
- Describe the experimental setup for the benchmark in sufficient detail to support independent replication.
- Specify the workload in sufficient detail to support independent replication.
- Allows different configurations of a system under test to compete on their merits without artificial limitations.
- Assesses stability or reliability using sufficient experiment repetitions and execution duration.
- Discusses the construct validity of the benchmark.

The overall research process, which is illustrated in Figure 4.1, includes the Benchmark Design, Benchmark Execution, and Data Analysis. The first step was benchmark design which included several aspects such as the design of the benchmark application and realistic workloads. This is explained in more depth in Section 4.2. The next step is Benchmark execution which is where the benchmark application developed in the previous step is utilized to generate and collect tracing data. The data will in the last step be used for analysis to answer the research questions.

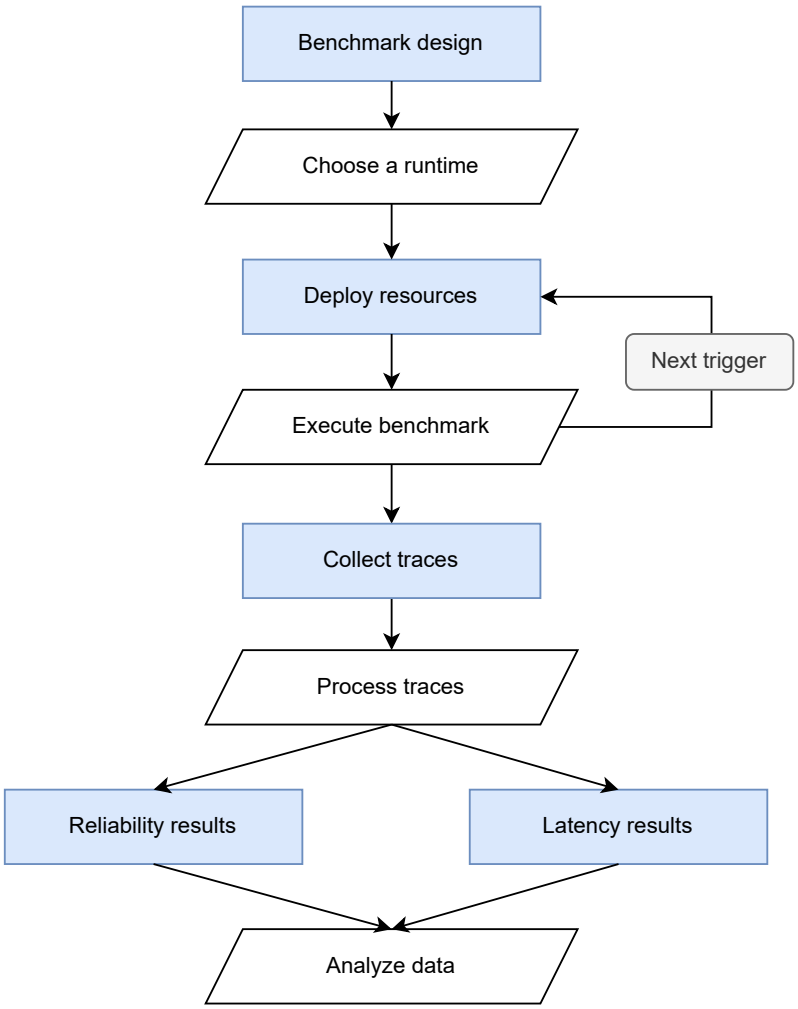


Figure 4.1: An overview of the flow of the benchmark.

4.1.1 Latency and Reliability Experiment

Regarding the analysis of trigger latency, this thesis builds upon a completed master thesis by Bertilsson and Grönqvist [26]. The differentiation is that this thesis intends to compare triggers and not cloud providers. Thus, this thesis focuses only on one cloud provider, Microsoft Azure, and provides performance analyses on additional triggers. Further, this thesis also investigates the impact of different runtimes on trigger types. The additional dimension of investigating the impact of runtimes on latency required redesigning parts of the benchmark that involved the deployment of resources. Figure 4.1 shows the flow of how this part of the research was conducted. First, understand the underlying benchmark design and frameworks e.g. Azure and Pulumi, and then implement additional triggers. This resulted in a set of trigger benchmarks which were executed to get relevant data for processing and analysis.

Research on the reliability of event delivery for Azure triggers was based on the data collected from the latency experiment. The extensions are analysis scripts to extract and aggregate relevant data to perform reliability analysis (order of delivery, missed,

and duplicate).

4.2 Benchmark Design

The initial structure of the benchmark was based on Bertilsson and Grönqvist's [26] benchmark design because of their experimental synergies with this research paper. The main differences are that Bertilsson and Grönqvist solely researched the latency of three different triggers on two cloud providers where one is Microsoft Azure, and did not investigate the impact of runtimes on latency. Even though the scope of the experiments is not the same, the benchmark design that Bertilsson and Grönqvist developed was still transferable to this research use case, with some modifications to accommodate the additional dimension of runtimes. The advantage of using the benchmark was that the limited time could be used to improve the benchmark and focus on other critical aspects.

A major benefit of using the initial benchmark was that both the problems of how to trace the invocation together with the receiver and how to run the benchmark unlimited times in a certain time frame already were solved. Resources could instead be allocated to automate the benchmark, which was only semi-automated in the beginning. The automatization was important in order to ensure better reproducibility and quicker deployment of triggers. The benchmark was expanded to the extent that it was completely automated, which is something that is not only positive for the reproducibility, but also for running a complete experiment containing all triggers without human interaction. The restructuring and expansion of the initial benchmark made it more solid and stable in terms of errors during benchmark execution.

One of the outcomes of this research paper was to compare reliability among the different triggers. In order to use the same benchmark for both latency and reliability, some additional attributes to the initial benchmark were added - mainly affecting how the distributed tracing was designed. There was also a need of writing new scripts that could analyze the correct traces and calculate the different aspects of the reliability namely OoO, missing, and duplicate executions.

4.2.1 Application Components

The high-level application design consists of three different main components. The first component is the shared resources, which can be seen in Figure 4.2. Its purpose is to create and hold all of the shared resources that the other two independent components need to function correctly. The shared resources also assigns necessary authorization roles to run the benchmark without interruptions.

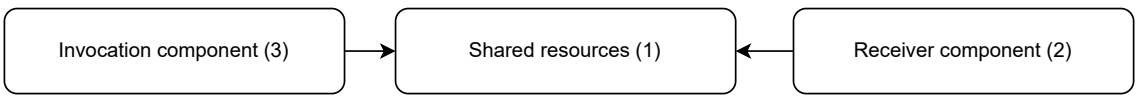


Figure 4.2: An overview of the high-level architecture, which consist of three components: shared resources, invocation component and receiver component.

The second component in the high-level architecture in Figure 4.2 is the receiver component which is responsible for deploying all trigger-specific resources that are needed. It is also responsible for publishing the crucial event trigger binding that is going to be triggered by the invocation component.

Lastly, the third component in Figure 4.2 is the invocation component, which acts as an entry point to create the correct circumstances for the trigger to execute. Depending on which trigger is targeted, a unique endpoint is generated to begin the invocation. This component is not dependent on the receiver component.

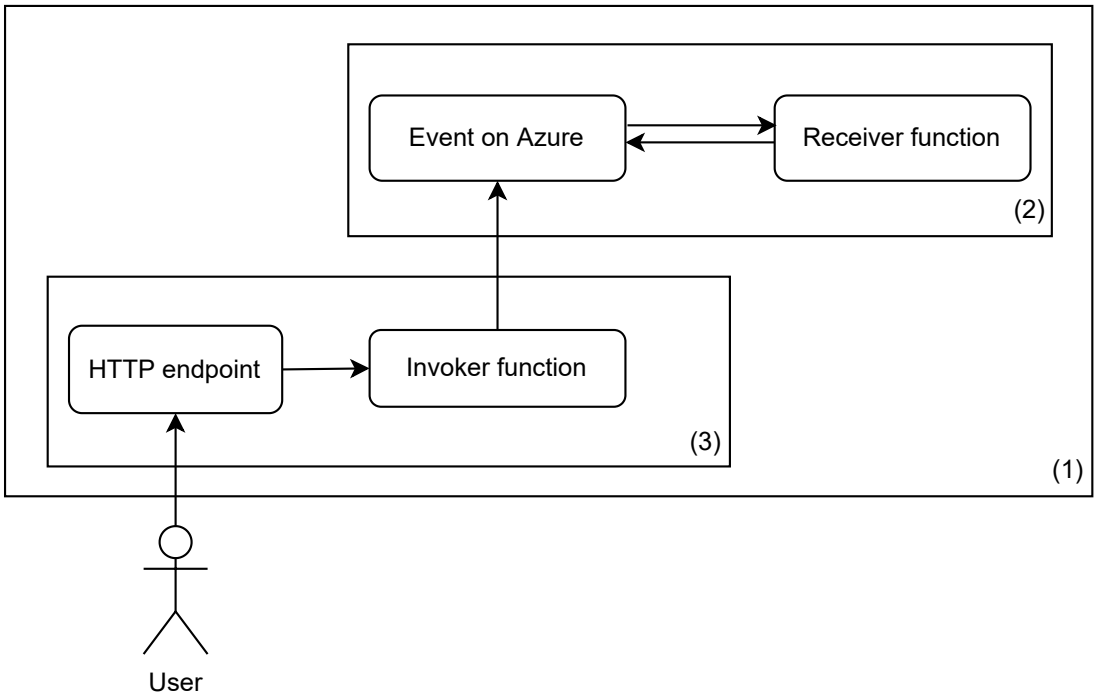


Figure 4.3: An overview of the high-level architecture of the application design.

The complete application, as shown in Figure 4.3 is ready when the three different components (1), (2) and (3) in Figure 4.2 is deployed. The receiver component (2) in Figure 4.2 then has deployed the necessary resources to enable specific events to be sent to the portal, by using the shared resources component (1). The trigger receiver also has a receiver function that is triggered whenever specific events are created. The receiver function is responsible to register whenever it is being triggered through distributed tracing.

Component (3) represents the invocation component that has an HTTP endpoint, which acts as the entry point for the application. The purpose of the endpoint is to invoke the system by sending an event to the Azure Portal e.g. uploading a data file or a message. The endpoint is traced whenever the event is being initialized through distributed tracing.

4.2.2 Deployment Design

The IaC framework Pulumi was used to automatically deploy resources to the cloud. The framework does not only allow the resources to be directly deployed on the Azure Portal for a specific trigger, but it also removes old resources that are no longer needed. Pulumi also takes care of the different authorizations that are needed in order to get control of resources in the Azure Portal. Therefore, Pulumi is crucial for the experiment to work since whenever the execution of one trigger is finished Pulumi will destroy its resources and deploy new trigger-specific resources for the next trigger. A result of this approach is that only one trigger is built and tested at a time.

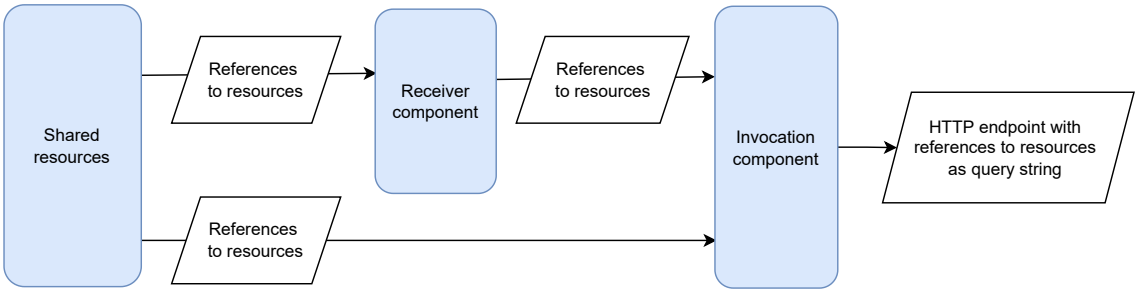


Figure 4.4: An overview of the high-level component deployment.

In Figure 4.4 the overview of the deployment architecture is viewed. The shared resources is the first component to deploy and is responsible of the following parts:

- Adds an Application in App Registrations.
- Creates the Application Insights that later will hold all traces.
- Managing roles and policies for the Application to access necessary permissions.
- Creates the shared Azure Function App where trigger-specific code and bindings will be published to.
- Outputs references, through Pulumi, to resources.

The shared component deploys the Function App that will be shared between all of the triggers for a specific runtime. The Function App will need to be re-deployed with updated configurations to test triggers with another runtime. Further, the runtime of a Function App has to match the runtime of the trigger published to the Function App. Therefore, when testing the trigger with another runtime, the shared resources have to be re-deployed. Publishing a trigger to a Function App is done with a one-line command through Azure CLI:

```
func azure functionapp publish <FUNCTIONAPP_NAME> --<RUNTIME> --force
```

After the shared resources are successfully deployed, the next component to be deployed is the receiver component which both uses data from the Pulumi output references and the local environment file:

- Creates the resources for the specific trigger.

- Publishes the trigger-specific code to the Azure Function App.
- Outputs references, through Pulumi, to trigger resources.

The last component to deploy is the invocation component which uses references from shared resources and references from the receiver component. The component is responsible to combine earlier deployments and deploying other resources responsible for invoking the triggers. It includes the following:

- Creates a Function App containing an HTTP endpoint that acts as the entry-point of the experiment.
- Depending on the selected trigger, the HTTP endpoint will execute different functions.
- Outputs the complete URL with trigger-specific parameters as query string.

The deployment is successful after the invocation component is fully deployed. The generated URL is the entry point to trigger the invoker function, see Figure 4.3.

4.2.3 Trace Design

Microsoft Azure Application Insights is the service that is provided by Azure for system tracing. The service is created as a shared resource between all of the other components and is used to group traces. The aim of measuring trigger latency and reliability is enabled by distributed tracing which provides end-to-end visibility.

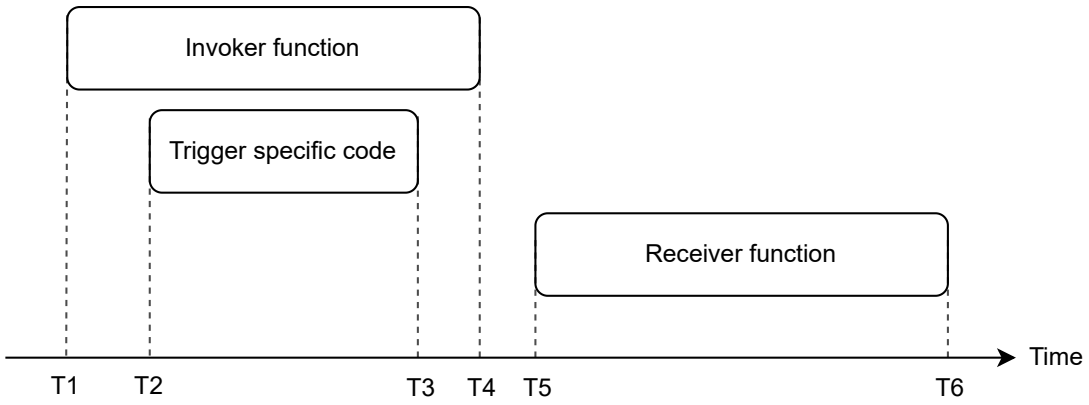


Figure 4.5: An overview of timestamps during an invocation and receiver of an asynchronous trigger.

