FaaS Benchmarking over Orama Framework's Distributed Architecture

Leonardo Rebouças de Carvalho[a], Bruno Kamienski[b] and Aleteia Araujo[c]
Computer Science Department, University of Brasilia, Campus Darcy Ribeiro, Brasilia, Brazil

Keywords: Distributed Benchmark, Function-as-a-Service, Orama Framework.

Abstract: As the adoption of Function-as-a-Service-oriented solutions grows, interest in tools that enable the execution of benchmarks on these environments increases. The Orama framework stands out in this context by offering a highly configurable and scalable solution to aid in provisioning, running benchmarks and comparative and statistical analysis of results. In this work, a distributed architecture for the Orama framework is presented, through which it is possible to run benchmarks with high levels of concurrency, as well as with bursts of geographically dispersed requests, just as in real environments. The results of the experiment showed that the proposed architecture was able to divide the loads between the distributed workers and able to consolidate properly in the return of the results. In addition, it was possible to observe specific characteristics of the providers involved in the experiment, such as the excellent performance of Alibaba Function, whose average execution time was the lowest of the tests and free of failures. Google Cloud Function and AWS Lambda recorded intermediate results for average execution time and recorded failures. Finally, Azure Function had the worst results in average execution time and cold start.

1 INTRODUCTION

Worldwide end-user spending on public cloud services is forecast to grow 20.7% to total $591.8 billion in 2023, up from $490.3 billion in 2022 (Gartner, 2022). All this perspective of growth shows the solidity of this area of computing. Although the main cloud models, such as Infrastructure-as-a-Service (IaaS) (MELL and Grance, 2011) and Platform-as-a-Service (PaaS) still lead the focus of investment, other models that integrate the archetype of Everything-as-a-Service (XaaS) (DUAN et al., 2015) also indicate growth. In this context, serverless-based cloud computing models such as Function-as-a-Service (FaaS) (Schleier-Smith et al., 2021) have been predicted as the main programming paradigms of the next generation of the cloud.

In this scenario, there has been a growing interest in evaluations and benchmark tools that analyze the deliveries of providers, such as Sebs (Copik et al., 2021), PanOpticon (Somu et al., 2020), FaaSDom (Maissen et al., 2020), BeFaaS (Grambow et al., 2021) and Orama framework (Carvalho and Araujo, 2022). However, given the wide range of possibilities for adoptable strategies by providers, as well as the different regions in which these services can be implemented, this type of study needs to be increasingly dynamic and adjustable to real-world situations. Therefore, this work presents a distributed architecture for the Orama framework (Carvalho and Araujo, 2022) in which it is possible to run benchmarks in FaaS environments, exploring a wider range of scenarios than was possible in the standalone version. Through the master/workers architecture, it is possible to divide workloads between several instances, including geographically dispersed ones, allowing both the expansion of assessable levels of concurrency, as well as expanding the capacity to represent a reality of distributed demand.

In experiments, it was verified that the distributed architecture allows high levels of requests in comparison to the standalone version. In addition, it was found that the performance calculated between the scenario whose workers were in the same region and the scenario in which the workers were geographically spread maintained equivalent results, demonstrating that the separation of workers, in the proposed architecture, does not affect the analysis of the results. In addition, it was possible to observe behaviors intrinsic to the services, such as the superior performance demonstrated by Alibaba Cloud Func-

[a] https://orcid.org/0000-0001-7459-281X
[b] https://orcid.org/0000-0001-5817-8694
[c] https://orcid.org/0000-0003-4645-6700

Rebouças de Carvalho, L., Kamienski, B. and Araujo, A.
FaaS Benchmarking over Orama Framework's Distributed Architecture.
DOI: 10.5220/0011948400003488
In *Proceedings of the 13th International Conference on Cloud Computing and Services Science (CLOSER 2023)*, pages 67-77
ISBN: 978-989-758-650-7; ISSN: 2184-5042

tions. AWS Lambda and Google Cloud Function both had intermediate results. Azure services, on the other hand, obtained the worst results, presenting high average execution times and cold start.

This article is divided into six parts, the first being this introduction. Section 2 presents the theoretical foundation that supports the work. Section 3 describes the distributed implementation of the Orama framework. Section 4 presents the related works. Section 5 shows the results obtained and finally, Section 6 presents the conclusions and future work.

2 BACKGROUND

Thenceforward NIST (MELL and Grance, 2011) defined the traditional cloud models in 2011 as IaaS, PaaS and SaaS, the main public cloud providers began to name their services with another set of acronyms, which culminated in the emergence of the term “XaaS” to define Everything-as-a-Service (DUAN et al., 2015). In 2014 AWS launched Lambda, which then inaugurated the Function-as-a-Service (FaaS) model (Motta. et al., 2022). In the FaaS paradigm, the customer delivers a piece of code of interest to the provider, generally written in some language supported by the provider. In addition, the client must configure a trigger to activate the function and this trigger can happen from other services that make up the provider’s platform or through a REST API.

Since FaaS is designed to run state-independently, some restrictions are imposed on customers, such as limits on allocable vCPUs, RAM and maximum execution time. Another aspect that significantly distinguishes FaaS from other cloud service models is the billing method. Instead of being charged based on the execution time of instances, as in the IaaS model, in FaaS the customer will only pay based on the activation of the service and its respective duration.

Major public cloud providers have FaaS offerings. In addition to AWS Lambda, which is the forerunner of this concept, it is possible to find solutions from Azure, GCP, Alibaba Cloud, among others. Azure Function (AZF) is Azure’s entry into this slice of the cloud market. Google Cloud Function (GCF) is the name of GCP’s FaaS. Alibaba Cloud offers Alibaba Function Compute (AFC) as its FaaS. Other providers such as Oracle and IBM also offer FaaS options.

