Orama: A Benchmark Framework for Function-as-a-Service

Leonardo Reboucas De Carvalho[a] and Aleteia Patricia Favacho Araujo[b]
Department of Computing Science, University of Brasília, Brasilia, Brazil

Keywords: Cloud, FaaS, Benchmark, AWS, GCP, Factorial Design, T-test, Orama Framework.

Abstract: The prominent Function-as-a-Service (FaaS) cloud service model has positioned itself as an alternative for solving several problems, and, interest in cloud-oriented architectural solutions that use FaaS has therefore grown rapidly. Consequently, the importance of knowing the behavior of FaaS-based architectures under different concurrency scenarios has also become significant, especially in implementation decision-making processes. In this work, the Orama framework is proposed, which helps in the execution of benchmarks in FaaS-based environments, orchestrating the deployment of pre-built architectures, as well as the execution of tests and statistical analysis. Experiments were carried out with architectures containing multiple cloud services in conjunction with FaaS in two public cloud providers (AWS and GCP). The results were analyzed using factorial design and t-test and showed that the use cases running on AWS obtained better results in runtime compared to their counterparts on GCP, but showed considerable error rates in competition situations. It is worth mentioning that the Orama framework was used from in the automated provisioning of use cases, execution of benchmarks, analysis of results and deprovisioning of the environment, supporting the entire process.

1 INTRODUCTION

Studies indicate that Function-as-a-Service (FaaS) (Malawski et al., 2020), also called Serverless (Nupponen and Taibi, 2020) or Backend-as-a-Service (Schleier-Smith et al., 2021), will become the main computing paradigm of the Cloud Era (Schleier-Smith et al., 2021). All this success can be explained, in part, by the great ease in using this service model, as well as in the automatic and transparent elasticity of delivery.

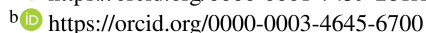
The growing adoption of FaaS-based architectures has increased concern about performance aspects related to environments deployed by this type of service. Some works such as FaaSdom (Maissen et al., 2020) and Sebs (Copik et al., 2021) for example, have investigated the behavior of FaaS services, mainly exploring the characteristics of programming languages and providers' strategies.

However, productive applications often involve more than one cloud service. In this context, it is important to know the behavior of this type of application when faced with different levels of concurrency. For this reason, this work proposes the Orama Framework to provide assistance in the execution of benchmarks in FaaS-oriented solutions, from automated deployment, to test execution and statistical analysis.

The Orama framework has built-in FaaS architectures ready to be deployed in cloud environments and then submitted to user-customized test batteries. When the tests are finished, depending on the configuration, the user will be able to access a 2^k factorial design (Jain, 1991), where k is the number of factors analyzed.

Experiments were performed to validate the framework on two public cloud providers (AWS and GCP). For the tests, six Orama built-in use cases were applied, which enabled explorationat various levels from simple activation of FaaS functions to integration with database services and object storage.

This article is divided into seven parts, the first being an introduction. Section 2 presents the theoretical foundation that supports the work, while Section 3 describes the proposal, that is, the Orama Framework. Section 4 presents the related works, while Section 5 describes the methodology used in the experiments carried out to validate the framework. Section 6 shows the results obtained and finally, Section 7 presents the conclusions and future work.

[a] https://orcid.org/0000-0001-7459-281X

[b] https://orcid.org/0000-0003-4645-6700

Carvalho, L. and Araujo, A.
Orama: A Benchmark Framework for Function-as-a-Service.
DOI: 10.5220/0011113900003200
In *Proceedings of the 12th International Conference on Cloud Computing and Services Science (CLOSER 2022)*, pages 313-322
ISBN: 978-989-758-570-8; ISSN: 2184-5042
Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

2 BACKGROUND

Since NIST (MELL and Grance, 2011) defined cloud service models in 2011 as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) several public cloud providers have named their services using "as-a-service" suffixes quite differently from the initial NIST definition.

Although these services differ from the NIST naming definition, their features generally fit with one of the traditional models. This profusion of services gave rise to the term "XaaS" (DUAN et al., 2015) to generalize "everything-as-a-Service". In this scenario Function-as-a-Service emerged.

2.1 Function-as-a-Service

Function-as-a-Service (FaaS) (Schleier-Smith et al., 2021) consists of a cloud computing service model in which the client submits a snippet of source code to the provider and configures a trigger, which can either be from other services within the provider's platform or through a REST API. This type of service is also known as Serverless (Nupponen and Taibi, 2020) and Backend-as-a-Service (Schleier-Smith et al., 2021).