The approach of collecting traces considers the developers' perspective. Developers' perspective means that it is more interesting to investigate the latency between when a trigger is executed and when the developers regain control over what code should be executed next, e.g. the first line of a file that a developer has written. The timestamps that would represent the latency would then be from T2 to T5 in Figure 4.2.3, which include other factors such as network connectivity and cold starts. An alternative measurement approach would be the latency between T3 and T5.

However, this approach requires timestamps whenever the trigger-specific code has been executed, which is not provided by Azure and makes this approach unsuitable because T3 could finish after T5 resulting in negative values.

Both T1 and T5 in Figure 4.5 are registered in the Application Insights automatically, corresponding to when the invoker and receiver function is being invoked. However, in order to get the correct timestamps, the timestamp T2 in Figure 4.5 needs to get registered. Since T2 is not a timestamp that is automatically monitored in Application Insights, it has to be manually added. In order to connect the custom T2 trace with the T5, another support trace needs to be sent under the receiver Function, T5, and T6. The purpose of the support trace is for the data analysis to receive the T1 operation id, which is an id used for tracing, and connect it with the T5 operation id.

In order to measure duplicate invokes when running an experiment, a custom id is also sent as a part of the query string which corresponds to the iteration of the invoke. This id is sent as a trace to Application Insights and used for analysis.

The following list describes how the outcome of an experiment is calculated:

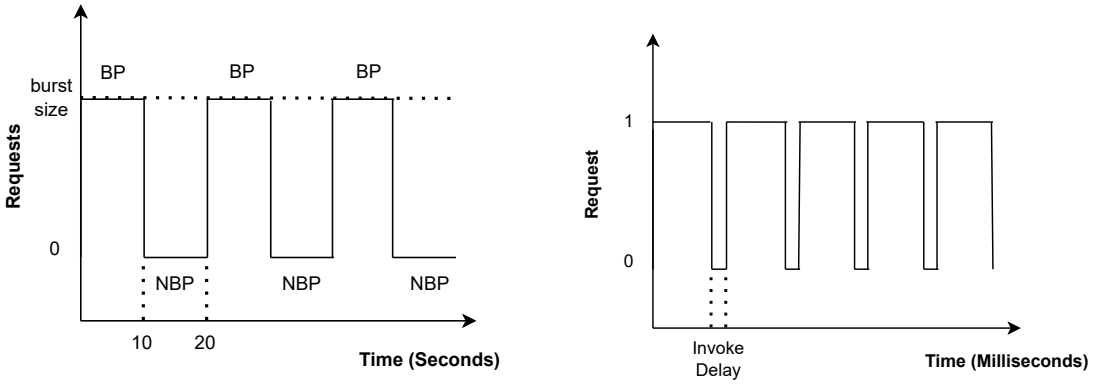
- The latency is calculated by the difference in timestamps between T2 and T5.
- An OoO execution is calculated by in which order T2 arrived compared to T5 e.g. if other traces appear in between there is an OoO execution.
- A missing execution is found if a T2 trace has occurred, but not the corresponding T5 trace.
- A duplicate execution is found if the invocation id has occurred more than once.

4.2.4 Workload Design

In order to receive traces for analysis, it is required to invoke the benchmark by sending requests to start executing the serverless functions. A workload or usage profile needs to be specified to let K6 generate and simulate the invocation pattern that is of interest. This is important for reproducibility and is one of the essential attributes of the empirical research method of benchmarking to specify. The worst cases of latency are due to cold starts, and since cold and warm invocations are not the focus of this thesis, the benchmark will detect the cold starts, filter them out, and only focus on warm invocations. According to Ustiugov et al. [36], one of the largest contributors to latency variability in modern serverless systems is short bursts of function invocations, therefore a bursty workload will be designed and used to study the latency, as well as missing and duplicate event deliveries. Figure 4.6a illustrates the bursty workload design that involves the burst phase (BP), and the non-burst phase (NBP). The BPs will reach a selected burst size of 1, 10, 50, 100, or 300 invocations, and the duration of the NBPs are 10 seconds in between the BPs.

A simultaneous burst of requests typically does not have an inherent order semantic.

Therefore, an additional workload design with thoroughly controlled inter-arrival times (IAT) was necessary to study the order of event deliveries. Controlled IAT is basically that the delay in-between invocations are manually configured, which allows the creation of the desired invocation pattern. The load generation tool K6 allowed us to create this workload with fine granularity of milliseconds for the start time for each request. Figure 4.6b depicts the workload pattern needed to test for order of delivery for each of the trigger types. As can be observed, only one request is sent per spike and the requests are sent with an *invoke delay*, which allows for a thoroughly controlled invocation pattern. In this design, the values of the invoke delay studied are 1ms, 10ms, 25ms, 50ms, 100ms, 150ms, and 250ms. The short invocation intervals will use multiple virtual users in order to keep the frequency, while the long intervals only will handle the frequency with a single virtual user. Since an order of invocations is implemented, it is possible to compare the order of when the invoker and receiver functions are triggered. An OoO event delivery is identified if the orders are not the same.



(a) Bursty Workload with parameterized burst size.

(b) Constant IAT controlled workload with parameterized invoke delay.

Figure 4.6: Two figures showing the workload designs, constant IAT and burst, that produce the desired traces.

The design of the workloads is not based on data of real workload patterns. Therefore, the findings from this thesis might not be directly applicable to specific workload patterns in the industry. However, as mentioned before a characteristic of patterns that occur is short bursts of invocations, which have been found to cause tail latency and therefore valid for evaluating the latency. Further, the IAT controlled workload is neither based on real workload patterns, and is primarily implemented to study sequential invocations with short delays.

4.2.5 Trigger Types

In this section, details about the characteristics, configuration, and implementation of each studied trigger type will be discussed.

4.2.5.1 Blob Storage

Azure Blob storage is one of Microsoft's storage solutions for modern data storage and is optimized and designed for multiple purposes such as writing to log files, storing files for distributed access, and serving images or documents directly to a browser. With the integration between Blob storage and Azure Functions through the Blob storage trigger, it allows the implementation of functions that is capable to react to modifications of the Blob storage. The input binding of the trigger is to read data within the blob that triggered the trigger. The Blob storage trigger can then, as the output binding, write to the Blob storage data [37].

However, it is important to note that the Blob storage trigger is poll-based and not event-based. Poll-based triggers are events that *periodically* make calls to services to scan for new data, whereas event-based triggers will get triggered when specific events occur. According to the documentation, the storage logs are created on a "best-effort" basis, which means that there is no guarantee that all events are captured. Therefore, logs may be missed. Further, the polling mechanism for Blob storage is a hybrid between inspecting created logs and periodically scanning containers for changes. The scanning of blobs is done in groups of 10,000 at a time, using a continuation token between intervals to know where to start the next scan [37].

In the benchmark, the Blob storage trigger is tested by deploying a storage account together with a data container. A function app is then created with a function that subscribes on events from the data container in the storage account e.g. if data is created, deleted, or edited. To run the storage trigger the storage and container name is sent as a query string to the invocation HTTP endpoint. The invocation firstly initiates Application Insight to generate an operation id of the invocation, then connects to the container and inserts a new file of dummy data with the operation id as metadata. The listening trigger in the function app fires as soon as it recognizes the insertion of a data file and sends a trace containing the invoker's and its own operation id from the metadata.

4.2.5.2 Cosmos DB

Cosmos DB is Azure's fully managed NoSQL database service for modern app development. Cosmos DB guarantees single-digit millisecond response times and automatic and instant scalability. A Cosmos DB trigger listens for changes in a cosmos database i.e. both inserts or updates but does not include deletions. Input binding of the trigger is the changed or added document in the database, while the output binding can save changes into the document [38]. Cosmos DB is poll-based with the possibility to be configured through different settings i.e. feed poll delay and max items per invocation. However, a lower polling interval increases of events in Azure which in turn can lead to higher costs. The following settings have been used in the benchmark (that differ from the default values):

```
"maxItemsPerInvocation": 1,
"checkpointDocumentCount": 1,
```

```
"feedPollDelay": 10
```

In the benchmark, the Cosmos DB trigger is to deploy the required resources, namely a cosmos DB account, a sqlDatabase, and a sqlContainer. The Cosmos DB account is deployed with an attached Azure Function that triggers on the *onChange* callback whenever there is a change, e.g. insertions, deletions, and updates, within the account resource. To trigger the Cosmos DB trigger, an item is inserted into the container (also called collection in Azure Portal).

4.2.5.3 Event Hub

An Event Hub is an event-based trigger that is triggered when an event is sent to an Event Hub event stream. The trigger receives the data from the stream as an input and can write events to the stream as the output binding. There are several settings available to change batch sizes and delays within the trigger [39]. However, a lower max batch size can lead to an increase in the execution of the trigger which in turn can lead to higher costs. The following settings have been used in the benchmark (that differ from the default values):

```
"maxEventBatchSize": 1,
```

In the benchmark, the Event Hub trigger is created by deploying an Event Hub inside of an Event Hub Namespace. A function app is then created that subscribes to batches that are created in the Event Hub. To run the Event Hub trigger both the Event Hub and Event Hub Namespace names are sent as a query string to the invocation HTTP endpoint. The endpoint firstly initiates Application Insights to generate an operation id of the invocation. The endpoint then connects to the event hub producer client and creates a new batch that only contains the operation id. The trigger in the function app fires as soon as it recognizes the new batch in the event hub, which extracts the operation id and sends a trace to Application Insights.

4.2.5.4 Event Grid

The Event Grid can be used for multiple purposes within the whole Azure resource since there are different publishers to subscribe to. A publisher is a resource or the service that is the source of an event. In this benchmark, the Event Grid has an Azure blob storage account as the publisher, which means that the Event Grid trigger listens to Azure blob storage events i.e. blob uploads or deletions. The Event Grid trigger receives an event grid event as input binding, in this case, a Storage Blob, and returns an Event Grid event as the output binding [40]. Even though the event grid trigger has its implementation, the functionality is the same as a Storage blob trigger - except that the Event Grid also reacts to deletions of blobs. The Event Grid is stream-based and does not need any further settings except for the default ones.

In the benchmark, the Event Grid trigger deploys a storage account and an Event Grid system topic. A function app is then deployed and subscribed to the storage

account (publisher) whenever an event is sent to the storage account. To run the Event Grid trigger both the storage account and storage account container are sent as a query string to the invocation HTTP endpoint. The endpoint firstly initiates Application Insights to generate an operation id of the invocation. The endpoint then connects to the storage account container and uploads a new blob only containing the operation id. The trigger binding published to the Function app triggers as soon as it recognizes the new blob being uploaded to the storage account. The trigger-specific code extracts the operation id from the message and sends a trace to Application Insights.

4.2.5.5 HTTP

Azure HTTP trigger is a trigger available at Microsoft Azure that enables one to invoke an Azure Function via HTTP requests. The trigger has an HTTP response as the output binding. The HTTP trigger is the only synchronous trigger in Microsoft Azure, which has the benefit of a guarantee of not being OoO seen from a single user's perspective. Two examples of use cases of an HTTP trigger are to respond to webhooks or build serverless APIs [41].

In the benchmark, the HTTP trigger is published to a Function app that triggers whenever a specific HTTP endpoint is called. To run the trigger the URL of the HTTP endpoint is sent as a query string to the invocation HTTP endpoint. The invocation firstly initiates Application Insights to generate an operation id of the invocation and then sends an HTTP GET request targeting the URL that was given through the query string. The operation id is added as a query string for the function to be able to trace and connect the invoke with the receiver of the trigger.

4.2.5.6 Service Bus

A Service Bus trigger is used to react to either queue or topic messages that are sent to the event-driven Service Bus. In the benchmark design, a topic is used for the Service Bus. The trigger receives a topic message as input binding and can send a message as the output binding. The default settings have been used since the service bus has no delay since it uses a stream to listen for new messages [42].

In the benchmark, to deploy a Service Bus trigger both a Service Bus Namespace and a Service Bus Topic are deployed. A function app is then deployed and subscribed to whenever an event is sent to the topic. To run the Service Bus trigger both the Service Bus and Service Bus Topic name is sent as a query string to the invocation HTTP endpoint. The endpoint firstly initiates Application Insights to generate an operation id of the invocation. The endpoint then connects to the Service Bus Client and creates a sender based on the topic name. A new message batch is created from the sender with only one message containing the operation id. The trigger published to the Function app fires as soon as it recognizes the new message batch being uploaded in the Service Bus Topic, which extracts the operation id from the message and sends a trace to the Application Insights.

4.2.5.7 Queue Storage

The Queue storage trigger triggers an Azure Function whenever data is added to a certain queue storage. The trigger receives the queue message as input and can send new messages as the output bindings [43]. The Queue storage trigger is poll-based and not event-based. In contrast to the Blob storage trigger, Azure does provides the possibility to control the polling interval and batch sizes for the Queue storage trigger. However, a lower polling interval means an increase of events in Azure which in turn can lead to higher costs. The following settings have been used in the benchmark (that differs from the default values):

```
"maxPollingInterval": "00:00:00.100",
"batchSize": 1
```

In the benchmark, the Queue storage trigger is tested by deploying a queue client inside of a storage account. A Queue storage trigger is published to a Function app and subscribed to events from the queue data storage inside of the storage account. To run the Queue trigger both the storage account and queue client name are sent as a query string to the invocation HTTP endpoint. The endpoint firstly initiates insight to generate an operation id of the invocation, then connects to the queue client and sends a message only containing the operation id. The listening function in the function app fires as soon as it recognizes the new message in the queue, which extracts the operation id and sends a trace to Application Insights.

4.3 Benchmark Execution

The experiment was run three times. For the bursty workload with five (1, 10, 50, 100, and 300) burst sizes, each trigger type (7) was invoked up to 900 times, per burst size, per experiment run, for each runtime. One exception is that with a burst size of 1, the target sample size was set to 50 for each run. The exception was due to the workload design with 10 seconds of NBP in-between bursts, see 4.6a, which would result in at least 14h to run all trigger types for a burst size of 1. The invocations for the bursty workload adds up to $(7*900*4*3*2)+(7*50*3*2) = 153,000$.

For the constant workload where inter-arrival time is controlled (1, 10, 25, 50, 100, 150, 250), the target sample size is 500, per inter-arrival time, per trigger type, per experiment run, for each runtime. This sums up to $500 * 7 * 7 * 3 * 2 = 147,000$ invocations. Total invocations for running the experiment three times for both runtimes sum up to 300,000 invocations.

4.3.1 Environment

The benchmark experiment was conducted on a local computer running OS X. Since the experiment was run locally, variability in performance caused by network connectivity and transmission discrepancy is inherent due to the location of the cloud provider's servers where the cloud resources were hosted. The impact on the

benchmark results is deemed limited because the only part that can get affected is the first timestamp (t1) taken from triggering the invoker function through the API. All of the other traces and timestamps are performed on the provider's side and therefore not impacted.

4.3.2 Region

To reduce the potential influences of a network, the location of where the cloud resources were deployed was chosen in favor of the location where the experiment was conducted. For this thesis, the chosen server location was *northeurope* located in Ireland. Azure does have a server location in Sweden that is closer to where the experiment was conducted, however, it does not support the classic resource mode for Application Insights resources, and therefore not chosen.

Locally-redundant storage was used in order to make sure that the storage resources only used one data-center, in the primary region-zone, during executions of the experiment [44]. Microsoft does recommend to use Geo-zone-redundant storage which spans over multiple regions, but that is not suitable for this experiment.