Faced with so many FaaS options, it is essential to understand how the strategies adopted by providers deliver environments. Considering the large number of regions in which the main providers are installed, it is possible that implementation differences between providers or between regions can meet different needs or may even be impeding, unfeasible or expensive. Since these strategies are the provider’s big trade secrets, the best way to elicit these strategies is by running benchmarks. As these environments are highly dynamic and maintain constant evolution, it is important that the solutions that promote benchmarks on these platforms are configurable to the point of better representing the problems found in real environments. The development of solutions aimed at the current multicloud context should favor the incorporation of new providers and their services as they enter this market. Because of this, it is essential to use a cloud orchestration solution.

2.1 Infrastructure Tools

Terraform (HashiCorp, 2021) is a cloud orchestrator that helps solutions integrate with different virtualization and automation platforms, especially in cloud environments. Through Terraform it is possible to create lightweight and portable infrastructure definition artifacts that can be easily incorporated into other solutions. Other cloud orchestration solutions, such as Heat (Gupta et al., 2014) and CloudFormation (Wittig and Wittig, 2018) propose similar approaches to Terraform, however, as evaluated in (Carvalho and Araujo, 2020) Terraform presents better results.

Performance testing (Erinle, 2013) is a type of testing intended to determine the responsiveness, reliability, throughput, interoperability, and scalability of a system and/or application under a given workload. It could also be defined as a process of determining the speed or effectiveness of a computer, network, software application, or device. Testing can be conducted on software applications, system resources, targeted application components, databases, and a whole lot more. It normally involves an automated test suite, such as JMeter (Erinle, 2013), as this allows for easy, repeatable simulations of a variety of normal, peak, and exceptional load conditions. Such forms of testing help verify whether a system or application meets the specifications claimed by its vendor. Technology solutions currently need to deal with large and fast flows of information and any instability in the service can lead to loss of information. Because of this, it is important to use queuing mechanisms in order to guarantee the correct processing of requests.

Apache Kafka (Sax, 2018) is a scalable, fault-tolerant, and highly available distributed streaming platform that can be used to store and process data streams. The Kafka cluster stores data streams, which are sequences of messages/events continuously produced by applications and sequentially and incrementally consumed by other applications. The Connect

API is used to ingest data into Kafka and export data streams to external systems such as distributed file systems, databases, and others. For data stream processing, the Streams API allows developers to specify sophisticated stream processing pipelines that read input streams from the Kafka cluster and write results back to Kafka. Apache Kafka is a solution that adheres to the microservices paradigm, that is, the development model in which the solution's complexity blocks are segmented into smaller processes. In order to manage this large amount of services, without the need to deal with several virtual machines, it is possible to use container-oriented environments.

Docker (Ibrahim et al., 2021) enables the containerization of a software package along with associated configuration and setup details. Such containers can be easily and rapidly deployed while avoiding compatibility issues. In fact, a recent study reports that Docker can speed up the deployment of software components by 10-15 folds. Much of today's applications are multi-component (i.e., multi-container) applications. For instance, a simple web application would require a web server and a database component. Docker Compose (Ibrahim et al., 2021), a natural progression of Docker, enables practitioners to compose such complex applications. Applications transcribe such compositions in a Docker Compose file, where components are specified by describing their Docker image and associated configuration as well as the relations between components. The various services of a solution supported by a container environment can generate large volumes of data that needed processing techniques to make any sense.

2.2 Statistical Analysis Tools

Statistical analysis of benchmark results allows the observation of several phenomena. The factorial design (Jain, 1991), for example, helps to identify the effect of mapped factors on the results. With the factorial design, it is possible to identify whether the variation of any factor in a given scenario causes (or does not) any statistically significant impact on the results. It is also possible to identify the existence of factors that were not initially mapped and that may have a significant influence on the results. Another important statistical analysis tool for understanding the results is the paired t-test. In this test, the difference between two results is evaluated in order to determine its statistical significance, as well as the respective degree of confidence. This test allows the user to establish whether the difference between two results can be considered relevant or insignificant, in the second case the results can be considered statistically equal.

With the purpose of implementing a specific benchmark solution for a FaaS environment that adheres to different scenarios as close to real ones as possible, this work uses tools such as Terraform, JMeter, Apache Kafka, Docker, factorial design, and t-test to propose a distributed architecture for Orama framework, whose detailed description will be presented in the next section.

3 DISTRIBUTED ORAMA FRAMEWORK

The Orama framework (Carvalho and Araujo, 2022) is a tool developed to conduct benchmarks on FaaS environments. Although it is possible to run benchmarks on other types of environment, its focus is directed towards the evaluation of cloud environments oriented to the FaaS paradigm. Through the Orama framework, it is possible to provision environments in an automated way, thanks to its integration with Terraform, which makes the incorporation of new providers as simple as building a Terraform infrastructure definition artifact. Once provisioned, the environment can also be de-provisioned using Terraform automations through the Orama framework. In addition, the entire process of running the benchmarks is conducted by the framework based on the settings entered in the system. It is possible to create several test scenarios varying the level of concurrency, the number of repetitions of a scenario, the establishment of intervals between tests and the execution of warm-up requests, whose objective is to observe the cold start phenomenon, which is very common, and impactful in FaaS environments.

The Orama framework comes with some pre-configured use cases that can be deployed to major providers without any user intervention. These use cases range from a simple calculator, whose objective is only to validate the implementation and correct operation of the service, as well as use cases that use other services from providers, such as object storage and databases. The use cases accept parameterizations that allow the provisioning of different configurations for the use cases, from the amount of allocable memory to the region of the provider where the use case must be implemented.