Once the service is activated through the trigger, the provider must guarantee its processing, automatically and transparently providing instances to meet new requests that exceed the service capacity of the provisioned environment (auto scaling). The billing model is based on requests and takes into account the execution time and the amount of resources allocated to each request. Providers often impose limits on runtime, amount of RAM and vCPUs, etc.

This service model has grown significantly thanks to its ease of adoption and its usage-adjusted charging model. In this it differs from the IaaS model, in which charging is based on how long the machines are in operation, even if they are not in use. In FaaS the customer is only charged when their application is effectively used.

Leading public cloud providers offer FaaS options such as AWS with Lambda (AWS, 2021), Google Cloud Platform (GCP) with Google Cloud Function (GCF) (Google, 2021), Microsoft Azure with Azure Function (Microsoft, 2021), IBM with IBM Cloud functions (IBM, 2021), Oracle with Oracle Cloud Function (Oracle, 2021), and Alibaba with Alibaba Cloud Function (Cloud, 2021).

Performance metrics, especially related to latency, are excellent tools in the decision-making process for deploying environments, especially in a cloud computing context. In this context, knowing the performance of the solutions is highly desirable and for this it is possible to use benchmarks to measure performance in a systematic way. Some works have developed benchmarks for performing performance tests in FaaS environments, such as FaaSdom (Maissen et al., 2020) and Serverless Benchmark Suite (Sebs) (Copik et al., 2021). It is important to know the behavior of FaaS-based architectures in collaboration with other service models (IaaS and PaaS, for example) offered by providers, as these scenarios in turn reflect the operational reality of production environments.

In a benchmark process it is important to have analytical tools that allow the identification of insights from the data obtained from the benchmarks. A tool that can accomplish this task is the 2^k factorial design (Jain, 1991). By defining two levels (lower and upper) for factors concerning variables supposed to have influence on a particular phenomenon, it is possible to estimate the percentage of the effects of each factor on the calculated result, as well as the effects of the combined factors.

Another statistical analysis tool is the paired t-test. In this test, two samples have their result difference analyzed in order to determine if the difference found is statistically significant. This determination occurs through the calculation of the difference confidence interval. If this interval contains the number zero, then the difference is not significant (within a certain confidence level), otherwise the difference can be considered as significant and its semantics can be analyzed, in relation to the problem in question. It is noteworthy that a t-test should only be performed on populations with up to 30 samples, since the t-student distribution is used for its calculations and it has such a restriction. The next section will present the Orama framework proposal in detail.

The framework proposed in this work has a factorial design module that is capable of performing analysis with two factors ($k = 2$) (provider and concurrence), having latency as the objective variable. In addition, it also has the execution of the t-test, if applicable. The next section will present the Orama framework proposal in detail.

3 ORAMA FRAMEWORK

With the purpose of facilitating the execution of benchmarks on FaaS-oriented environments, especially when the use cases involve other cloud service models, in this work the Orama framework was proposed. "Orama" is a word of Greek origin that means "Sight". The name chosen for the framework demonstrates the intention to shed light on the behavior of FaaS environments. The main features of the

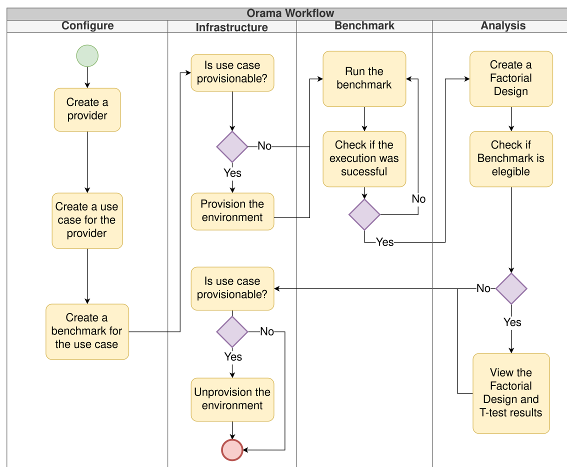


Figure 1: Orama workflow.