4.3.3 Time

The choice of which time of the day the experiment was executed could potentially affect the results. This is due to the shared pool nature of cloud providers, with resources shared between tenants, which means that during peak hours it might affect resource availability and scalability. The execution of the benchmark was conducted during day-time between 9AM and 7PM, but in order to minimize the risk of time affecting the results, the three executions of the experiment were conducted at different times and days.

4.4 Data Analysis

In this section, a basic overview of the process of post-processing and analysis is presented. Subsection 4.4.2 presents the process of filtering out cold start traces. Subsection 4.5 presents the unit tests implemented to validate the analyzed data for correctness.

4.4.1 Analysis Scripts

The data analysis consists of two Python scripts, responsible to generate latency and reliability data. The data analysis starts by fetching data from the Application Insights database through an API query call. The received trace data is then processed in two scripts, as described in Section 4.2.3 with the results being saved in Comma-separated values (CSV) files.

In order to visualize the latency and reliability data, another two Python are used to generate plots. Cumulative distribution function (CDF) and violin plots will mainly be used to visualize the latency. The CDF plot is used to bring a more easily understanding overview of the data and compare each trigger with different workloads. The violin plot on the other hand can compare data between runtimes and triggers. Bar plots and tables will be used in order to visualize the reliability since that data is represented by percentages.

4.4.2 Filtering of Cold Starts

As mentioned in Sections 2.3 and 4.2.4, cold starts have a significant impact on latency and since cold starts are not a focus of this thesis it is necessary to exclude them from the collected data. During an investigation of how to detect cold starts from logs received from Application Insights, at the time of conducting this thesis, there were no solutions found. An explanation to why there is no direct way of identifying cold starts could be because Azure offers a premium plan for their Azure Functions that handle cold starts [45] by e.g. keeping the functions warm.

A workaround was implemented based on the fact that cold starts only occur in the first invocation of newly deployed or idle Azure Functions. Keeping track of the first invocations enable filtering of these during the analysis of the collected data. However, this workaround comes with a trade-off of filtering out samples that is the first invocation of an Azure Function but might not be a cold start, because it does not check whether the invocation is warm or cold. In addition, the detection of first invocations is VM instance-based, which means that during an experiment run, where N number of VM instances are used, N or N-1 warm invocation samples will get filtered out. The drawback of losing samples is not significant because it is a small number. In one of the experiment runs, 38 first invocations were excluded out of approximately 53,000 invocations.

4.5 Unit tests

To guarantee that the data analysis part of the benchmark is correctly built and behaves as intended, unit tests have been implemented into the benchmark. Unit tests are crucial to protect the analysis part of the benchmark against present and future bugs. The unit tests are automatically run before the analyzing scripts to verify that the scripts are still valid.

In order for the analysis data scripts to be able to run tests, a *–test* flag was added. Whenever the *–test* flag is sent, the analysis script uses the value from the flag to find a separate CSV-file, with mock-up data, in a test folder. The script reads the mock-up data, performs the unit tests, and saves the results within the test folder which is shown in the dotted box in Figure 4.7.

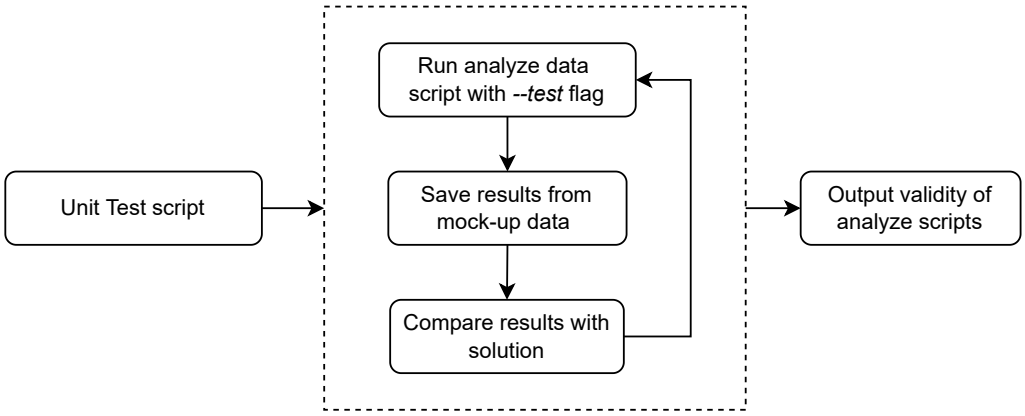


Figure 4.7: Overview of the unit test script.

To run all unit tests, the benchmark uses a separate python script that systematically calls each unit test in the reliability and latency analyzing scripts and compares it with their corresponding predefined solution within the test file. The tests cover different scenarios such as duplicate event deliveries, missing event deliveries, OoO occurrences, and latency. It also tests different invocation modes and input sizes.

5
Results

The results from the conducted experiments are grouped by latency, OoO, missing event, and duplicate event deliveries. Latency results are further divided based on two workload designs (bursty and constant workload), and each workload design shows results from two runtimes (Node.js and .NET). Note that all of the results are visualized in a logarithmic scale. These results will be mapped to the research questions, which are restated below:

RQ1: How does the choice of trigger types affect the latency of invoking functions?

- **RQ1.1**: How does the choice of runtime affect triggers' latency?
- **RQ1.2**: How does a bursty workload affect triggers' latency?
- **RQ1.3**: How does an inter-arrival time (IAT) controlled workload with a constant flow of trigger invocations affect triggers' latency?

RQ2: How reliable and consistent are results of a function delivered?

- **RQ2.1**: How frequent are function invocations OoO?
- **RQ2.2**: How frequent are function invocations missing or delivered multiple times?

5.1 Latency (RQ1)

This section presents the results for latency, covering different runtimes and workloads. Firstly, it is essential to establish a baseline for comparison between the results achieved from the experiments, which the violin plots in Figure 5.1 shows. The baseline is divided into two plots. The plot to the left shows the baseline for comparison between the number of invocations per burst. The reason for using one invocation per burst is because it is the smallest positive integer a burst could consist of. The second baseline, shown in the plot to the right, visualizes the baseline for comparison between different invoke IATs. The value of 250ms was chosen as the baseline due to the low number of OoO event deliveries at that workload, as seen in Figure 5.6.

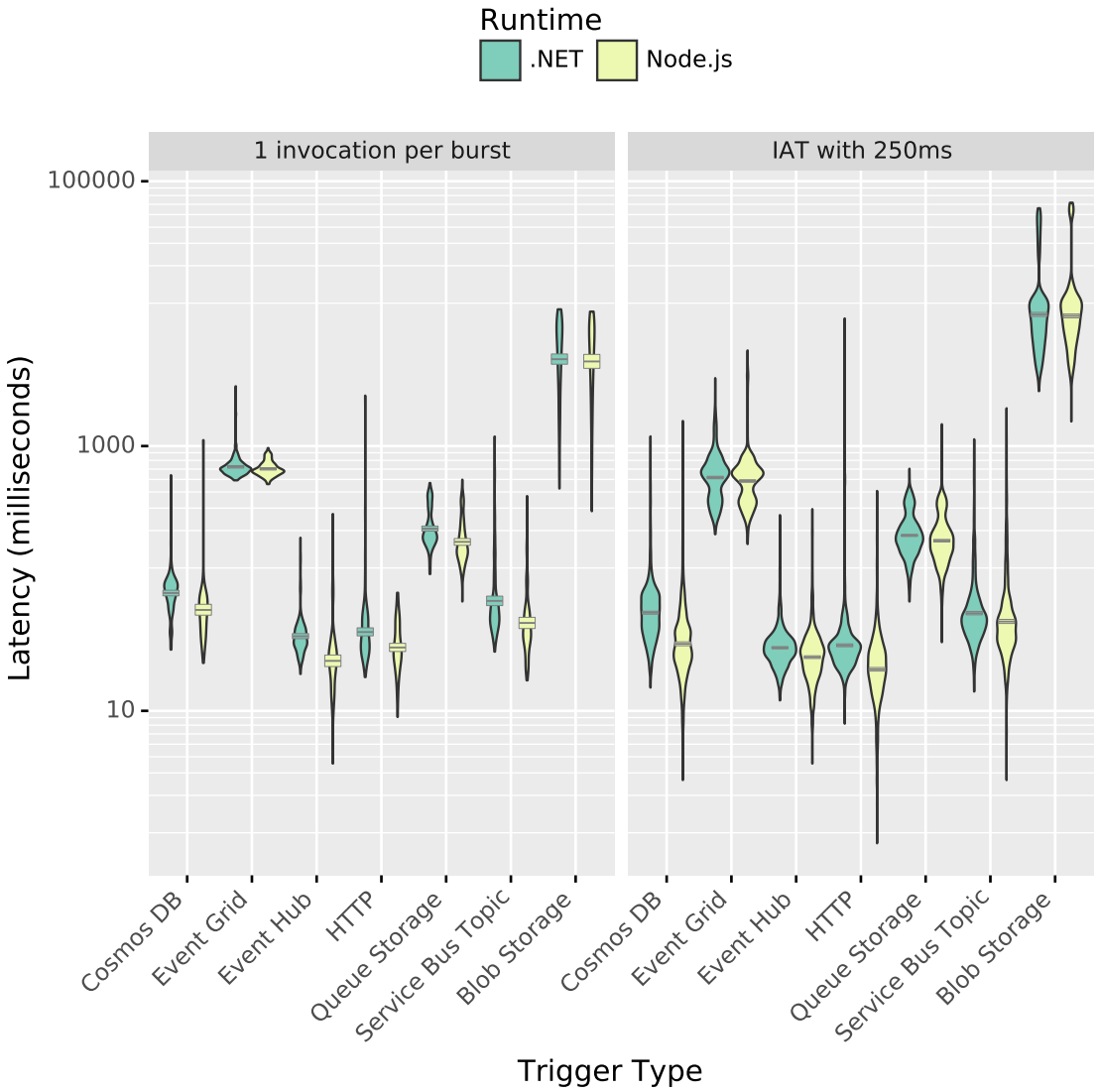


Figure 5.1: Two violin plots showing the baseline comparison for burst workload to the left and IAT controlled workload to the right for .NET and Node.js. The y-axis is based on a logarithmic scale. The gray-line across the violin body on each trigger in both plots is the confidence limit for the mean, without assuming a normal distribution.

5.1.1 Runtime (RQ1.1)

The triggers in the baselines in Figure 5.1 have similar latency, both regarding the runtimes and workloads. However, there seems to be marginally lower latency for the majority of triggers with burst workload for Node.js compared to .NET.

In order to further investigate how runtimes affect latency, multiple workloads were tested. In Figure 5.2, four larger burst workloads are shown, scaling from 10 to 300 invocations per burst for both .NET and Node.js. There seems to be no significant

difference in latency between the different runtimes of each trigger for the first three burst workloads (10 to 100 invocations). However, at the highest workload in Figure 5.2 one can notice that Node.js has a slightly higher latency compared to .NET for the majority of the triggers, which is the opposite compared to the baseline.

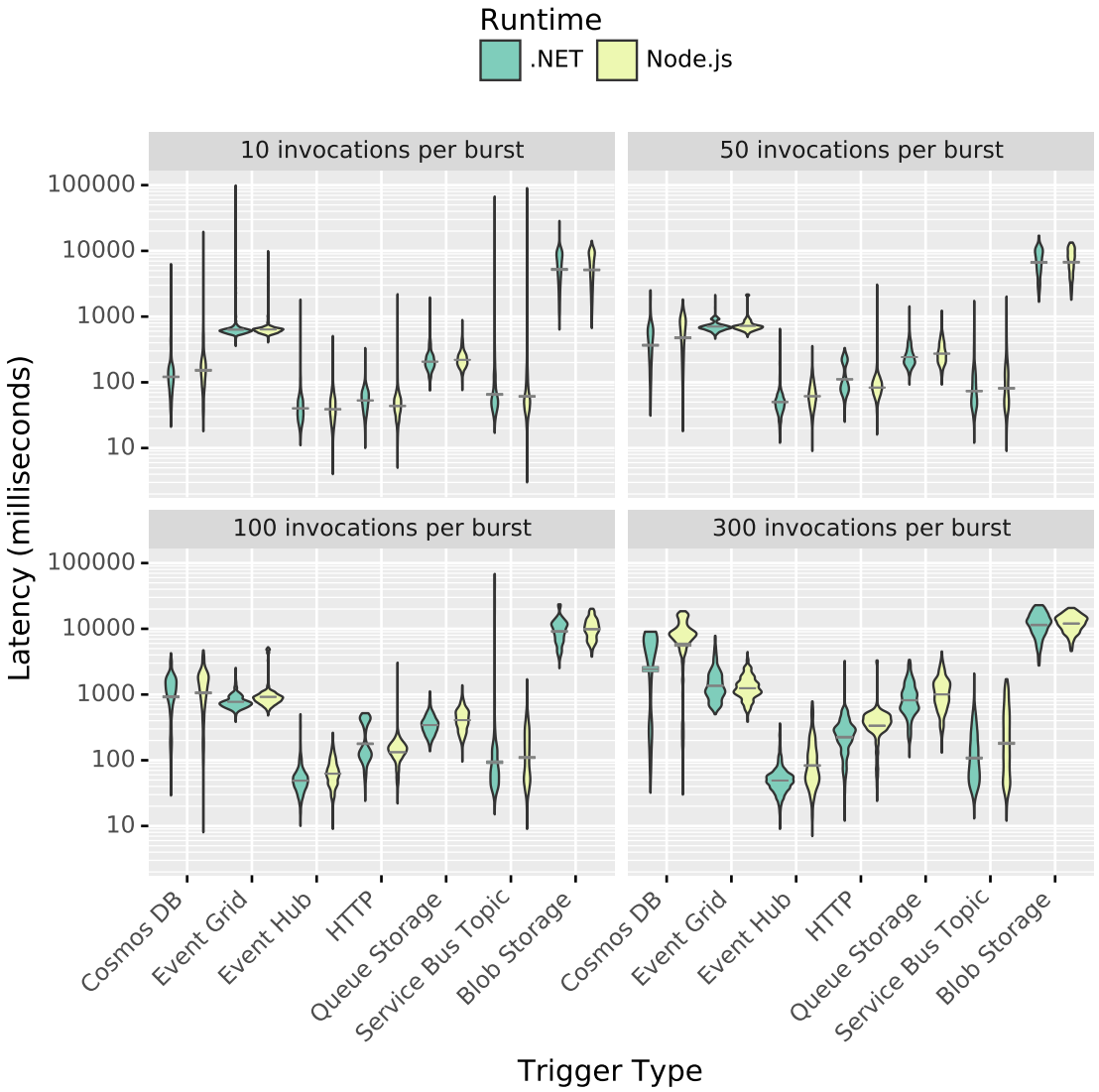


Figure 5.2: Violin plots showing latency for all triggers in both Node.js and .NET for different burst workloads. The y-axes are based on a logarithmic scale. The gray-line on each trigger is the confidence limits for the mean, without assuming any normality.

Furthermore, multiple other IATs were tested. Below, in Figure 5.3, the six lower IATs are shown. The plots scale from 1 to 150 milliseconds in delay between each invocation for both .NET and Node.js. Just as the highest burst workloads in 5.2, Node.js has a marginally higher latency at the most stressful workloads (IAT of 1ms

and 10ms) in Figure 5.3. However, there are no substantial differences with using the longer invocation delays.