The standalone version of the framework presented at (Carvalho and Araujo, 2022) has all of its components implemented in a container environment, including the "benchmarker" which is the component responsible for triggering bursts of requests that simulate real demands on the environments. It turns out that using this approach, the level of concurrency that

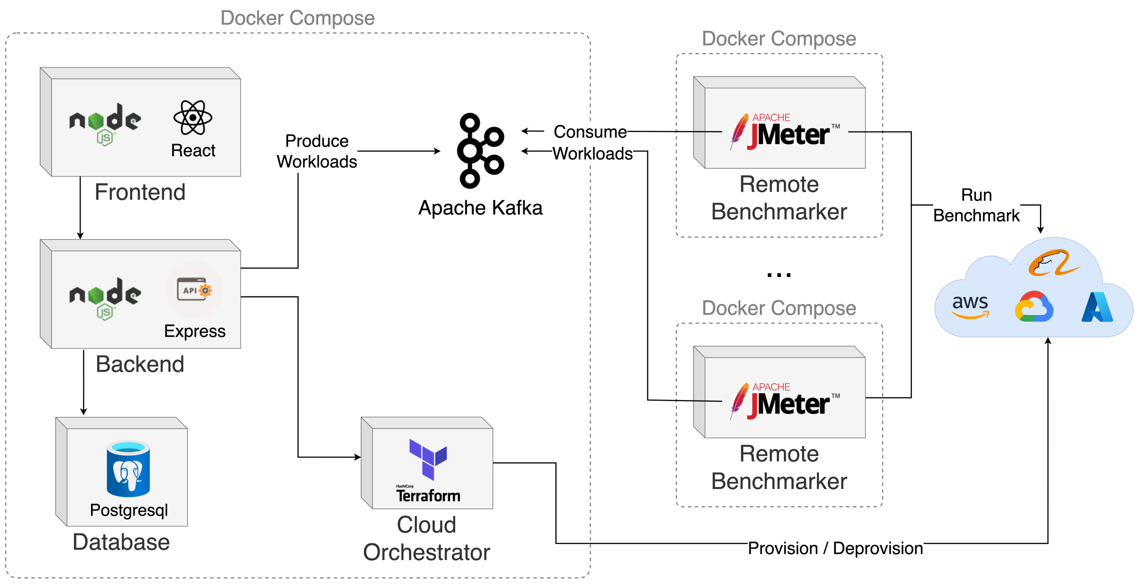


Figure 1: Distributed Orama framework architecture.

the Orama framework can simulate on the environment is limited to the amount of resources available on the machine where it is installed. Furthermore, test requests compete with framework management traffic, as framework components also communicate via HTTP.

In this work, a distributed architecture for the Orama framework is presented, as shown in Figure 1. In this architecture, it is observed that the benchmarker component is deployed outside the main environment and therefore its installation can occur in another instance. Furthermore, the number of instances of the benchmarker is variable and can be adjusted to the characteristics of the test to be performed. It is possible, for example, to concentrate the benchmarker's workers in the same region or to distribute them among several regions. It is also possible to include a number of workers, whose load distribution of requests prevents the saturation of resources on the machine, preventing the occurrence of faults attributed to the test execution worker.

Communication between the remote benchmarkers (workers) and the main Orama framework environment (master) composed of frontend, backend, database, and orchestrator is done through Kafka, as can be seen in Figure 1. There are three types of "topics" that are managed by Kafka. The "Health check" topic allows remote workers to inform the master that they are able to receive triggers to run benchmarks. Once health is up to date, this worker will be considered in the distribution of loads of a respective benchmark and then a topic with the "uuid" of the respective worker will be included in Kafka by the backend of

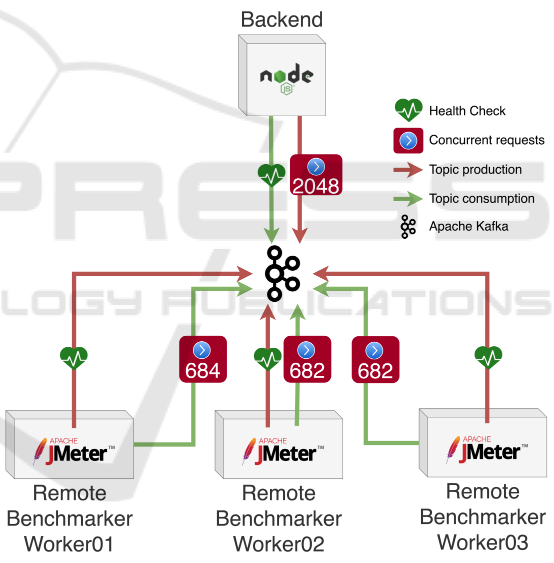


Figure 2: Kafka workflow in Orama framework.

the Orama framework, to be consumed by the respective worker and executed, as can be seen in Figure 2. After running the benchmark, the remote worker records the partial result and inserts it into a third Apache Kafka topic, for consumption by the master's backend. It is worth noting that the Orama framework implements a balance between the number of requests requested for each worker to execute on the target environment.

Figure 2 shows a scenario with three workers and a request to run a benchmark with 2048 concurrent requests. It is possible to notice that all workers are asked for 682 requests, which adds up to 2046 requests. This leaves 2 requests that are assigned to one of the workers, in the case shown in Figure 2 the "Remote Benchmarker Worker 01". As soon as the backend perceives the receipt of all partial results, it promotes the consolidation of the results. If these results do not appear within a configured timeout, then the respective partial result is assigned a failure result.

Once the results are consolidated, it is possible to create factorial designs combining two benchmarks and two levels of concurrence. If the benchmarks involve two different providers, it will be possible to observe the influence of the providers on the results, as well as the difference from the concurrence. Using the factorial design as established by the Orama framework, it is possible to identify whether the strategies adopted by the providers impact the results and whether the level of concurrence is the factor whose influence prevails. It is still possible to identify the existence of some other factors apart from the provider and the level of concurrence influencing the results, in this case the portion of the influence related to the statistical error will be significantly high.