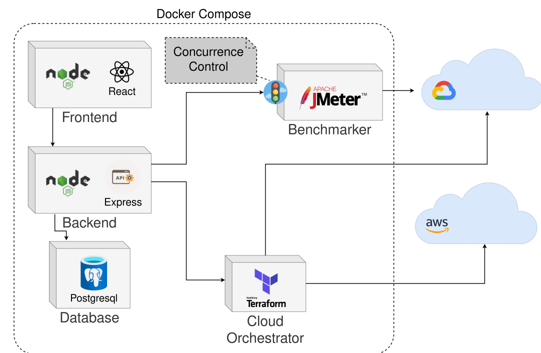


Figure 2: Orama architechture.

framework are: open-source; built-in use cases; automatic provisioning of built-in use cases; custom use cases (without provisioning); semaphore controlled simultaneous benchmark execution; detailed results of benchmark executions; possibility of re-running benchmarks; control of the number of requests made by benchmarks; additional request option to bypass cold-start; 2^k ($k = 2$) factorial design module for statistical analysis between two benchmarks; and t-test to analyze the difference between two benchmarks.

3.1 Orama Workflow

Figure 1 presents the Orama framework workflow. In the diagram shown, it is possible to observe four lanes that represent the main phases of the framework: configuration, infrastructure preparation, benchmark execution and analysis. In the configuration phase, it is necessary to create, at least, a provider, a use case and a benchmark to start the operation of the framework.

When installing the framework, it is possible to run a command to automatically create built-in use cases and some benchmark scenarios. Use cases consist of architectural definitions that have automation artifacts that allow them to be provisioned and deprovisioned in an automated way. It is also possible, at the time of framework installation, to configure the AWS and GCP credentials that will enable the deployment of use cases related to these providers. Framework installation instructions are available in Orama's public repository[1]. In addition, it is also possible to install it using pre-built and published Docker images from the Docker-Hub[2].

Once properly installed and configured, the next step is to deploy the environment in which the tests will run. The Orama framework allows the configuration of custom use cases that do not need to be deployed, in this case the user configures directly in the framework the URLs that are part of the use case, whether they are GET, POST, PUT or DELETE.

If the environment needs deployment, the user can command this through the framework's frontend interface and monitor its deployment. The built-in use cases have all the automation artifacts needed to be deployed to the respective providers and these artifacts are applied in this phase.

After being deployed, environments are ready to be tested. In this phase the user can run the benchmarks as many times as necessary. A benchmark consists of running bursts of requests over a use case URL. It is possible to configure a list of bursts in a benchmark, as well as the number of repetitions of that scenario, in order to obtain samples for statistical analysis.

It is also possible to configure in a benchmark the execution of an additional request, before the beginning of the tests, as a service warm-up, in order to avoid a cold-start. Furthermore, they may be configured in such a way that they intervene between the executions of the bursts, if the intention is to analyze the cold start, or any other requirement that demands an interval between requests.

Once the benchmarks are run, it is possible to carry out comparative analyzes between them, as long as certain conditions are respected. For this version of the framework, it is only possible to compare benchmarks containing different providers, two equal levels of concurrency and the same number of repetitions. Once these criteria are met, it will be possible to visualize the 2^2 factorial design and paired t-test results.

The factorial design will show, among other metrics, the percentage of the effect of the provider, concurrence and both combined for the results obtained in the benchmark. The t-test will present the signif-

[1] https://github.com/unb-faas/orama
[2] https://hub.docker.com/u/oramaframework

icance of the difference found between the latency means of the results at various confidence levels from 60% to 99.95%.

At the end of the analysis, the user can deprovision the deployed environment to carry out the tests (if applicable), thus ending the workflow of the framework.

3.2 Orama Architecture

In order to achieve the intended objective of the Orama framework an architecture was defined and an application was developed consisting of five main components as shown in Figure 2. All components of the framework are designed to be built on top of container runtime and cooperate with each other via the HTTP protocol, as well as with external entities such as cloud providers, or other target testing platforms. The five main components of the Orama Framework are:

- **Frontend:** application developed in Node.js using React and Material UI frameworks whose objective is to provide a friendly interface between the user and the framework, allowing the management of application assets such as providers, use cases, benchmarks and factorial designs. It is also possible to execute the provisioning and deprovisioning processes, as well as the benchmarks and the visualization of the results;
- **Backend:** application developed in Node.js using the express.js framework whose objective is to provide a centralized API for the frontend so that requests from the frontend are always directed to this component for treatment and eventual redirection to another service within the Orama ecosystem framework. It is also the component responsible for interacting with the application's database;
- **Database:** consists of the framework's data storage layer, composed of the Postgresql database;
- **Cloud Orchestrator:** application developed in Node.js using the express.js framework whose objective is to provide an API for provisioning and deprovisioning the environment through Terraform (HashiCorp, 2021). This component waits for infrastructure definition artifacts to exist for a use case in order to apply it to the selected provider. The use cases can be easily extended just by creating new directory structures with the respective Terraform definition files and the proper framework configuration to trigger it;
- **Benchmarker:** application developed in Node.js using the express.js framework whose objective is to provide an API for executing benchmarks through the JMeter tool (Halili, 2008). The framework has a generic JMeter benchmark execution artifact that takes several parameters that are expected by the Benchmarker component. This component also generates a web page with test metrics from a CSV file that is accessed from the frontend after running the benchmark when viewing the results. This component allows simultaneous executions of benchmarks, but does not effect them at the same time, there is a semaphore control that guarantees that only one benchmark is running at a time.

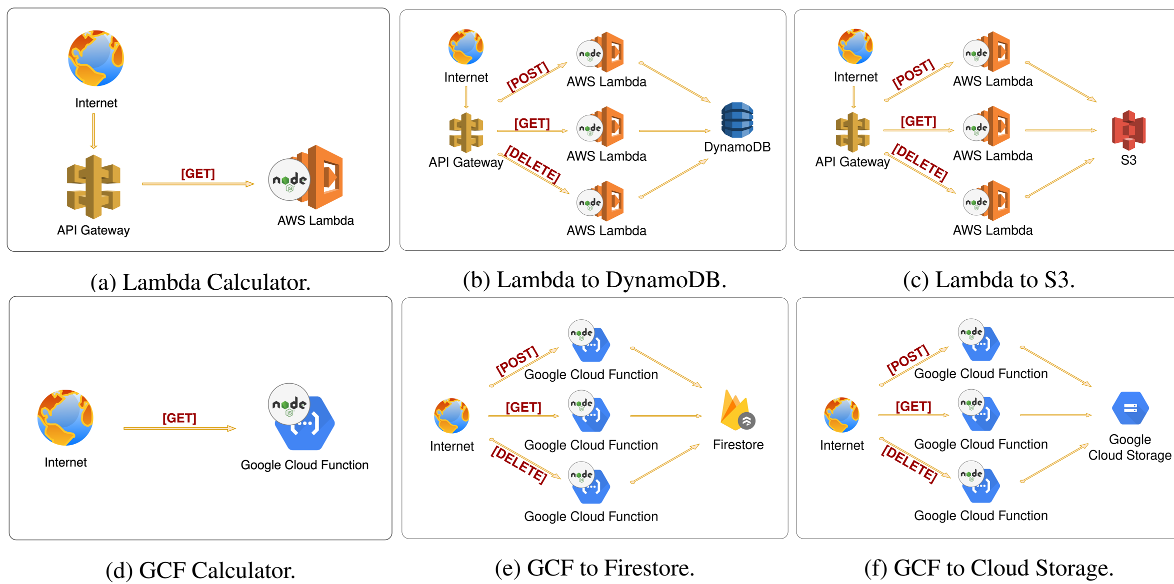
The backend, cloud orchestrator and benchmark components have an API documentation view using Swagger[3] in order to facilitate an eventual direct consumption of data from these components.

3.3 FaaS Built-in Use Cases

The Orama framework has some built-in FaaS-oriented use cases that can be deployed in AWS and GCP in order to pass through batteries of tests, or even serve as bases for productive applications, since the framework makes available the artifacts, definition of the infrastructure, and source codes of the functions. In this version Orama provides six built-in use cases with the purpose of representing a simple use of the FaaS paradigm in both providers and two other more complex uses involving integrations between FaaS and other providers services, these use cases are described below.

1. **Lambda Calculator:** Figure 3a shows the architecture of the Lambda Calculator use case in which one simple Node.js math calculator function in AWS is deployed in a Lambda service, as well as an API Gateway that makes it accessible via REST API.
2. **GCF Calculator:** Figure 3d shows the architecture of the GCF Calculator use case in which one simple Node.js math calculator function in GCP is deployed in a GCF service. The GCF service makes available a REST API.
3. **Lambda to DynamoDB:** Figure 3b shows the architecture of use case Lambda to DynamoDB in which three Lambda functions are deployed (accessible via REST API through the API Gateway) to meet requests to include, obtain and exclude data in json format in a database. For data storage, a DynamoDB service is provided with which

[3]Swagger is an Interface Description Language for describing RESTful APIs expressed using JSON.