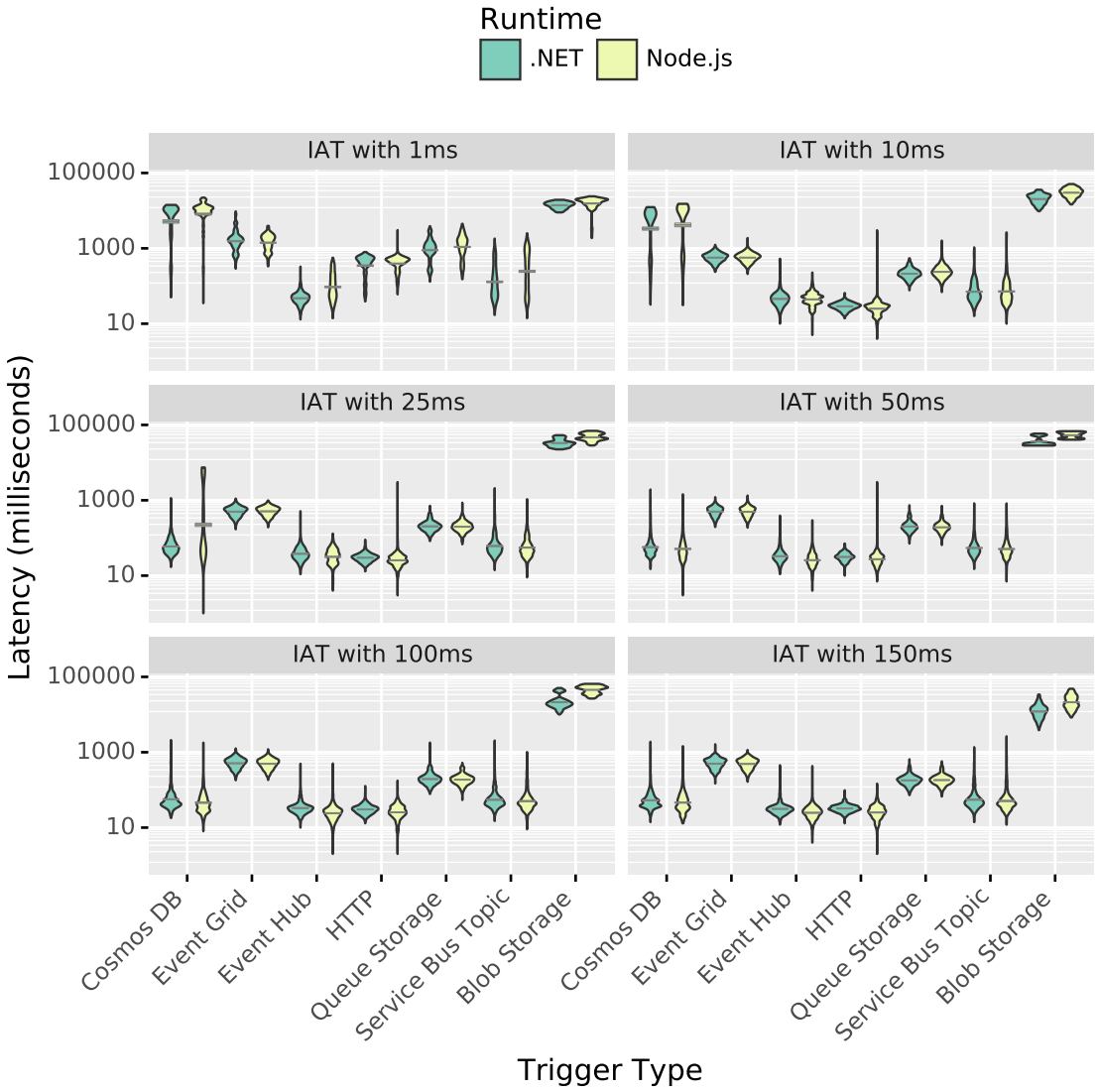


Figure 5.3: Violin plots showing latency for all triggers in both Node.js and .NET for different constant workloads. The y-axes are based on a logarithmic scale. The gray-line on each trigger is the confidence limits for the mean, without assuming any normality.

5.1.2 Burst (RQ1.2)

There is a clear correlation between latency and the burst sizes since the trigger types in Figure 5.2 do not have the same behavior at the higher burst workloads, compared to the burst baseline in Figure 5.1. It seems to be that the tails of the violins are on average higher in the lower burst workloads, but decrease as the burst sizes gets larger.

Furthermore, inspecting the violin plot tails of the service bus topic in 5.2, it seems to suffer from unusual long tails at the lower burst sizes. There are almost no tails, except for the oval violin, at the higher invocations per burst. Another trigger type where the tails behave differently is the Cosmos DB. Whenever Cosmos DB receiving a high burst size, the trigger gets a long bottom tail and a high latency.

Figure 5.4 shows the CDFs for all triggers' invoked with the bursty workload in .NET. The CDF plots show that the trigger types have an increase in latency in correlation with a higher burst. An exception is the Event Hub, which is the most stable trigger type in terms of latency, only having a mean from 39ms (1 in burst size) to 55ms (300 in burst size) corresponding to only +41% increase. The Event Grid, Service Bus Topic, and Queue Storage on the other hand also being stable, until the highest workload of 300 invocations per burst. Further, both the Cosmos DB and HTTP triggers have gaps in the CDF showing higher latency from the lowest burst size and upwards, where the Cosmos DB stands outgoing from a mean of 85ms to 4477ms in latency (+5170%). The Blob storage has a more unique CDF curve with the highest latency of all going from a mean of 5558ms to 12469ms in latency (+124%).

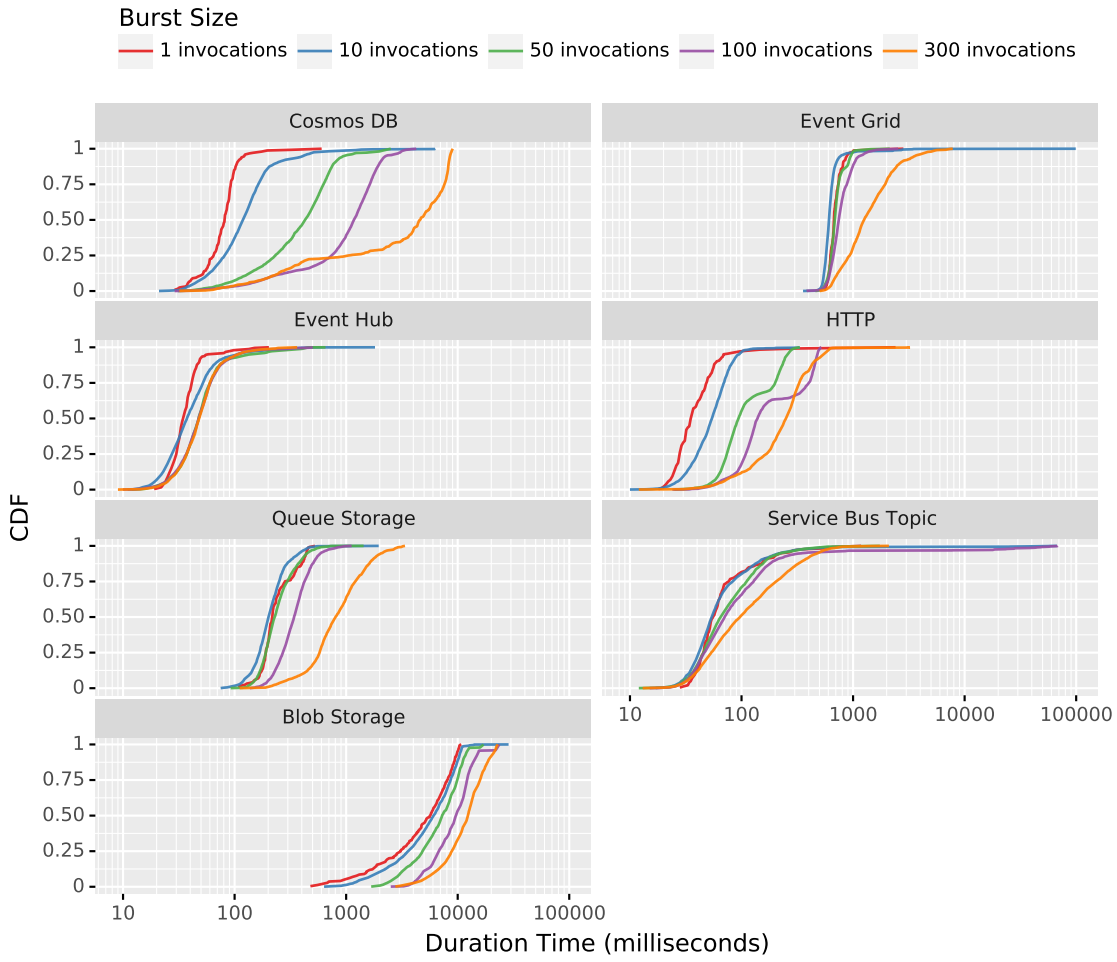


Figure 5.4: CDF plots showing latency for all triggers in .NET for different burst workloads. The x-axes are based on a logarithmic scale.

5.1.3 Inter-arrival time (RQ1.3)

There is a clear correlation between IAT and latency by comparing the IAT baseline in Figure 5.1 and the lower IAT in Figure 5.3. However, the tails are not as long for the longer invocation delays, as compared to 300 invocations per burst in Figure 5.2, but instead stay consistent until the lowest delays at 1ms where they almost disappear (with Cosmos DB as an exception). As shown in the bursty workload in Figure 5.2, Cosmos DB tails change for certain workloads. The same behavior is shown in the IAT-controlled workload where Cosmos DB trigger initially has a tail upwards but at the lowest IATs (1ms and 10ms) change to a tail downwards.

Furthermore, in Figure 5.5 all triggers are visualized in another CDF plot, but for the IAT controlled workload. In contrast to bursty workloads in Figure 5.4, the IAT shows that all triggers, except for Blob Storage, are stable in terms of latency with invocation delay from 250ms to 25ms, but for the lowest delays at 1ms, a clear latency gap is present (except for Blog Storage and Event Hub). Cosmos DB on

the other hand is the only trigger that also has a large increase in latency for IAT at 10ms, while the rest seems to handle the workload well. However, there is one trigger that is stable for all invocation delays, the Event Hub trigger. In contrast, the Blob Storage, similar to the bursty workload in Figure 5.4, has a unique behavior compared to the rest of the triggers. The most noteworthy for Blob Storage is that the IATs do not seem to have a clear correlation to the latency where for example the lowest IAT at 1ms is the third fastest.

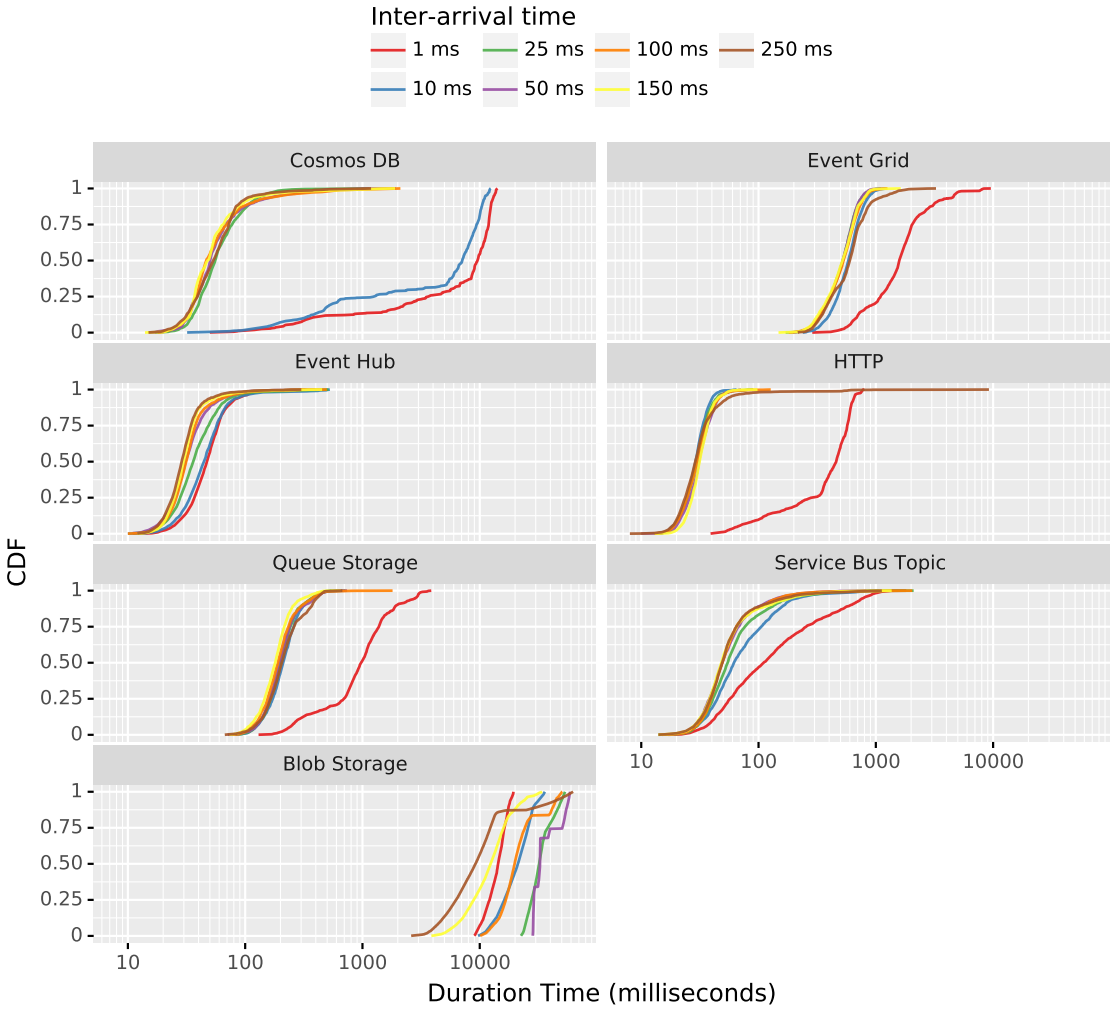


Figure 5.5: CDF plots showing latency for all triggers in .NET for different IATs. The x-axes are based on a logarithmic scale.

5.2 Reliability (RQ2)

This section presents the results gathered for analysis of reliability in terms of OoO, missing, and duplicate event deliveries. The results for OoO event deliveries are presented in Section 5.2.1 and visualized using bar plots in Figure 5.6, and the results for missing and duplicate event deliveries are presented in 5.2.2 and summarized in Table 5.1, 5.2, 5.3, and 5.4. Some of the cells of the tables are colored blue based on the percentage of observed missing and duplicate event deliveries. This is because the frequency of some observations is too low to determine the relevancy. Therefore, those observations are not colored and not further discussed.

5.2.1 Out-Of-Order (RQ2.1)

Figure 5.6 contains the results for OoO event deliveries for both Node.js and .NET. The plots show the probability of OoO event deliveries for each trigger type. The common observation across both runtimes is that the longer of an invocation delay, the fewer OoO deliveries there are. Further, almost all trigger types with an invocation delay of 250ms result in almost zero OoO event deliveries. However, the observation is less significant for the event grid where for each longer invocation delay, only a small decrease can be observed, and for 250ms there is still more than 25% OoO event deliveries. The results from the Blob storage trigger show low OoO for all values of invocation delays.