The distributed architecture of the Orama framework presented in this work opens up a wide range of possibilities for running benchmarks on FaaS environments. At this moment of leveraging this approach, it is essential to have a tool available that allows the evaluation of environments delivered by providers, including high levels of concurrence and distributed customers, whose characteristics are more similar to critical situations in the real world experienced by solutions supported by FaaS environments.

In the next section, the results of an experiment using the aforementioned architecture will be presented. The capabilities of the Orama framework using a distributed approach and the insights obtained from the results of the experiment will be illustrated.

4 RELATED WORKS

This article presents a distributed architecture for executing benchmarks in FaaS environments over the Orama framework. The related work is discussed from the perspective of benchmarking FaaS platforms in general as well as work on serverless benchmark frameworks. A comparison between related works is presented in Table 1.

In the paper (Back and Andrikopoulos, 2018) the authors used a microbenchmark in order to investigate two aspects of the FaaS: the differences in observable behavior with respect to the computer/memory relation of each FaaS implementation by the providers, and the complex pricing models currently in use. They used AWS, IBM, GCP, Azure, and OpenWhisk in their evaluation. However, the authors did not present an evaluation of the performance of their microbenchmark in different regions of the providers, nor with different levels of concurrence, as presented in this work.

The quality impacts of operational tasks in FaaS platforms as a foundation for a new generation of emerging serverless big data processing frameworks and platforms are evaluated in (Kuhlenkamp et al., 2019). The authors presented SIEM, a new evaluation method to understand and mitigate the quality impacts of operational tasks. They instantiated SIEM to evaluate deployment package and function configuration changes for four major FaaS providers (AWS, IBM, GCP, and Azure), but only in European regions for the same level of concurrence. In this work, on the other hand, several levels of concurrence are evaluated using two different worker approach (concentrated and distributed).

PanOpticon (Somu et al., 2020) provides a comprehensive set of features to deploy end-user business logic across platforms at different resource configurations for fast evaluation of their performance. The authors conducted a set of experiments testing separate features in isolation. An experiment comprising a chat server application was conducted to test the effectiveness of the tool in complex logic scenarios in AWS and GCP. Furthermore, in this work, the range of tests that the Orama framework can evaluate was extended beyond the execution of benchmarks on AWS and GCP, to include the execution of benchmarks on Azure and Alibaba, which are two other important players in this market.

FaaS-dom (Maissen et al., 2020) is a modular set of benchmarks for measuring serverless computing that covers the major FaaS providers and contains FaaS workloads (AWS, Azure, Google, and IBM). A strong element of FaaS-dom's functions is that they were created in a variety of languages and for a variety of providers. However, the operations that the FaaS-dom functions carry out can be viewed as basic, and they lack Orama's approach to integration with other cloud services.

BeFaaS (Grambow et al., 2021) offers a benchmark methodology for FaaS settings that is application centric and focuses on evaluating FaaS apps using real-world and prevalent use cases. It offers enhanced result analysis and federated benchmark testing, where the benchmark application is split across several providers. It does not, however, provide a superior approach to statistical analysis, such as the factorial design or t-test that are covered by this study.

In paper (Barcelona-Pons and García-López,

Table 1: Related works.

Paper	**Providers**	**Configu-rable**	**Factorial Design**	**T-test**	**Distri-buted**
(Back and Andrikopoulos, 2018)	AWS, IBM, GCP, Azure, and OpenWhisk	No	No	No	No
(Kuhlenkamp et al., 2019)	AWS, IBM, GCP, and Azure	No	No	No	No
(Somu et al., 2020)	AWS and GCP	No	No	No	No
(Grambow et al., 2021)	AWS , GCP, Azure, TinyFaaS, OpenFaaS, and OpenWhisk	Yes	No	No	No
(Barcelona-Pons and García-López, 2021)	AWS, IBM, GCP, and Azure	No	No	No	No
(Copik et al., 2021)	AWS, Azure, and GCP	No	No	No	No
(Wen et al., 2020)	AWS, Azure, GCP, and Alibaba	No	No	No	No
(Carvalho and Araujo, 2022)	AWS and GCP	Yes	Yes	Yes	No
This paper	**AWS, Azure, GCP, and Alibaba**	**Yes**	**Yes**	**Yes**	**Yes**

2021) the authors analyzed the architectures of four major FaaS platforms: AWS Lambda, AZF, GCP, and IBM Cloud Functions. The research focused on the capabilities and limitations the services offer for highly parallel computations. The design of the platforms revealed two important traits influencing their performance: virtualization technology and scheduling approach. This work, on the other hand, focuses on investigating the differences in performance of the main providers with different levels of concurrence.

In the Serverless Benchmark Suite (Sebs) (Copik et al., 2021), typical workloads are chosen, the implementation is tracked, and the infrastructure is assessed. The benchmark's applicability to several commercial vendors, including AWS, Azure, and Google Cloud, is guaranteed by the abstract concept of a FaaS implementation. Based on the executed test cases, this work assesses variables including time, CPU, memory, I/O, code size, and cost. However, unlike the Orama framework used in this work, their solution can't work in distributed mode.

In (Wen et al., 2020), the authors run a test flow employing micro benchmarks (CPU, memory, I/O, and network) and macro benchmarks to evaluate FaaS services from AWS, Azure, GCP, and Alibaba in detail (multimedia, map-reduce and machine learning). The tests made use of specific Java, Node.js, and Python methods that investigated the benchmarking attributes to gauge resource usage efficiency and initialization delay. However, they do not present evaluations in different levels of concurrence and also do not perform a statistical analysis using factorial design and t-test as is done in this work.

Although the aforementioned work presents an ad-hoc evaluation of the providers, the scalability and manageability of this evaluation is limited to the parameters that were used in the work. On the other hand, this article uses a fully manageable solution prepared for the incorporation of new providers and use cases.