(a) Lambda Calculator. (b) Lambda to DynamoDB. (c) Lambda to S3.

(d) GCF Calculator. (e) GCF to Firestore. (f) GCF to Cloud Storage.

Figure 3: Orama built-in use cases.

integrations between Lambda functions are implemented.

4. **GCF to Firestore:** Figure 3e shows the architecture of use case GCF to Firestore in which three GCF functions are deployed to meet requests to include, obtain and exclude data in json format in a database. For data storage, a Firestore service is provided with which integrations between GCF functions are implemented.
5. **Lambda to S3:** Figure 3c shows the architecture of use case Lambda to S3 in which three Lambda functions are deployed (accessible via REST API through the API Gateway) to meet requests to include, obtain and exclude data in json format in a S3 bucket.
6. **GCF to Cloud Storage:** Figure 3f shows the architecture of use case GCF to Cloud Storage in which three GCF functions are deployed to meet requests to include, obtain and exclude data in json format in a Cloud Storage bucket.

The use cases offered by Orama can be easily deployed to providers from the frontend of the framework. When deploying a use case, the architectural definition artifacts will be offered to Terraform, which will communicate with the provider and carry out the deployment, including the deployment of functions, whose source code is also part of the use case's directory structure.

Conversely, for environment de-provisioning, it is enough for the user to command it from the frontend and the aforementioned artifacts will be offered to Terraform, now for environment demobilization.

It is noteworthy that unlike other tools aimed at performing FaaS benchmarks with a focus on exploring the implementations of runtimes offered by providers for some programming languages, Orama is intended to serve as a basis for the orchestration of FaaS benchmarks whose integration between services is present.

With high extensibility, the proposed framework can be easily adjusted to run tests in other scenarios, shortening the distance between not knowing the behavior of a solution in certain situations and the possibility of a detailed analysis using robust statistical tools.

4 RELATED WORKS

FaaS-dom (Maissen et al., 2020) is a modular set of benchmarks for evaluating serverless computing that includes FaaS workloads and supports the main FaaS providers (AWS, Azure, Google, and IBM). The functions implemented to carry out the tests by FaaS-dom are a strong point, as they are written in several languages and for several providers. However, they do not have an integration approach with other cloud services as Orama has, and the tasks that the FaaS-dom functions perform can be considered trivial.

Serverless Benchmark Suite (Sebs) (Copik et al., 2021) consists of specifying representative workloads, monitoring the implementation, and evaluating the infrastructure. The abstract model of a FaaS implementation ensures the applicability of the benchmark to various commercial vendors such as AWS,

Azure and Google Cloud. This work evaluates aspects such as: time, CPU, memory, I/O, code size and cost based on the performed test cases. Despite this, their test cases do not involve integration with other cloud services as in the framework proposed in this work, Orama.

In (Wen et al., 2021) the authors perform a detailed evaluation of FaaS services: AWS, Azure, GCP and Alibaba, by running a test flow using microbenchmarks (CPU, memory, I/O and network) and macrobenchmarks (multimedia, map-Reduce and machine learning). The tests used specific functions written in Python, Node.js and Java that explored the properties involved in benchmarking to assess the initialization latency and efficiency of resource use. The benchmarks used in the analysed work also do not integrate with other cloud services as Orama does, making their scope limited.

BeFaaS (Grambow et al., 2021) presents an application-centric benchmark framework for FaaS environments with a focus on evaluation with realistic and typical use cases for FaaS applications. It supports federated benchmark testing where the benchmark application is distributed across multiple vendors and supports refined result analysis. However, it does not offer an improved method of statistical analysis such as factorial design or t-test, as proposed in this paper.

5 METHODOLOGY

With the purpose to assess the behavior of the Orama framework, three test scenarios were defined as shown in Figure 4. In Scenario 1, the framework was installed on a local physical machine with Docker engine available and its six use cases were deployed in the respective clouds, that is, three in AWS and three in GCP.