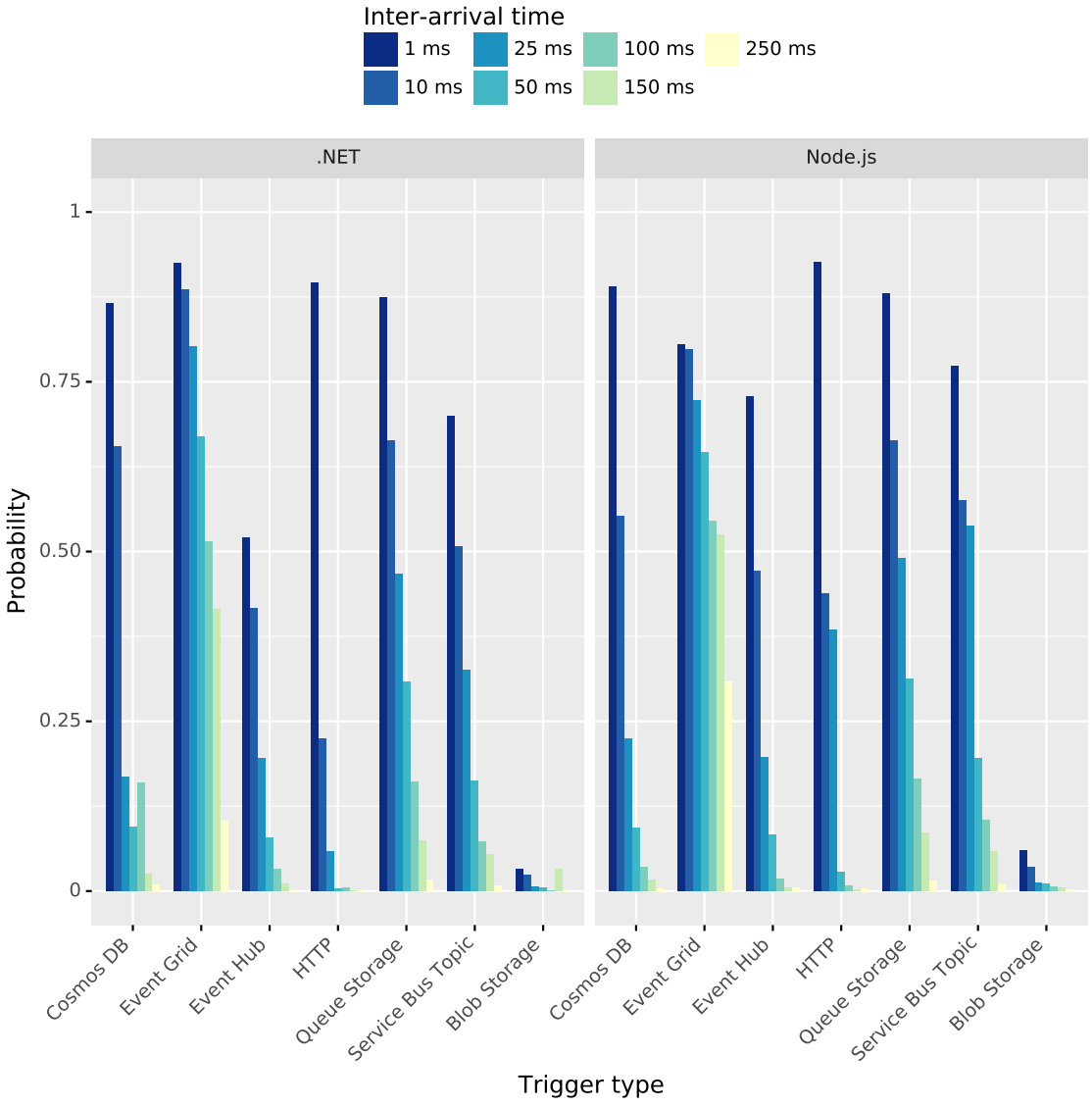


Figure 5.6: Bar plots showing out-of-order results for IAT for both Node.js and .NET.

5.2.2 Missing events (RQ2.2)

Table 5.1 shows that missing event deliveries common for bursty workloads for both runtimes are observed for all trigger types except for HTTP and Event grid trigger. Observed missing deliveries are present for Blob storage trigger already at burst size 10, while for Cosmos DB missing events are only observed for burst size 300 with a difference of approximately 20% between the runtimes.

		Burst size (invocations) [Samples]				
Trigger	Runtime	1	10	50	100	300
Blob storage	Node.js	-	33.5% [753]	61.2% [1379]	65.6% [1575]	68.4% [1849]
	.NET	-	31.8% [591]	60.5% [1363]	63.6% [1528]	67.1% [1813]
Cosmos DB	Node.js	-	0.04% [1]	0.09% [2]	0.04% [1]	41.3% [1115]
	.NET	-	-	-	-	62.8% [1695]
Event Hub	Node.js	-	-	-	0.04% [1]	0.07% [2]
	.NET	-	-	-	-	-
Service Bus Topic	Node.js	-	0.04% [1]	-	-	0.04% [1]
	.NET	-	-	-	-	-
Queue Storage	Node.js	-	-	0.04% [1]	0.08% [2]	0.12% [3]
	.NET	-	-	-	-	-
Others[1]	Node.js	-	-	-	-	-
	.NET	-	-	-	-	-

Table 5.1: Missing event delivery results with **bursty** workload. Zero occurrences is denoted with '-' symbol.

Table 5.2 shows, similar to the bursty workload, that most missing deliveries come from Cosmos DB trigger and Blob storage trigger for both runtimes. Both of them perform poorly for invocation delays of 1ms and 10ms. With increasing invocation delay, the missing deliveries are decreased for both triggers, except for Blob storage with a delay of 250ms which resulted in 6.21% missing event deliveries. The only trigger with no missing event deliveries observed for both workloads is the HTTP trigger.

		Invoke delay (ms) [Samples]						
Trigger	Runtime	1	10	25	50	100	150	250
Blob storage	Node.js	55.5% [833]	11.7% [176]	3.0% [45]	0.2% [3]	0.27% [4]	0.07% [1]	6.21% [93]
	.NET	58.5% [878]	14.5% [218]	3.7% [56]	1.2% [19]	0.07% [1]	0.07% [1]	-
Cosmos DB	Node.js	35.3% [529]	52.1% [782]	-	-	-	-	-
	.NET	57.8% [867]	37.8% [568]	-	-	-	-	-
Event Grid	Node.js	-	0.06% [1]	0.06% [1]	0.06% [1]	-	0.06% [1]	-
	.NET	-	-	-	-	-	-	-
Event Hub	Node.js	-	-	-	-	-	0.07% [1]	-
	.NET	-	-	-	-	-	-	-
Service Bus Topic	Node.js	-	-	-	-	0.07% [1]	-	-
	.NET	-	-	-	-	-	-	-
Queue Storage	Node.js	0.07% [1]	0.07% [1]	-	0.07% [1]	-	-	-
	.NET	-	-	-	-	-	-	-
HTTP	Node.js	-	-	-	-	-	-	-
	.NET	-	-	-	-	-	-	-

Table 5.2: Missing event delivery results with **IAT controlled** workload. Zero occurrences is denoted with '-' symbol.

[1] HTTP, Event Grid.

5.2.3 Duplicate events (RQ2.2)

Table 5.3 shows duplicate event deliveries for the bursty workload. No duplicates were observed for burst size 1. For Cosmos DB trigger, surprisingly there were no duplicate deliveries for Node.js, while some occurred for .NET. The Event grid trigger had duplicate deliveries for burst sizes of 100 and 300 for Node.js, and only a couple with burst size 10 for .NET. Further, for burst sizes 10 and 50 Service bus topic also have some duplicates.

		Burst size (invocations) [Samples]				
Trigger	Runtime	1	10	50	100	300
Cosmos DB	Node.js	-	-	-	-	-
	.NET	-	2.2% [50]	10.1% [238]	6.0% [153]	-
Event Grid	Node.js	-	-	0.04% [1]	6.1% [156]	5.2% [148]
	.NET	-	0.67% [15]	-	-	-
HTTP	Node.js	-	1.0% [24]	0.04% [1]	0.33% [8]	-
	.NET	-	-	-	-	-
Service Bus Topic	Node.js	-	1.5% [43]	6.4% [156]	0.21% [5]	-
	.NET	-	0.04% [1]	-	0.04% [1]	-
Others[2]	Node.js	-	-	-	-	-
	.NET	-	-	-	-	-

Table 5.3: Duplicate event delivery results with **bursty** workload. Zero occurrences is denoted with '-' symbol.

Table 5.4 shows duplicate event deliveries for the constant IAT-controlled workload. As can be seen, only Event Grid and HTTP have duplicate event deliveries, where Event Grid has duplicates up to invoke delay of 150ms. The triggers that did not produce any duplicate event deliveries are Blob storage, Queue storage, and Event Hub.

		Invoke delay (ms) [Samples]						
Trigger	Runtime	1	10	25	50	100	150	250
Event Grid	Node.js	5.5% [91]	6.1% [99]	7.5% [122]	6.0% [97]	6.7% [109]	3.5% [55]	-
	.NET	-	-	-	-	-	-	-
HTTP	Node.js	0.2% [4]	0.1% [2]	0.2% [4]	-	-	-	0.33% [5]
	.NET	-	-	-	-	-	-	-
Others[3]	Node.js	-	-	-	-	-	-	-
	.NET	-	-	-	-	-	-	-

Table 5.4: Duplicate event delivery results with **IAT-controlled** workload. Zero occurrences is denoted with '-' symbol.

[2]Blob Storage, Queue Storage and Event Hub.
[3]Blob Storage, Cosmos DB, Service Bus Topic, Queue Storage and Event Hub.

6
Discussion

This chapter reflects upon and discusses the results from Chapter 5. Findings from the discussions are summarized into a table containing guidelines and suggestions on usage, benefits, and drawbacks of each Azure trigger type regarding the metrics used during experimentation. Thereafter, threats to validity are discussed, reflections of characteristics introduced in Section 2.7 are revisited, and lastly, the eight reproducibility principles by Papadopoulos et al. [30], introduced in Chapter 3, are discussed.

6.1 Latency

A trigger that changes behavior as the workload got more stressful was Cosmos DB, which is best visualized in the CDF plot for IAT in Figure 5.4 where it is very stable until workloads of 10ms and 1ms. At the two lowest delays, Cosmos DB seems to struggle to process the invocations where the CDF curve changes completely. The same can be seen in the Violin plot in Figure 5.2, where Cosmos DB trigger suddenly have a higher latency with an extremely long latency tail below the body. In terms of latency critically application, this behavior is not optimal and creates a lot of uncertainty.

Blob Storage is by far the trigger with the highest latency across all workloads with being the only trigger reaching latencies over 10 seconds. It is interesting that at even the least stressful workloads, latency way over five seconds can be observed, which no other trigger is even close to. In contrast to the burst workloads, in Figure 5.4 where growth in latency is presented in correlation with the workload, the latency of the IAT workloads seems to finish in disorder. The results indicate that Blob Storage should not be used on any latency-critical occasions. However, an advantage of this thesis is that the Event Grid and Blob Storage can be compared head-to-head since the Event Grid trigger is based on blob uploads as described in Section 4.2.5.4. By comparing the results from these two triggers in Figure 5.4 and Figure 5.5 one can see that the latency of Event Grid is more stable and better, making it very difficult to motive why to use the Blob storage trigger.

Event hub was the only trigger that stood out in terms of stability and low latency for all workloads, which is visible in the CDF plots in Figure 5.4 and Figure 5.5. Even at the highest and most stressful workloads, the Event Hub performed well with

a latency mean of under 100ms. The fact that Event Hub performed best at higher workloads was not surprising due to the purpose of an Event hub being designed for stream processing big data with very low latency [46].

6.2 Reliability

The results presented in Section 5.2.1 suggest that with an increased invocation delay, fewer OoO deliveries will occur, which is the behavior one would expect. The longer the time difference between consecutive invocations, the more time requests have to get the requests processed and receive responses before the next request is sent. Thus, resulting in less OoO. There are, however, two triggers that stand out from the rest, the Event Grid and Blob storage trigger [37].

There is almost no OoO from the Blob storage trigger for both runtimes compared to the other triggers. Looking at Figures A.1c and A.1d for missing executions, even though there are missing event deliveries for Blob storage that impact OoO deliveries, it can not explain the significant lower OoO observed compared to the other triggers. The expectation is that the results should be similar to the other asynchronous triggers. The exact reason for this outcome is not pinpointed but as mentioned in Section 4.2.5.1, the fact that the Blob storage trigger is poll-based, uses a hybrid between inspecting logs and scanning containers, and scans 10,000 blobs at a time to detect changes in a blob container is a reasonable explanation of the low probability of OoO observed since the order of the uploads in the blob containers is already determined during scans. As mentioned in Section 5.2.1, the Event grid trigger has many OoO across all the different invocation delays, which is in line with Azure's claim that the event grid trigger does not have an order guarantee for event delivery [6]. With the results gained from this study, it is possible to get an indication of the number of OoO event deliveries for the invocation delays used for the specific workload.

Looking at the results for missing event delivery for bursty workloads, Cosmos DB and Blob storage are the two triggers for which the most missing deliveries are observed. Focusing on the Cosmos DB trigger, missing deliveries are only seen for burst size of 300, and that is also when the trigger's performance in terms of latency is declining, see Figure 5.4, the orange line. The reason for the declining performance is speculated to be because of the overload of requests. It is necessary to test for more fine-grained burst sizes between 100 and 300 to more accurately pinpoint the occurrence of the issue. For Blob storage, the issue is even more apparent, where already at burst size of 10 over 30% of the invocations are missing for both runtimes. This percentage increases with each increased burst size. The reason is the same as previously mentioned, the nature of the polling used the for Blob storage trigger. Therefore, there is no guarantee that all events are captured and logs could also be missed, which is in line with what Azure claims [37].

The trigger types that have the most missing event deliveries for the IAT-controlled workload are the same triggers with many missing event deliveries for the bursty

workload, namely, Cosmos DB and Blob storage. From the results, the two triggers perform poorly with invoke delays of 1ms and 10ms. This observation could be caused by the implementation of both triggers, where both detects or scans for changes, e.g. upload/deletion of a file in a container, and insertion/update of a database. Further, the HTTP trigger appears to not produce any missing event deliveries for the runtimes. A simple and valid explanation could be tht the HTTP trigger is a synchronous trigger.

The reliability, in terms of duplicate deliveries for the bursty workload, Cosmos DB trigger has the most apparent difference between Node.js and .NET. Duplicates are also observed for Event grid, HTTP, and Service Bus Topic trigger. For the IAT-controlled workload, the most duplicate event deliveries were observed for the Event Grid trigger with Node.js and no duplicates with .NET. There are no guarantees that event deliveries will only be delivered once, and from the results of this study, one should expect that duplicates could occur. However, it is explicitly mentioned for Event Grid that events can be delivered more than once and that it is the responsibility of the event handler to be implemented defensively [47], which might apply to the rest of the triggers. Further, Blob storage, Queue storage, and Event Hub triggers have no duplicates observed. For the Blob storage trigger it could be explained by blob receipts which help Azure Functions runtime to ensure that the blob trigger will only get triggered once for the same, new, or updated blob [37]. For the Queue storage trigger it could be explained by the peek lock pattern that is implemented for the queue trigger, which means that if the function is successful, then the execution finishes and the message is deleted [43]. Since the message is deleted, no duplicate event delivery can be triggered by the same message. For Event Hub it might be because of the configuration *maxEventBatchSize* was set to one.

6.3 Threats to Validity

This section discusses the threats to validity of the experiment performed in this thesis, namely construct, internal, and external validity.