5 RESULTS

The Orama framework makes it possible to run benchmarks on a FaaS environment in different scenarios, especially with the introduction of the distributed architecture presented in this work. In order to explore some possibilities of running benchmarks, an experiment was designed to allow the behavior of the architecture to be observed, as well as obtaining some insights into FaaS environments. Details of how the experiment was conducted will be presented in Section 5.1, while an analysis of the results is provided in Section 5.2.

5.1 Methodology

Considering that the main objective of this experiment is to analyze the behavior of the distributed architecture of the Orama framework, a use case was selected which had the role objective of guaranteeing the correct execution of FaaS in the providers, without integration with other services, such as Database or Object Storage, which could introduce specific features of these services and divert the focus from the architecture itself. Therefore, the use case that deploys a simple math calculator in each of the four main FaaS providers supported by the Orama framework was selected. The providers are AWS, Azure, Google and

Table 2: Average execution times (in milliseconds).
(C) → concentrated workers — (D) → distributed workers.

Concurrence level	**Scenario**							
	AWS Lambda		**GCF**		**AZF**		**AFC**	
	(C)	**(D)**	**(C)**	**(D)**	**(C)**	**(D)**	**(C)**	**(D)**
1	581	793	592	684	643	954	**102**	281
2	595	749	605	757	612	910	**93**	360
4	663	776	699	769	715	1004	**91**	302
8	592	859	595	832	622	1149	**133**	511
16	637	897	638	817	695	1145	**100**	358
32	709	983	707	918	890	1414	**100**	367
64	911	1158	924	1097	1303	1732	**109**	378
128	1328	1569	1367	1548	2264	2390	**128**	393
256	2274	2417	2386	2466	4130	3749	**233**	565
512	4068	4418	4447	4489	7506	6648	**168**	426
1024	6852	7334	5181	5787	11514	11884	**116**	367
2048	6139	6072	3596	3270	18378	17234	**67**	338

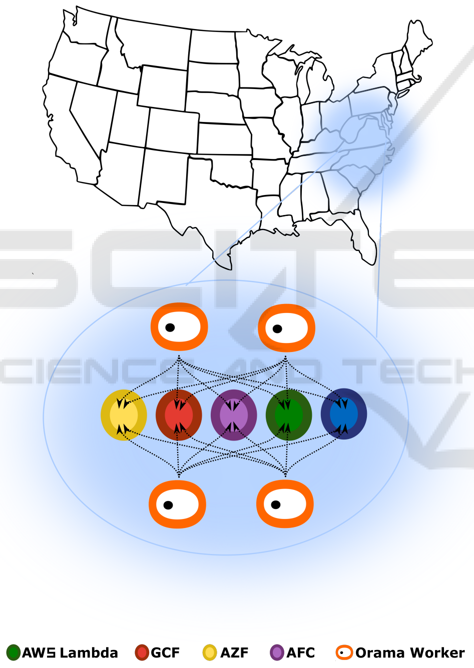


Figure 3: Concentrated workers scenario.

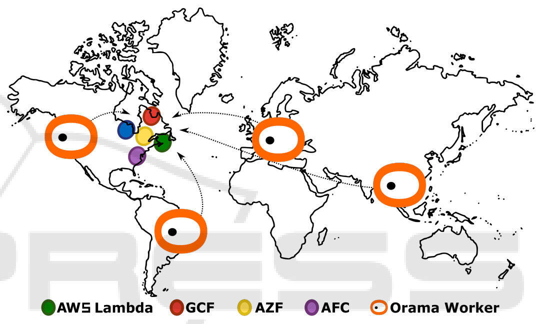


Figure 4: Distributed workers scenario.

Alibaba.

The distributed architecture of the Orama framework allows workers in different regions to be deployed. Therefore, in this experiment two scenarios were elaborated. In the first scenario, shown in Figure 3, the workers were all deployed in the same region where the target FaaS environments were also provisioned (concentrated workers scenarios). In the second scenario, workers were distributed in geographically distant regions, as shown in Figure 4. With these two scenarios, it was expected to demonstrate the influence of latency on FaaS, however, their behavior should be equivalent and demonstrate the same strategy applied by providers.

Both scenarios were subjected to concurrent loads of requests from just one simultaneous request to up to 2048 requests with exponential growth. Thus, the FaaS of each provider were submitted individually to loads of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, and 2048. It is noteworthy that the tests performed with the Orama standalone architecture framework were limited to up to 512 simultaneous requests, noting the limitations of opening a connection with FaaS services imposed by the instance in which the Orama framework was installed. As 2048 was the maximum level of this experiment, a quantity of 4 benchmarker workers was defined so that each one was responsible for carrying out a maximum of 512 concurrent requests. Furthermore, each battery of tests with different levels of concurrency was repeated 10 times in order to establish a statistical data mass for further analysis.

In the first scenario (concentrated workers), the Orama framework was deployed on an instance of GCP's Compute Engine in the US-East region with 16GB of RAM and 4 vCPUS, and each of the four workers was also deployed in the same GCP region, the servers had 4GB of RAM and 2 vCPUS each. For the second scenario, the master installation of Orama framework was kept on the same instance in the US-East GCP region as used in the first scenario. However, the workers were spread out in other GCP regions such as: asia-northeast1 (Tokyo, Japan), US -West (The Dalles, Oregon, United States), europe-west2 (London, England), and southamerica-east1 (Osasco, Brazil). All instances of GCP used in this experiment ran Debian 11 as the operating system. The FaaS that were the target of this experiment in both scenarios were configured to be deployed in the respective East American regions at each of the respective providers, namely AWS Lambda, AZF, GCF and AFC.