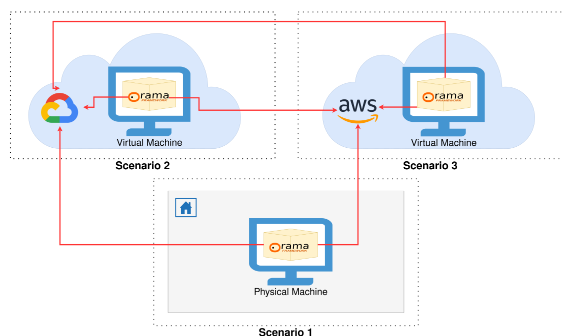


Figure 4: Test architecture.

In Scenario 2, a virtual machine was instantiated in the GCP provider that received the installation of the Docker engine and then the installation of Orama framework. As in Scenario 1, the six use cases were deployed in the respective clouds, but in Scenario 2 the distance between the framework and the GCP use cases is much smaller, as both the framework and the use cases are running inside from the same provider.

In Scenario 3, a virtual machine was instantiated in the AWS provider that had received the installation of the Docker engine and then the installation of the Orama framework. As in Scenario 2, in Scenario 3 there is greater proximity between the framework and the use cases related to AWS, since both are on the same provider.

It is noteworthy that the orchestration component of Orama uses random names to name the resources in the cloud and because of that there is no name conflict, even if the three scenarios are active at the same time.

Table 1 presents the configuration of the physical machine used in testing Scenario 1, as well as the configurations of the instances created for testing Scenarios 2 and 3.

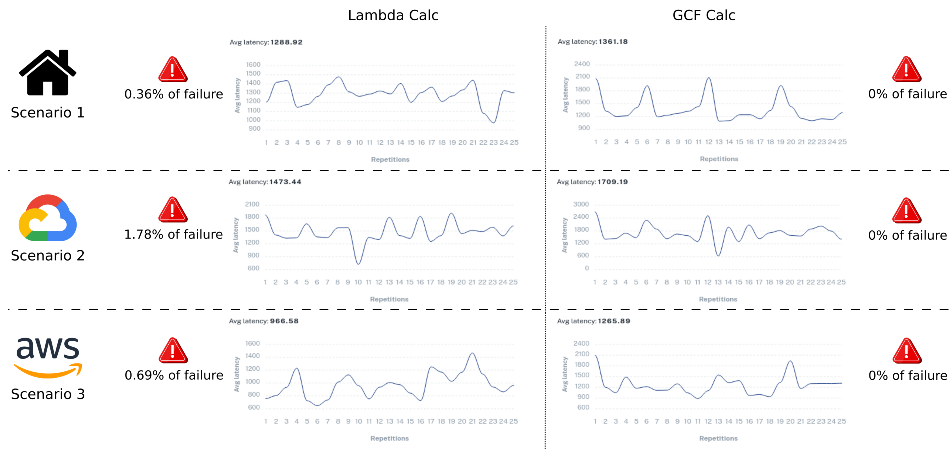
Table 1: Experiment scenarios parameters.

	Place	RAM	CPU	Region	OS	Flavor
1	Local	16GB	8	Brasilia/Br	Ubuntu 20.04	N/A
2	AWS	16GB	8	us-east-1	Ubuntu 20.04	c5.2xlarge
3	GCP	16GB	8	us-central1	Ubuntu 20.04	Custom

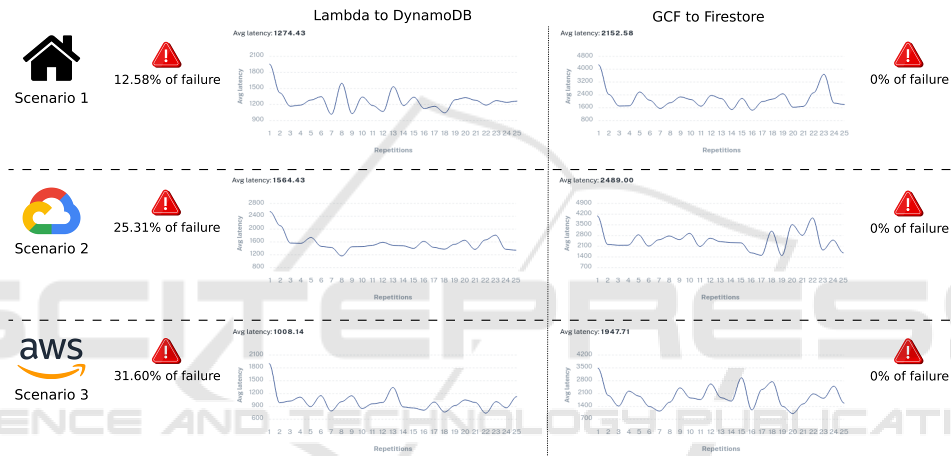
Once the framework was installed and the use cases implemented in each provider, identical batteries of tests were run for all use cases in all scenarios. The test battery consists of subjecting the environment to two levels of concurrent load. A first level with 10 simultaneous requests and another level with 100 simultaneous requests and each test was repeated 25 times. From these test batteries it was possible to visualize the results of the factorial design and the t-test, as will be presented and discussed in Section 6.