6.3.1 Construct Validity

Construct validity is related to the correctness of the measurements. It addresses whether the results collected are what is intended to be measured.

The threat, Mono-Method Bias, refers to the method that is used to collect correct measurements, in this case, latency. This threat concerns the service provided by Azure, Application Insights, to provide accurate and correct timestamps. Furthermore, processing of the traces, extraction of relevant timestamps, and calculation of latency are also steps that might affect the correctness. As of writing, there is no other drastically different method for timestamp extraction, known to the authors, that would have an impact on the correctness of the measurements. Methods have to rely on utilizing Application Insights to instrument the code and retrieve server-sided timestamps. Section 4.2.3 addresses this threat by describing exactly at what

timestamps the latency is calculated, and also the fact that three repetitions of the experiment were conducted at different points in time further reduces the threat.

6.3.2 Internal Validity

Internal validity addresses the confidence of whether the cause-effect relationship observed in the results of this study is not affected by a confounding factor. In the context of this study, one of the most relevant threats to internal validity is concerning the black-box nature of the experimentation environment provided by Microsoft Azure. The internal infrastructure and implementation are not publicly available and therefore can not be controlled for, which means that there might be internal factors at the cloud providers that could affect the measures and data collected in this study. Details about the shared infrastructure between cloud users at cloud providers are not accessible, and therefore it is not possible to determine potential noise that is introduced by other customers. For example, a heavy workload from another customer in the same virtual machine that shares the same hardware might affect the available resources such as CPU and bandwidth, which in turn influence the outcome of this study.

Another threat concerns clock synchronization, specifically the precision and accuracy of instruments used for measurements in distributed systems. Najafi et al. [48] argue that understanding how to measure time accurately is critical to system research and that all system evaluations should make sure that the clocks are calibrated before running experiments. Failure of identifying inaccurate clocks or not calibrating before conducting experimentation could potentially result in actual benchmark errors. The threat lies in the nature of cloud distributed system that consists of numerous interconnected virtual machines (VM), which each have a local clock that can be affected by various operations. According to Microsoft [49], the potential consequences are e.g. authentication failure, incorrect time of logs, and inaccurate billing. If only one VM is used, the effects might not be significant unless there is strict timekeeping. However, in practice, there are often many interconnected VMs operating together, and since the clock accuracy error accumulates it could result in erroneous synchronization. The issue is best addressed from the cloud provider's side since developers using FaaS do not have access to the VMs. Microsoft Azure addresses the problem by having time synchronization services running on all of its VMs. The service knows what time servers to use and periodically checks if a clock needs to be calibrated. In addition, the synchronization between Azure hosts and internal Microsoft time servers is provided by Microsoft-owned Stratum 1 devices, with GPS antennas [49].

6.3.3 External Validity

External validity concerns the generalizability of the results and insights derived from the study to other settings within the same domain.

The most relevant threats to external validity are to what extent the outcome

and insights are generalizable to other trigger types, and other cloud providers' trigger types. Methodology-wise it is feasible to execute the same procedures to derive benchmarks for other trigger types and cloud providers, which will, in turn, enable meta-analysis and comparisons between the results from different cloud providers' trigger types with similar functionality. However, due to the difference in internal infrastructure, implementations, and representations of various trigger types throughout cloud providers, it is not possible to generalise the insights from this specific study to other cloud providers' trigger types.

6.4 Building a Benchmark

By revisiting the characteristics discussed by Kisoski et al. [20], helps facilitating discussion about the implementation of the benchmark.

The relevance and usefulness of the created benchmark are considered to be beneficial for researchers that intend to get insights into the latency and reliability of Azure triggers. For practitioners, the benchmark itself might not be relevant but the findings from the data, gathered with the benchmark, can be highly valuable. The benchmark's breadth of applicability involves all Azure triggers, and adding new trigger types to the benchmark is a fairly simple process. The workloads are designed to be relevant for scenarios that produce high tail latency, and for creating an order semantic to test for OoO deliveries, which was the focus of this study. Creating new workloads with K6 to use with the implemented benchmark is a simple matter of just writing new code for specifying the desired pattern invocation behavior. The intended purpose of comparing trigger types from one provider poses limitations on the benchmark regarding breath of applicability and scalability because transferability of insights to other cloud providers is not possible. The main reasons for lacking of transferability is no direct mapping between different cloud providers' trigger types and difference in underlying implementations of infrastructure and triggers between providers.

In terms of reproducibility, much effort has been devoted to automating the execution process of the benchmark to conduct experiments by running a single script. The script handles everything from automating the deployment of cloud resources using Pulumi, parameterizing essential experiment input values, to executing load tests automatically. Automated execution of the benchmark reduces the effort needed to run the benchmark and the risk of bugs introduced by manual setup.

The fairness of comparing Azure trigger types comes down to only trigger-specific configurations and software because the hardware is handled by the cloud provider and no hardware is calibrated or manipulated by the benchmark. For a fair comparison between the triggers, some compromises have been made, such as configuring the polling intervals of poll-based triggers to behave similarly to an event-based trigger. This will, however, increase the cost of poll-based triggers because of the increased number of poll events. Another example of a compromise is configuring event batch sizes of triggers that use batches when processing the events.

Verifiability is addressed by implementing unit tests focusing on verifying the logic of calculating out of order, duplicate, and missing event deliveries. The work in this area is rather thin and the benchmark would benefit from more testing.

6.5 Reproducibility

In Section 6.4, reproducibility was briefly discussed, and in this section, more effort will be devoted to further discussing reproducibility based on the eight principles by Papadopoulos et al. [30] introduced in Chapter 3.

P1: *Repeated experiments (statistical). Decide how many repetitions with the same configuration of the experiment should be run, and then quantify the confidence in the final result.*

In Section 4.3, a paragraph has been devoted to present numbers regarding sample sizes (invocations). Papadopoulos et al. observe that P1 is often only partially fulfilled by performing a non-justified number of repetitions or by choosing a longer duration for experiment runs, which is not sufficient. To justify the sample sizes used in this study, it is necessary to provide a statistically sound assertion to ensure that the results are not by chance. In Section 4.3, the experiment setup, the number of repetitions executed and data points gathered, are presented.

Hoefler and Belli [50] establish common rules to help experimenters to improve the interpretability of results and introduce techniques to analyze collected data to ensure that the conclusions, which are based on the data, are reliable. For summarizing results, one has to choose which statistical method to use, parametric or non-parametric. The choice depends on the distribution of the data, and after checking the results from this study with Q-Q plots it was possible to conclude that the distribution of the collected data is not normal. Therefore, a non-parametric technique could be used for statistical inference and reporting confidence intervals.

R provides a library called **boot**[1] to generate non-parametric bootstrap confidence intervals. Non-parametric bootstrap allows for estimations on parameters, in this case, the mean, of a population or probability distribution from a set of observations without having to assume the distribution. Tables 6.1 and 6.2 are attempts to summarize the results using non-parametric bootstrap confidence intervals with 6000 repetitions to find the confidence interval of the actual population mean compared to the sample mean.

[1] https://cran.r-project.org/web/packages/boot/boot.pdf

	Burst size	Blob storage	Cosmos DB	Event Grid	Event Hub	HTTP	Service Bus Topic	Queue Storage
Bootstrapping 95% Confidence Interval (BCa) - **Node.js**	1	−8.81% +8.67%	−22.04% +42.27%	−2.13% +2.18%	−13.22% +27.25%	−6.88% +7.91%	−13.47% +20.44%	−6.32% +7.42%
	10	−2.64% +2.59%	−15.05% +22.06%	−1.65% +2.90%	−3.07% +3.64%	−5.88% +9.31%	−36.86% +63.02%	−1.32% +1.35%
	50	−2.56% +2.58%	−2.47% +2.30%	−1.23% +1.37%	−2.23% +2.32%	−2.93% +6.90%	−4.43% +5.40%	−1.77% +1.83%
	100	−2.49% +2.43%	−2.40% +2.62%	−2.80% +3.12%	−2.02% +2.27%	−2.84% +5.33%	−4.18% +3.80%	−1.58% +1.62%
	300	−1.92% +1.93%	−2.71% +2.73%	−1.57% +1.62%	−3.84% +4.02%	−2.49% +3.10%	−4.05% +4.39%	−2.07% +2.28%
Bootstrapping 95% Confidence Interval (BCa) - **.NET**	1	−6.52% +6.60%	−5.58% +9.60%	−2.44% +4.69%	−5.29% +8.80%	−31.05% +87.82%	−14.31% +22.86%	−4.27% +5.01%
	10	−6.60% +6.61%	−5.69% +10.00%	−2.42% +4.85%	−5.25% +8.26%	−30.42% +79.62%	−13.87% +23.63%	−4.23% +4.74%
	50	−2.87% +2.85%	−3.28% +3.46%	−0.77% +0.78%	−3.76% +4.33%	−2.29% +2.46%	−4.42% +5.23%	−1.59% +1.68%
	100	−2.65% +2.72%	−2.39% +2.36%	−1.03% +1.16%	−2.91% +3.42%	−3.00% +3.01%	−20.20% +25.15%	−1.82% +1.84%
	300	−2.51% +2.59%	−4.25% +4.13%	−2.36% +2.58%	−2.10% +2.34%	−3.38% +4.43%	−3.85% +4.34%	−2.37% +2.48%

Table 6.1: Confidence intervals of the true population mean expressed as a percentage of its mean for **latency** data collected from the **bursty** workload

	Invoke Delay	Blob storage	Cosmos DB	Event Grid	Event Hub	HTTP	Service Bus Topic	Queue Storage
Bootstrapping 95% Confidence Interval (BCa) - **Node.js**	1	−2.18% +1.99%	−2.83% +2.90%	−2.42% +2.39%	−4.46% +4.37%	−2.07% +2.46%	−4.92% +5.08%	−3.15% +3.26%
	10	−1.45% +1.53%	−4.90% +4.70%	−1.46% +1.59%	−2.21% +2.42%	−7.58% +34.95%	−9.27% +12.18%	−2.09% +2.24%
	25	−1.19% +1.16%	−7.80% +8.33%	−1.40% +1.46%	−2.37% +2.53%	−13.46% +32.33%	−5.19% +6.38%	−1.75% +1.78%
	50	−0.83% +0.85%	−7.25% +9.33%	−1.59% +1.63%	−3.10% +3.56%	−12.03% +30.27%	−4.83% +6.07%	−1.60% +1.77%
	100	−1.20% +1.13%	−8.93% +12.25%	−1.67% +1.64%	−3.72% +6.05%	−2.23% +2.41%	−5.71% +7.14%	−1.69% +1.75%
	150	−2.15% +2.15%	−7.16% +9.99%	−1.59% +1.58%	−2.85% +4.21%	−2.29% +2.59%	−7.36% +11.12%	−1.69% +1.83%
	250	−4.93% +5.87%	−8.19% +13.27%	−3.30% +4.18%	−3.55% +4.92%	−4.61% +6.78%	−7.02% +9.66%	−2.23% +2.32%
Bootstrapping 95% Confidence Interval (BCa) - **.NET**	1	−1.46% +1.48%	−4.26% +4.25%	−3.31% +3.64%	−2.45% +3.11%	−2.82% +2.80%	−5.46% +6.02%	−3.50% +3.49%
	10	−1.56% +1.66%	−4.23% +4.10%	−1.40% +1.46%	−4.32% +5.54%	−1.49% +1.49%	−5.00% +6.27%	−2.01% +2.17%
	25	−1.25% +1.18%	−3.74% +5.02%	−1.45% +1.48%	−3.76% +4.90%	−1.71% +1.82%	−5.47% +8.55%	−2.09% +2.23%
	50	−1.41% +1.45%	−6.28% +8.18%	−1.46% +1.42%	−2.88% +3.92%	−1.59% +1.76%	−4.07% +5.23%	−2.22% +2.52%
	100	−2.52% +2.50%	−8.38% +11.73%	−1.54% +1.62%	−3.18% +4.19%	−1.63% +1.86%	−5.31% +8.65%	−2.36% +3.21%
	150	−2.25% +2.37%	−7.81% +10.32%	−1.61% +1.63%	−3.23% +4.37%	−1.39% +1.46%	−5.70% +7.02%	−1.70% +1.85%
	250	−4.96% +4.95%	−5.00% +6.96%	−2.24% +2.66%	−2.73% +3.40%	−18.94% +59.52%	−5.61% +7.36%	−1.94% +2.08%

Table 6.2: Confidence intervals of the true population mean expressed as a percentage of its mean for **latency** data collected from the **IAT controlled** workload

The results from bootstrapping are to give a rough estimate and an indication of whether the target sample size used for the experiments is considered appropriate to draw trustworthy conclusions. The 95% confidence interval shows that there is a 95% confidence that the interval contains the true mean of the population. Looking at the confidence intervals, the majority of the confidence intervals are quite narrow around the sample means which indicates a high confidence that the sample means are close to the population means. The positive values are often higher than the negative values which indicate skewness of the distribution, and some intervals are extreme with over 20% higher values than the sample means. The observations are likely due to outliers, but the extreme ones require further statistical testing to validate the results.

P2: *Workload and configuration coverage. Should cover a representative sample*

space.

This principle is partially addressed in terms of covering workloads relevant to the focus of this thesis. Workload types were not the focus of this thesis, however, as explained in Section 4.2.4, there is no inherent order semantic for bursty workloads. Therefore, it was necessary to create another workload design with low granularity control of inter-arrival times of events. The principle further suggests using randomization of configurations to increase coverage of all possible combinations. However, this is not the focus of the study and is therefore not implemented.

P3: *Experimental setup description. Hardware and software setup should be described and the objective should be stated for each experiment.*

Chapter 4, thoroughly describes and motivates all the aspects of the research method used for the experiments. Hardware is not mentioned as much since this is the responsibility of Azure, but software from trigger configurations to workload designs and analysis scripts are all introduced.

P4: *Open access artifact. At least a representative subset of the results should be made publicly available.*

The source code of the benchmark and the data sets that Chapter 5 is based on is available on Github [51], and also included in Appendix A.

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

All results are presented in Chapter 5 as violin plots, bar plots, and cumulative distribution functions. The plots and CDF show a visual representation of the results for interpretation and comparison.

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

An attempt to provide an indication on statistical significance of the results was to use confidence intervals shown in Tables 6.1 and 6.2. No additional tests for significance were done.

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

All of the measurements clearly state the unit of measurement in the figures and tables.

P8: *Cost. The cost of running the experiment should be included.*

Azure provides a service called Cost Management + Billing which records expenditures from using Azure services. For an experiment run Azure billed $0.75 at the time of writing.

7
Conclusion

In this thesis, a benchmark has been implemented to gain insights into the latency and reliability in terms of out-of-order, missing, and duplicate event deliveries for Azure. The benchmark automatically deploys necessary resources to conduct experiments for specific Azure Function triggers and utilizes k6 for load testing to simulate scenarios relevant for studying latency and reliability. Further, Bash and Python scripts have been implemented for automatizing the execution of the benchmark, processing collected traces, and generating relevant plots to visualize the results.

RQ1: How does the choice of trigger types affect the latency of invoking functions?

There is no doubt that choosing an arbitrary trigger could lead to unexpected high latency since various trigger types have major differences in latency compared to each other. It is, therefore, crucial to analyze trigger types' latencies before choosing a trigger type if low latency is important within an application.

RQ1.1: How does the choice of runtime affect triggers' latency?