At the end of the execution of the repetitions of both test scenarios, factorial designs and t-tests were created in order to analyze the effects of the difference between the providers and the difference between the lowest level of concurrency (only one request) and the highest level, with 2048 concurrent requests. The analysis of the obtained results is presented in the next section.

5.2 Outputs Analysis

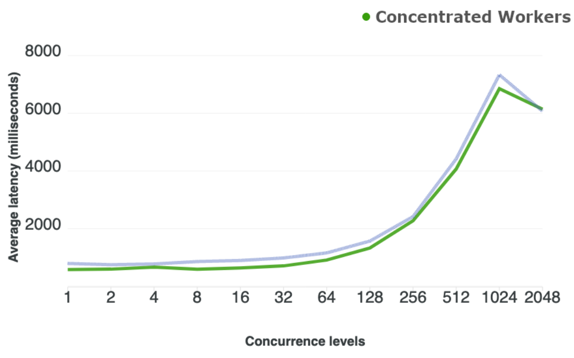
Table 2 presents the consolidated result of all eight test scenarios conducted in the experiment. It is possible to observe that the provider whose scenarios had the lowest average execution times was AFC, highlighted in bold. Next in very close range are the average execution times for AWS Lambda and GCP. Finally, the average execution times for AZF were the highest in this experiment. In addition, in Table 2, it is also possible to observe that the scenarios with distributed workers have generally higher averages than the averages found in the scenario with workers concentrated in the same region, this is the expected result, since the greater geographic distance between workers and target FaaS should raise averages by introducing higher latency to traverse the network.

Figure 5a presents a comparative result between the scenarios involving AWS Lambda. It may be seen that the average execution time of both scenarios in AWS Lambda follow the growth of the concurrency level the previous level to the maximum (2048 concurrent requests) when there is a small drop in the average. This decrease in the average execution time after 1024 indicates that, when faced with a growing demand, the provider reinforced the service infrastructure in order to maintain its quality in terms of execution time. Despite this effort by the provider, failures occurred from 256 concurrent requests, as shown in Figure 5b, which shows the percentage failure rates that occurred at each concurrency level. To corroborate the indication that the provider promoted an escalation before 2048 requests, as well as the average time, the failure rate also showed a reduction.

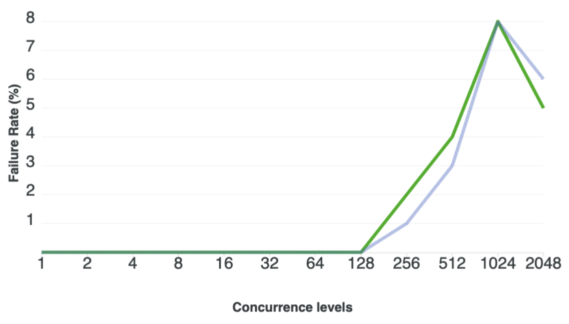
Figures 5c and 5d show the average execution times of the scenarios involving the GCF and their respective failure rates at each concurrency level, respectively. It is possible to observe that the graph of the average execution time of the GCF is similar to the same graph for AWS Lambda, in which the average time accompanies the growth of the concurrency level until the intervention of the provider causes a decrease in the average time. However, GCF's average time threshold is lower than AWS Lambda's. While in AWS Lambda the peak point is 7.3 seconds (at 1024 in the distributed scenario), in CGF the peak is recorded at 5.7 seconds (at 1024 in the distributed scenario). This proximity between AWS Lambda and GCF average times demonstrates a strategic equivalence between providers. Despite this, with regard to the failure rate, the occurrence of failures in GCF started only after 1024 requests, while AWS Lambda already had failures at 256. However, the level of failures recorded by GCF was higher compared to AWS Lambda, because while in AWS Lambda the failures peaked at 8% of requests, in GCF these failures reached around 20%. Another difference in the AWS Lambda and GCF results is the continuous growth of failures demonstrated in the GCF distributed scenario, which indicates that the provider's monitoring layer had greater difficulty in dealing with requests from different regions than those whose origin was the same.

Unlike AWS Lambda and GCF, which showed a point of reduction in the average execution time, AZF, as shown in Figure 5e, maintained a continuous growth in the average execution time, registering the highest average times in the experiment. This indicates that in the AZF environment there was no reinforcement of the infrastructure as demand grew, although this is a FaaS premise. Despite the high average time, AZF recorded a low failure rate, with failures occurring in only 3% of the 2048 concurrent requests in the distributed scenario, as can be seen in Figure 5f.

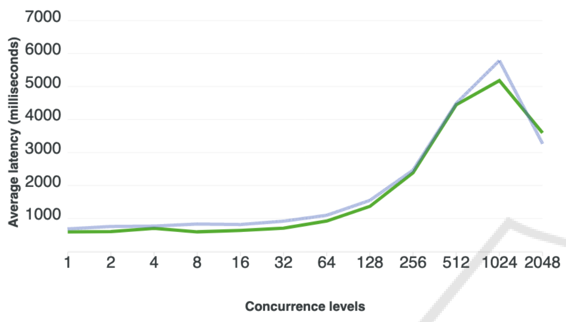
Figure 5g shows the average AFC execution times. It is possible to observe three points where the provider seems to have reinforced the infrastructure, such as from 2 to 4 requests, from 8 to 16, and from 256 to 512. This meant that the provider maintained the lowest average times of the experiment and did not present an increasing curve of the average time, as occurred with the other providers. In addition, there



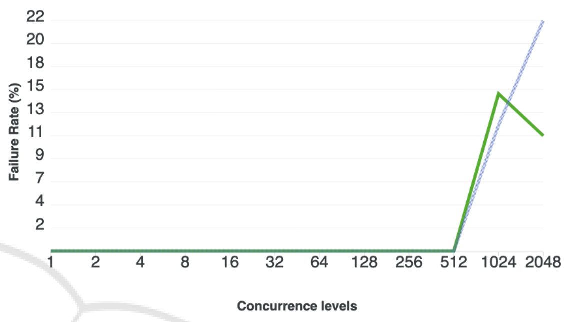
(a) AWS Lambda execution times.



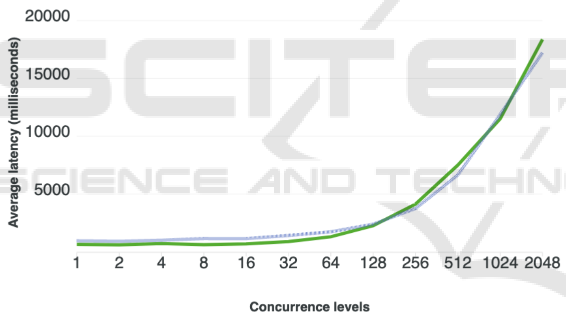
(b) AWS Lambda failure rates.



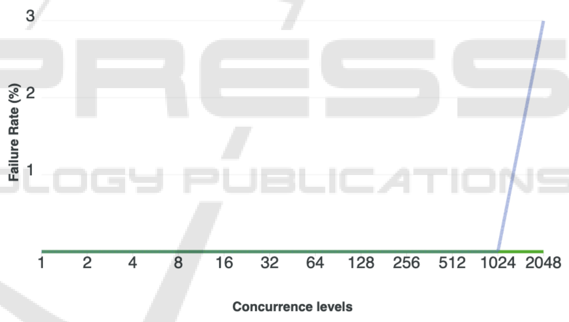
(c) GCF execution times.



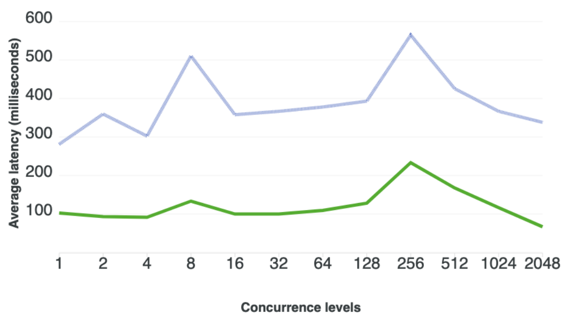
(d) GCF failure rates.



(e) AZF execution times.



(f) AZF failure rates.



(g) AFC execution times.

Figure 5: Comparative results.

was no occurrence of failure during the ten repetitions of the test batteries. The AFC result also shows the expected result of the experiment, in which it is possible to observe the same behavior both in the scenario with workers concentrated in the same region, and with distributed workers, differing only in the average level.

Another interesting aspect to be analyzed in this

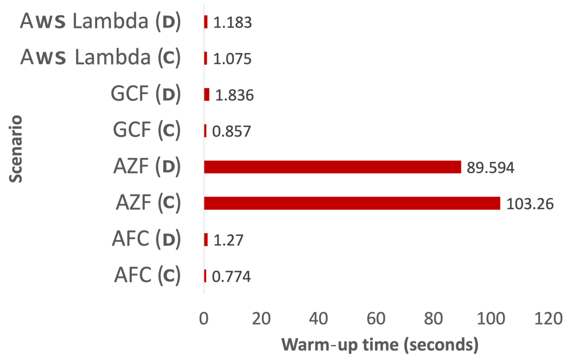


Figure 6: Warm-up times. (C) → concentrated workers — (D) → distributed workers.

experiment is the cold start phenomenon, that is, the time it takes the provider to put its FaaS into operation for the first time or after a certain period of inactivity. Figure 6 provides the warm-up times for all eight scenarios. This is the first request made before the start of the battery of tests. The arcing times showed a big difference between AZF and the other providers. While AWS Lambda, GCF and AFC recorded warm-up times around 1s, AZF took 89 seconds in the distributed scenario and 103 seconds in the concentrated scenario. This difference shows that the AZF deployment strategy is considerably different from the others and this can significantly impact the performance of applications supported by this service, since after some downtime, AZF FaaS will have a much higher response time than usual and this will certainly negatively impact the user experience.

In order to understand the impact of the provider and concurrence factors on the results, six factorial designs were elaborated comparing the four providers with each other and the minimum and maximum concurrence levels (1 and 2048), as shown in Figure 7. In Figures 7a, 7b and 7d, it is possible to observe the predominance of the concurrence factor, to the detriment of the provider factor, which indicates that in

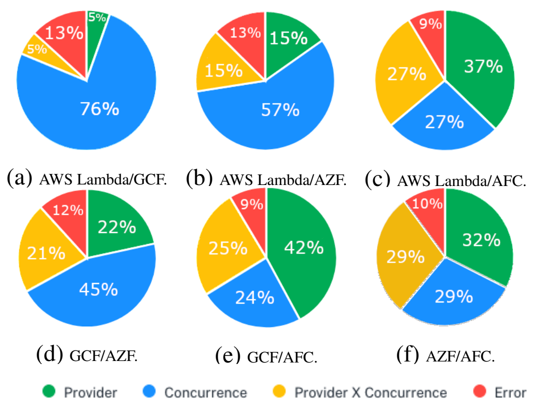


Figure 7: Factorial design results.

these benchmark results, the providers' strategies influenced the result less than the differences between the concurrences. In Figures 7c, 7e and 7f, there is a predominance of the provider factor and this corroborates the results shown previously for average execution time and failure rate, since AFC participates in both factorial designs.

The result of the t-tests for the six comparisons between the providers is shown in Table 3. It is possible to notice that all the differences between the average results were considered statistically relevant with a 99.95% confidence level. From which it may be stated that these differences are not irrelevant and may be considered as defining the result of the benchmarks.

Table 3: T-test results.