A runtime does affect a trigger's latency but is highly dependent on the density of incoming requests to the trigger. A runtime that has better latency performance compared to another runtime for a certain workload, might not be performing better with a different workload. However, the choice of runtime, in terms of latency, is not as substantial compared to the choice of which trigger type to use. The best performing runtime at light workloads is Node.js, while .NET performed better at heavy workloads.

RQ1.2: How does a bursty workload affect triggers' latency?

The latency is mainly affected by the heaviest workloads but is also affected to a certain degree for all workloads. For the baseline, one invocation per burst, the HTTP and Event Hub performed best, while the Blob Storage performed the worst. At the heaviest burst workloads, the Event Hub had the best performance followed by HTTP and Service Bus Topic that were considered second best due to their similar performances. The worst performing trigger at the heaviest burst workload was Blob Storage, with Cosmos DB not far behind.

RQ1.3: How does an inter-arrival time (IAT) controlled workload with a constant

flow of trigger invocations affect triggers' latency?

The time between each invocation has an effect on the majority of the trigger types, especially at very low IATs. Event Grid, Cosmos DB, HTTP, Queue Storage and Service Bus Topic did not change in their latency until the lowest IAT, where they increased significantly in latency. The best performing triggers at the baseline of the IAT of 250 ms, are the HTTP and Event Hub while the worst performing trigger is the Blob Storage. For the highest IAT workload of 1ms, the Event Hub had the lowest latency while the Blob Storage and Cosmos DB had the highest latencies.

RQ2: How reliable and consistent are the results of a function delivered?

The reliability of Azure trigger types can vary greatly depending on the configurations and the underlying implementation of each trigger.

RQ2.1: How frequent are events delivered OoO?

Depending on the trigger type and the delay between consecutive invocations, various levels of OoO event deliveries can be expected. Therefore, the ordering of event deliveries is critical, and there is no guarantee for trigger ordering, an invocation delay of more than 250ms is shown to ensure low frequencies of OoO for the trigger types studied in this thesis. Perhaps testing for even longer delays would have allowed a more precise conclusion of when no OoO will be encountered. The best performing trigger in terms of OoO is the Blob storage trigger. However, choosing the Blob storage trigger will sacrifice low latency triggering.

The reliability differences between runtimes are out of the scope of this thesis. However, the outcome of out-of-order reliability between Node.js and .NET for all the triggers indicates very similar results for both runtimes. Quantifying the differences and determining the significance could be future work.

RQ2.2: How frequent are events missed or delivered multiple times?

Missing event deliveries with regards to both the bursty and IAT controlled workloads are most apparent for Blob storage and Cosmos DB trigger in both runtimes. The HTTP trigger did not show any missing event deliveries for either workload since it is a synchronous trigger. The other trigger types have a few missing deliveries but nothing substantial to draw any conclusions. Duplicate event deliveries are not observed for Blob Storage, Queue Storage, and Event Hub trigger for both workloads and runtimes. Developers have the responsibility to implement defensive and idempotent event handlers to avoid duplicate deliveries. The results suggest that there might be a difference between runtimes for duplicate event deliveries depending on trigger type.

7.1 Visual summary of findings

In order for practitioners to get a better visualization of the findings, two color-coded tables were created. The colors in the tables represent intervals, to more easily find better or worse outcomes. The intervals have been chosen by common sense and do not have any weights or labels, more than a color.

Table 7.1 shows all findings for each trigger for three burst workloads: The baseline, a low bursty workload, and a high bursty workload. Table 7.2 shows the same, but instead for three workloads for the IAT: The baseline, an IAT with a long invocation delay, and an IAT with a short invocation delay.

There are four colors in the tables which have the following intervals:

- Green for latency: 0-100ms and Green for reliability: 0-1%.
- Yellow for latency: 100-1000ms and Yellow for reliability: 1-33%.
- Orange for latency: 1000-10000ms and Orange for reliability: 33-66%.
- Red for latency: Above 10000ms and Red for reliability: 66-100%.

Trigger type	Latency[1]			OoO			Missing executes			Duplicate executes		
	BL	LW	HW	BL	LW	HW	BL	LW	HW	BL	LW	HW
Blob Storage	5732ms	7267ms	12386ms	0%	3.4%	7.9%	0%	60.5%	67.1%	0%	0%	0%
Cosmos DB	82ms	428ms	4597ms	0%	64.5%	79.1%	0%	0%	62.8%	0%	10.1%	0%
Event Hub	35ms	48ms	49ms	0%	51.3%	62.3%	0%	0%	0%	0%	0%	0%
Event Grid	674ms	685ms	1255ms	0%	90.8%	93.7%	0%	0%	0%	0%	0%	0%
HTTP	36ms	94ms	250ms	0%	77.6%	83.9%	0%	0%	0%	0%	0%	0%
Service Bus Topic	56ms	66ms	97ms	0%	59.7%	77.8%	0%	0%	0%	0%	0%	0%
Queue Storage	220ms	590ms	1279ms	0%	82.5%	87.6%	0%	0%	0%	0%	0%	0%

Table 7.1: Overview of the findings for **bursty** workload. BL = Baseline, LW = Low workload (50 invocations), HW = High workload (300 invocations).

Trigger type	Latency[2]			OoO			Missing executes			Duplicate executes		
	BL	LW	HW	BL	LW	HW	BL	LW	HW	BL	LW	HW
Blob Storage	9193ms	32633ms	14324ms	0%	0.5%	3.9%	0%	1.2%	58.5%	0%	0%	0%
Cosmos DB	54ms	51ms	9588ms	0%	9.4%	85.9%	0%	0%	57.8%	0%	0%	0%
Event Hub	29ms	32ms	48ms	0%	7.5%	52.3%	0%	0%	0%	0%	0%	0%
Event Grid	601ms	521ms	1586ms	0%	68.5%	92.2%	0%	0%	0%	0%	0%	0%
HTTP	29ms	32ms	459ms	0%	0.4%	89.7%	0%	0%	0%	0.33%	0%	0.2%
Service Bus Topic	50ms	49ms	111ms	0%	16.3%	69.3%	0%	0%	0%	0%	0%	0%
Queue Storage	208ms	189ms	1836ms	0%	26.1%	83.0%	0%	0%	0%	0%	0%	0%

Table 7.2: Overview of the findings for **IAT** workload. BL = Baseline, LW = Low workload (50ms), HW = High workload (1ms).

7.2 Future Work

Based on the limitations of this benchmark, future work in studying latency and reliability of function triggers could be to conduct a similar study for other cloud

[1]The latency median is used in the table.
[2]The latency median is used in the table.

providers, such as AWS and Google Cloud. Even though the insights from this study are not transferable to other providers, the methodology is. Other extensions could be to investigate other runtimes such as Python and Java, and perhaps investigate other trigger types, e.g. triggers for third-party services (though these might have other requirements), Durable Functions, and using SDK to invoke functions. Further, it would be interesting also to investigate other metrics e.g. transfer rates and payload size, and the effect of these on latency and reliability.

In this thesis, almost all of the configurable settings for the triggers were kept at default values. The only time when these settings were adjusted was to make some of the triggers more comparable with the others, e.g. by adjusting batch sizes and polling delays. An interesting extension would be to investigate different configurations of the triggers and perhaps gain insights into how these could affect latency.

The current version of the benchmark only allows for two regions to deploy resources, namely, North Europe and East US. However, since Azure provides numerous regions all across the continents it could be relevant to examine the impact of the geographical location of the resources on latency to help practitioners choose suitable regions for their purposes.

As mentioned in Section 4.3, the experiments were run locally which increases the risks of the results being affected by network connectivity and transmission discrepancy. A future work could be to deploy the benchmark and conduct the experiments in a cloud VM.

7.2.1 Improvements to the Current Benchmark

With the experience gained from implementing this cloud benchmark and conducting experiments with it, there are suggestions for improvements to the benchmark.

Two parts of the benchmark execution that are significantly more time-consuming than other parts, mainly, conducting the load testing according to the workload designs, and post-processing of the received traces. The duration of the former mentioned is something that can not be optimized, unless the optimization is made to K6, or if the workloads are redesigned. The latter is performed using Pandas data frames e.g. manually correlating traces by switching operation Ids and replacing certain values in the cells of the data frame. Optimizations of the post-processing script would involve implementing more efficient ways of performing these operations and perhaps introducing parallel processing since Pandas only utilizes one core.

As mentioned in Section 4.4, to exclude cold starts the benchmark filters out the first invocation of each instance of an Azure VM. This will exclude any possibility of cold start invocations. However, this is not ideal because it does not check whether the first invocation is a cold start. An improvement would involve redesigning the current version of the benchmark, where all the triggers share the same Function App, to deploy a Function App for each trigger type. This approach will enable

systematic triggering of cold starts and removal of only cold start invocations.

References

[1] S. Eismann, J. Scheuner, E. Van Eyk, *et al.*, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, 2021.

[2] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[3] P. Sbarski and S. Kroonenburg, *Serverless architectures on Aws: with examples using Aws Lambda*. Simon and Schuster, 2017.

[4] Microsoft, *Triggers and bindings in azure functions*, `https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings`, May 29, 2022.

[5] D. Balla, M. Maliosz, and C. Simon, "Performance evaluation of asynchronous faas," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, IEEE, 2021, pp. 147–156.

[6] Microsoft, *Azure event grid delivery and retry - azure event grid*, `https://docs.microsoft.com/en-us/azure/event-grid/delivery-and-retry`, Jan. 13, 2022.

[7] J. Hollan, *Ordered queue processing in azure functions with sessions*, `https://dev.to/azure/ordered-queue-processing-in-azure-functions-4h6c`, May 30, 2019.

[8] A. Acharya and N. S. Sidnal, "High frequency trading with complex event processing," *2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW)*, pp. 39–42, 2016.

[9] L. Qian, Z. Luo, Y. Du, and L. Guo, "Cloud computing: An overview," in *IEEE international conference on cloud computing*, Springer, 2009, pp. 626–631.

[10] P. Mell, T. Grance, *et al.*, "The NIST definition of cloud computing," *Special Publication 800-145*, 2011.

[11] S.-J. Wiggers, *How to build a reactive solution with azure event grid?* `https://www.serverless360.com/blog/building-reactive-solution-with-azure-event-grid`, Jul. 2, 2022.

[12] Microsoft, *Understanding serverless cold start*, `https://azure.microsoft.com/sv-se/blog/understanding-serverless-cold-start/`, Feb. 7, 2018.

[13] I. C. Education, *Infrastructure as code*, `https://www.ibm.com/cloud/learn/infrastructure-as-code`, Dec. 2, 2019.

[14] Microsoft, *What is infrastructure as code?* `https://docs.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code`, Jun. 29, 2021.

[15] Pulumi, *Migrating to pulumi from terraform*, `https://www.pulumi.com/terraform/`.

[16] Microsoft, *What is distributed tracing?* `https://docs.microsoft.com/en-us/azure/azure-monitor/app/distributed-tracing`, Apr. 26, 2022.

[17] R. R. Sambasivan, R. Fonseca, I. Shafer, and G. R. Ganger, "So, you want to trace your distributed system? key design insights from years of practical experience," *Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-PDL*, vol. 14, 2014.

[18] E. Folkerts, A. Alexandrov, K. Sachs, A. Iosup, V. Markl, and C. Tosun, "Benchmarking in the cloud: What it should, can, and cannot be," in *Technology Conference on Performance Evaluation and Benchmarking*, Springer, 2012, pp. 173–188.

[19] Spec, *Standard performance evaluation corporation*, `https://www.spec.org/`.

[20] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao, "How to build a benchmark," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015, pp. 333–336.

[21] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," *Journal of Systems and Software*, vol. 170, p. 110 708, 2020.

[22] Q. Gao, W. Wang, G. Wu, X. Li, J. Wei, and H. Zhong, "Migrating load testing to the cloud: A case study," in *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, IEEE, 2013, pp. 429–434.

[23] K6, *Welcome to the k6 documentation*, `https://k6.io/docs/`.

[24] S. Eismann, J. Scheuner, E. Van Eyk, *et al.*, "Serverless applications: Why, when, and how?" *IEEE Software*, vol. 38, no. 1, pp. 32–39, 2020.

[25] V. Bobrovs, S. Spolitis, and G. Ivanovs, "Latency causes and reduction in optical metro networks," in *Optical Metro Networks and Short-Haul Systems VI*, International Society for Optics and Photonics, vol. 9008, 2014, p. 90080C.

[26] M. Bertilsson and O. Grönqvist, "Performance comparison of function-as-a-service triggers," 2021.

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

[28] J. Wen, Z. Chen, Y. Liu, *et al.*, "An empirical study on challenges of application development in serverless computing," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 416–428.

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

[30] A. V. Papadopoulos, L. Versluis, A. Bauer, *et al.*, "Methodological principles for reproducible performance evaluation in cloud computing," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1528–1543, 2019.

[31] Microsoft, *Azure functions reliable event processing*, `https://docs.microsoft.com/en-us/azure/azure-functions/functions-reliable-event-processing`, Nov. 5, 2020.

[32] I. Finta, G. Èliás, and J. Illés, "Packet loss and duplication handling in stream processing environment," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, IEEE, 2018, pp. 000 179–000 184.

[33] M. Li, M. Liu, L. Ding, E. A. Rundensteiner, and M. Mani, "Event stream processing with out-of-order data arrival," in *27th International Conference on Distributed Computing Systems Workshops (ICDCSW'07)*, IEEE, 2007, pp. 67–67.

[34] D. Jackson and G. Clynch, "An investigation of the impact of language runtime on the performance and cost of serverless functions," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, IEEE, 2018, pp. 154–160.

[35] W. Hasselbring, "Benchmarking as empirical standard in software engineering research," in *Evaluation and Assessment in Software Engineering*, EASE 2021, 2021, pp. 365–372.

[36] D. Ustiugov, T. Amariucai, and B. Grot, "Analyzing tail latency in serverless clouds with stellar," *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 51–62, 2021.

[37] Microsoft, *Azure blob storage trigger for azure functions*, `https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob-trigger`, Apr. 19, 2022.

[38] ——, *Azure cosmos db trigger for azure functions*, `https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-cosmosdb-v2-trigger`, Mar. 22, 2022.

[39] ——, *Azure event hub trigger for azure functions*, `https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-event-hubs-trigger`, Mar. 17, 2022.

[40] ——, *Azure event grid trigger for azure functions*, `https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-event-grid-trigger`, Mar. 9, 2022.

[41] ——, *Azure http trigger for azure functions*, `https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook-trigger`, Mar. 9, 2022.

[42] ——, *Azure service bus trigger for azure functions*, `https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-service-bus`, Apr. 19, 2022.

[43] ——, *Azure queue storage trigger for azure functions*, `https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-queue`, Apr. 19, 2022.

[44] ——, *Azure storage redundancy*, `https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy`. (visited on 05/22/2022).

[45] ——, *Azure functions pricing*, `https://azure.microsoft.com/en-us/pricing/details/functions/`.

[46] ——, *Azure event grid trigger for azure functions*, `https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-about`, Apr. 4, 2022.

[47] ——, *Managing event delivery with azure event grid*, https://docs.microsoft.com/en-us/archive/msdn-magazine/2018/september/azure-managing-event-delivery-with-azure-event-grid, Jan. 4, 2019.