Scenario	Difference (ms)	Standard deviation	Confidence level (%)
AWS x GCF	1,317.03	281.82	99.95
AWS x AZF	7,792.25	928.90	99.95
AWS x AFC	5,454.88	259.53	99.95
GCF x AZF	9,109.28	898.69	99.95
GCF x AFC	4,137.84	110.14	99.95
AZF x AFC	13,247.13	891.95	99.95

The results of this work can serve as input in decision-making processes according to the characteristics and requirements of the real use case. For example, if the use case requires high reliability of the FaaS service, AFC would be the best option among the evaluated providers, since it did not present failures. On the other hand, if the use case is very sensitive to cold start, then the Azure provider should be avoided, as it presented high values in this regard.

6 CONCLUSION

In this work, the distributed architecture of the Orama framework was presented, with which it is possible to perform benchmarks with high levels of concurrency on a FaaS environment, as well as configure the load to be triggered from different locations on the globe, considerably expanding the range of possibilities for benchmarks against state-of-the-art tools.

The Orama framework allows the visualization of different scenarios for the same use case, especially for concurrency levels above 512 concurrent requests, which is the amount observed as the safest maximum for management by an intermediate configuration instance. In addition, it was evidenced that the Orama framework in its distributed architecture is capable of visualizing the same behavior as the FaaS provider when subjected to a concentrated and distributed approach of bursts of requests, as well as eventual differences.

In the experiments, it was also evident that the AFC FaaS delivers greater consistency in terms of average execution time and occurrence of failures, followed by AWS Lambda and CGF, which registered close results, and finally the AZF results with high average execution times and cold start.

In future work, other providers will be integrated into the Orama framework, such as IBM and Oracle, in order to expand the coverage of the analyses presented in this work. Furthermore, even higher levels of concurrency can be evaluated by designing experiments that include a larger number of distributed workers. Furthermore, evolutions in the Orama framework will allow the analysis of benchmark results using percentiles.

REFERENCES

Back, T. and Andrikopoulos, V. (2018). Using a microbenchmark to compare function as a service solutions. In *ECSOCC*, pages 146–160. Springer.

Barcelona-Pons, D. and García-López, P. (2021). Benchmarking parallelism in faas platforms. *Future Generation Computer Systems*, 124:268–284.

Carvalho, L. and Araujo, A. (2022). Orama: A benchmark framework for function-as-a-service. In *Proceedings of the 12th CLOSER*, pages 313–322. INSTICC, SciTePress.

Carvalho, L. R. and Araujo, A. P. F. (2020). Performance comparison of terraform and cloudify as multicloud orchestrators. In *2020 20th IEEE/ACM CCGRID*, pages 380–389.

Copik, M., Kwasniewski, G., Besta, M., Podstawski, M., and Hoefler, T. (2021). Sebs: A serverless benchmark suite for function-as-a-service computing.

DUAN, Y., Fu, G., Zhou, N., Sun, X., Narendra, N. C., and Hu, B. (2015). Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends. volume 00, pages 621–628.

Erinle, B. (2013). *Performance testing with JMeter 2.9*. Packt Publishing Ltd. ISBN: 9781782165842.

Gartner (2022). Gartner forecasts worldwide public cloud end-user spending to reach nearly $600 billion in 2023. [online; 01-Jan-2023; url: https://tinyurl.com/3z72zebw].

Grambow, M., Pfandzelter, T., Burchard, L., Schubert, C., Zhao, M., and Bermbach, D. (2021). Befaas: An application-centric benchmarking framework for faas platforms.

Gupta, P. R., Taneja, S., and Datt, A. (2014). Using heat and ceilometer for providing autoscaling in openstack. *JIMS8I-International Journal of Information Communication and Computing Technology*, 2(2):84–89.

HashiCorp (2021). Terraform: Write, plan, apply. [online; 11-Aug-2021; url: https://www.terraform.io].

Ibrahim, M. H., Sayagh, M., and Hassan, A. E. (2021). A study of how docker compose is used to compose multi-component systems. *Empirical Software Engineering*, 26(6):128.

Jain, R. (1991). The art of computer systems: Techniques for experimental design, measurement, simulation, and modeling. ISBN:13.978-0471503361.

Kuhlenkamp, J., Werner, S., Borges, M. C., El Tal, K., and Tai, S. (2019). An evaluation of faas platforms as a foundation for serverless big data processing. In *Proceedings of the 12th IEEE/ACM*, UCC'19, page 1–9, NY, USA. ACM.

Maissen, P., Felber, P., Kropf, P., and Schiavoni, V. (2020). Faasdom. *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*.

MELL, P. and Grance, T. (2011). The NIST definition of cloud computing. *National Institute of Standards and Tecnology*.

Motta., M. A. D. C., Reboucas De Carvalho., L., Rosa., M. J. F., and Favacho De Araujo., A. P. (2022). Comparison of faas platform performance in private clouds. In *Proceedings of the 12th CLOSER,*, pages 109–120. INSTICC, SciTePress.

Sax, M. J. (2018). *Apache Kafka*, pages 1–8. Springer International Publishing, Cham. ISBN: 978-3-319-63962-8.

Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N. J., Popa, R. A., Gonzalez, J. E., Stoica, I., and Patterson, D. A. (2021). What serverless computing is and should become: The next phase of cloud computing. *ACM*, 64(5):76–84.

Somu, N., Daw, N., Bellur, U., and Kulkarni, P. (2020). Panopticon: A comprehensive benchmarking tool for serverless applications. In *2020 COMSNETS*, pages 144–151.

Wen, J., Liu, Y., Chen, Z., Chen, J., and Ma, Y. (2020). Characterizing commodity serverless computing platforms. DOI:10.48550/ARXIV.2012.00992.

Wittig, M. and Wittig, A. (2018). *Amazon web services in action*. Simon and Schuster. ISBN: 978-1617295119.