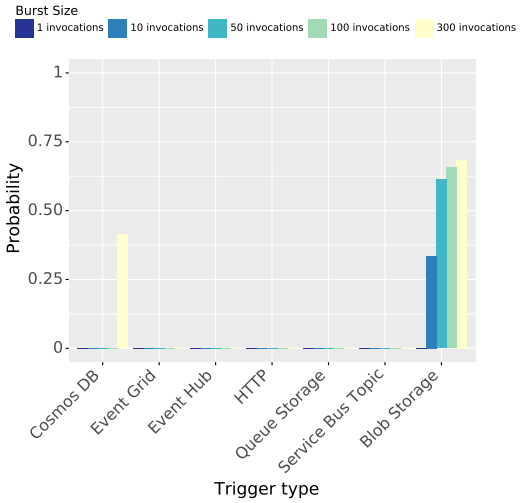
[48] A. Najafi, A. Tai, and M. Wei, "Systems research is running out of time," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 65–71.

[49] Microsoft, *Time sync for windows vms in azure*, https://docs.microsoft.com/en-us/azure/virtual-machines/windows/time-sync, Jan. 15, 2022.

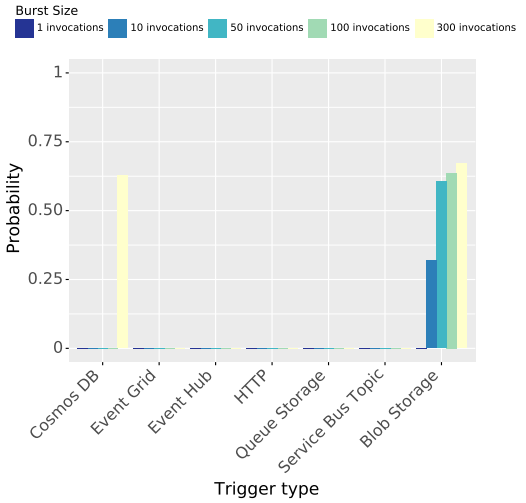
[50] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2015, pp. 1–12.

[51] L. Henrik and T. Henrik, *Azure-triggers-study*, https://github.com/henriklagergren/azure-trigger-benchmark.

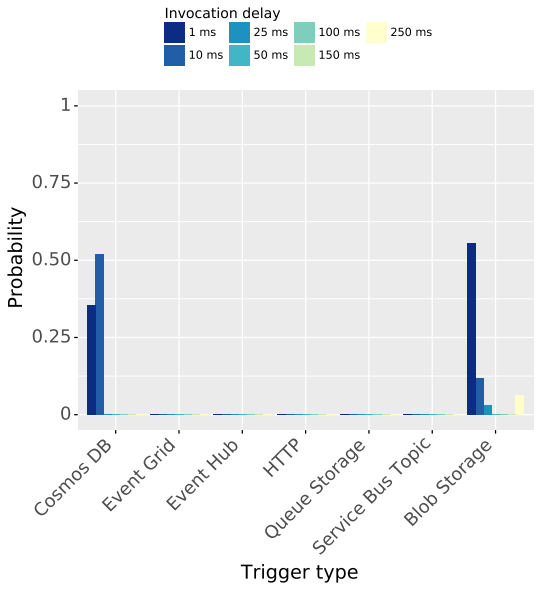
A
Appendix 1



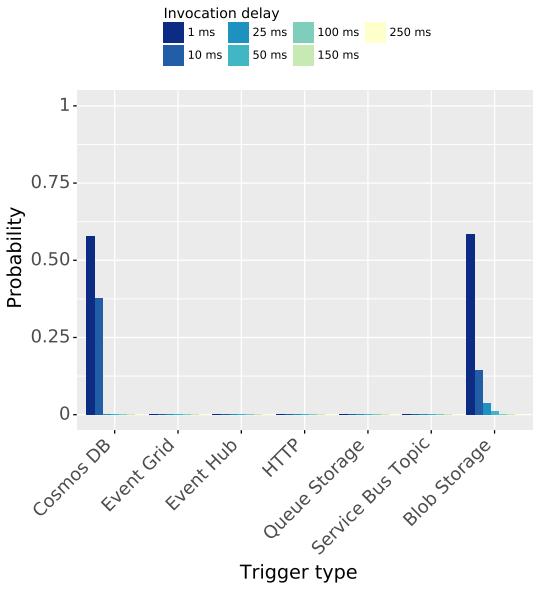
(a) All burst workloads in Node.js.



(b) All burst workloads in .NET.

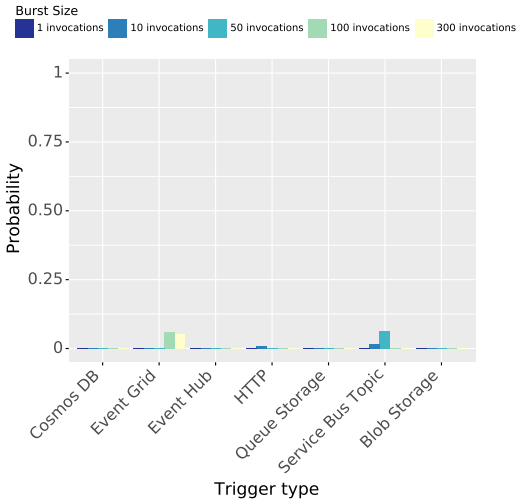


(c) All constant workloads in Node.js.

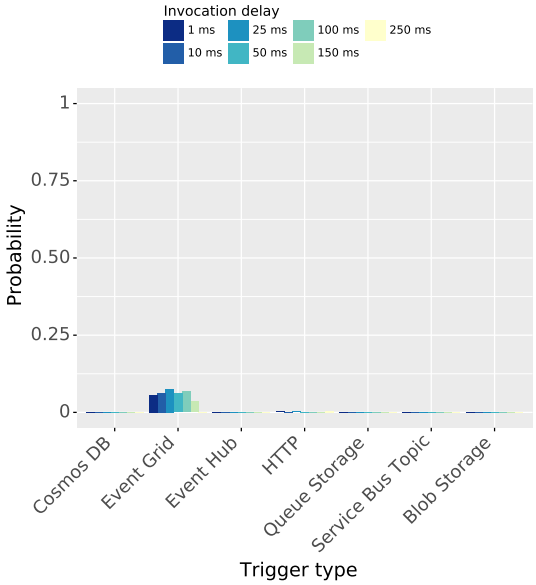


(d) All constant workloads in .NET.

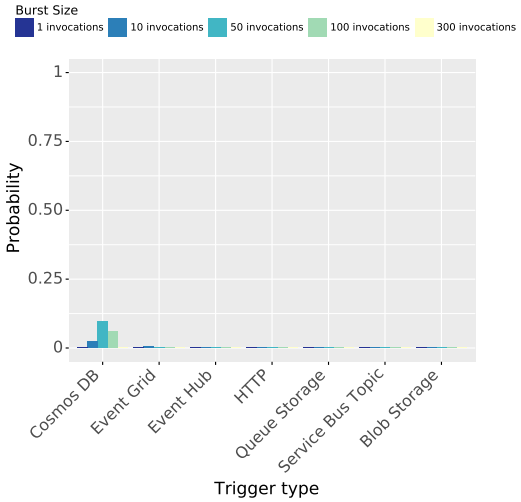
Figure A.1: Bar plots showing missing executes results for burst and constant workloads in both Node.js and .NET.



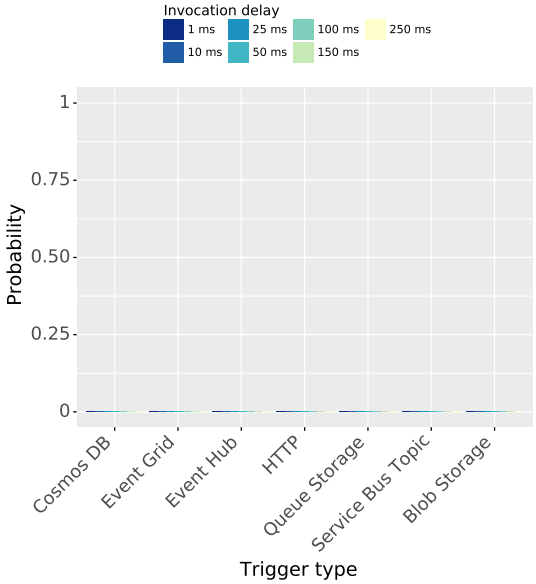
(a) All burst workloads in Node.js.



(b) All constant workloads in Node.js.



(c) All burst workloads in .NET.



(d) All constant workloads in .NET.

Figure A.2: Bar plots showing duplicate executes results for burst and constant workloads in both Node.js and .NET.

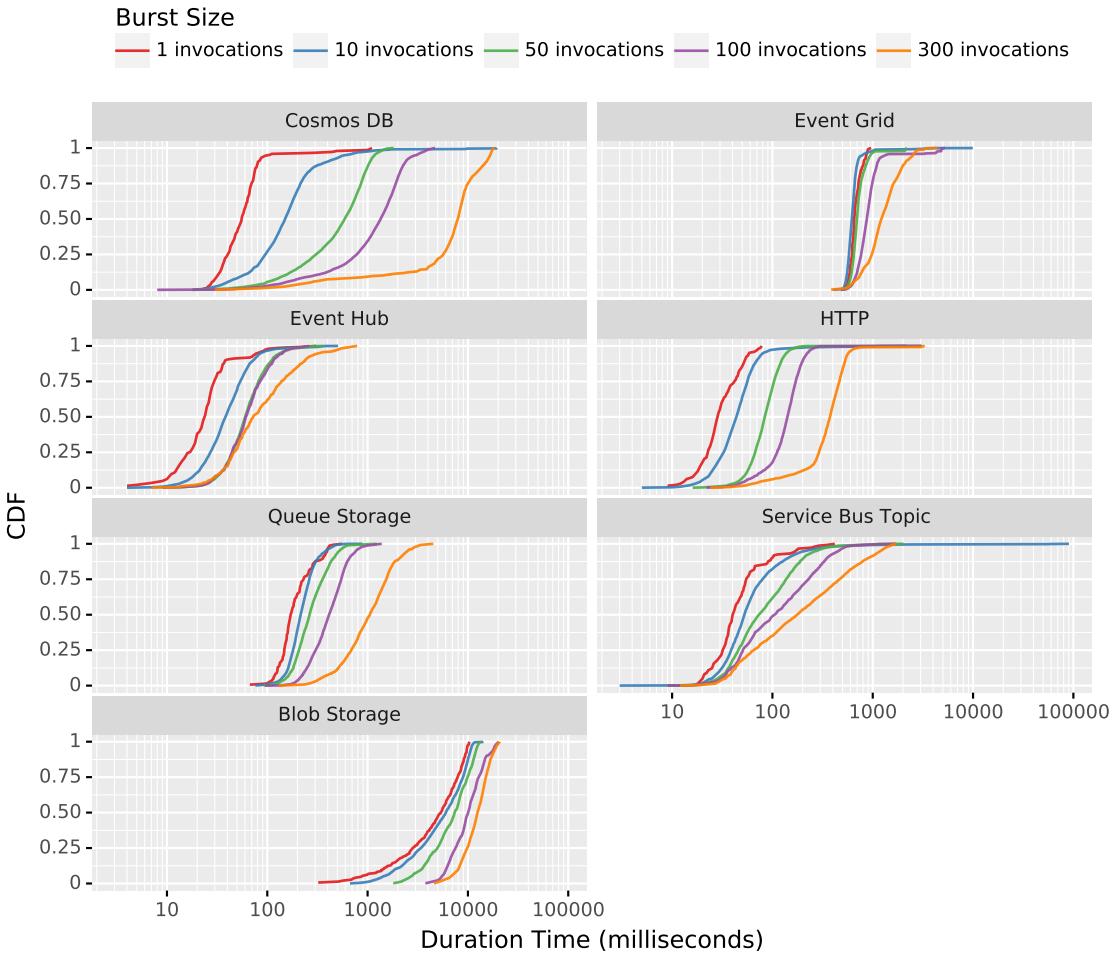


Figure A.3: CDF plots showing latency for all triggers in Node.js for different burst workloads.

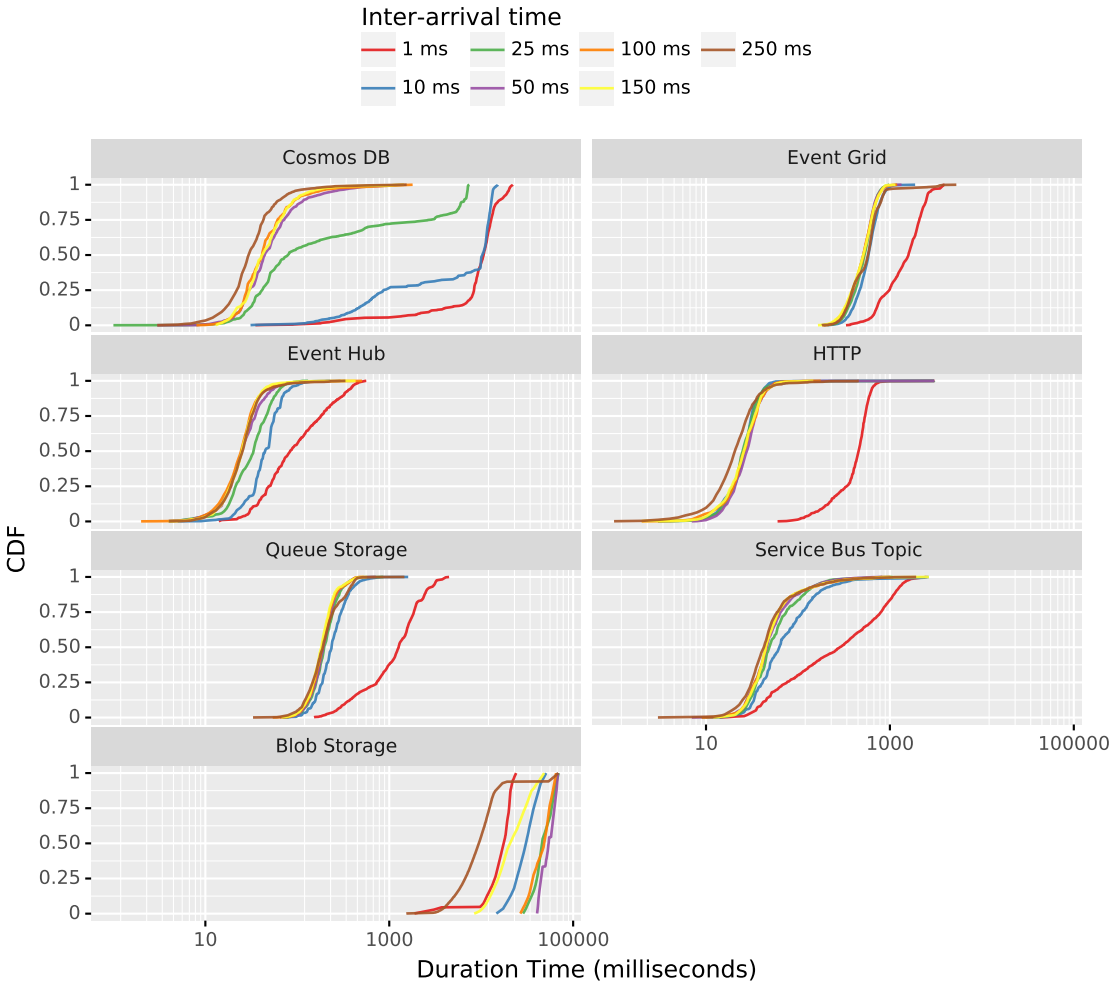


Figure A.4: CDF plots showing latency for all triggers in Node.js for different inter-arrival times.