6 RESULTS

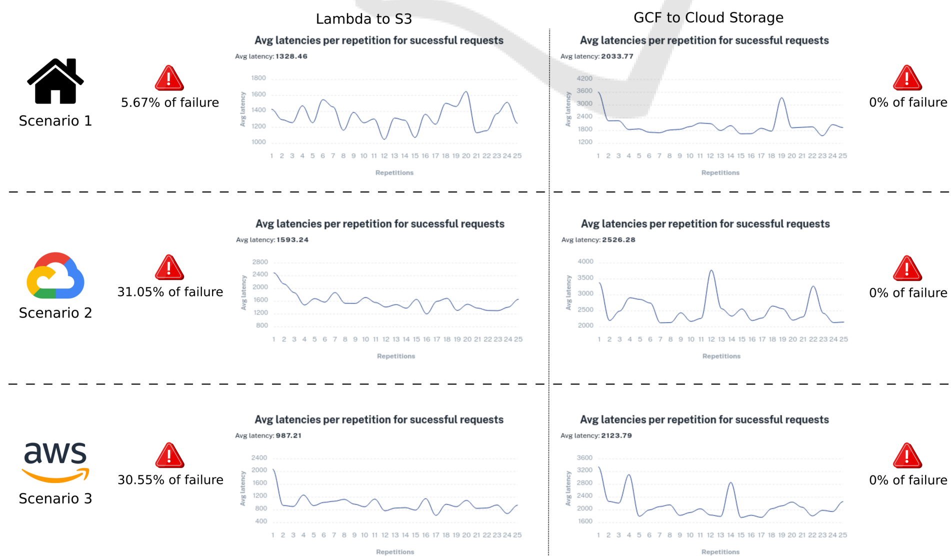
Once the environment for each scenario was provisioned, the six built-in test cases were executed. At the end of the execution of the test cases, factorial designs were created comparing the equivalent use cases. A factorial design was created between Lambda Calc and GCF Calc, as both are simple use cases aimed at creating a mathematical calculator service. A factorial design was created between Lambda to DynamoDB and GCF to Firestore, as they are use cases that relate FaaS to Database-as-a-Service (DBaaS). A factorial design was created



(a) Calculators use cases.



(b) Database-relateds use cases.



(c) Object storage-relateds use cases.

Figure 5: Paired experiment results.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

between Lambda to S3 and GCF to Cloud Storage, which are the use cases that relate FaaS with object storage services.

In Figure 5a it is possible to observe that the graphs show the alternation of average latency times between repetitions. This is due to the unpredictable characteristic of the data network, however some peaks and troughs are noticeable and these indicate the provider's performance in order to adjust the elasticity of the environment during the execution of the test batteries.

The difference in the mean latency time plateau that exists between the results in the two providers is evident. For example, in Scenario 1 the range calculated for the Lambda Calc graph was fixed between 900 and 1600 milliseconds, that is, the average latency times in the repetitions varied within this range. On the other hand, the graph referring to GCF Calc in Scenario 1 had its range established between 900 and 2400, that is, the average latency of the GCF Calc varied in a greater range than the average latency of the Lambda Calc. This difference in level was reflected in the overall average, in which Lambda Calc scored 1288.92 while GCF Calc scored 1361.18. This difference was also evidenced in Scenarios 2 and 3 and this indicates that the use case Lambda Calc on average is able to execute requests faster than the GCF Calc.

However, when analyzing the failure rates, that is, the percentage of unsuccessful requests, it is possible to see that in the three scenarios the use case Lambda Calc presented percentages above 0%, while the use case GCF Calc remained continuously at 0% in all scenarios, giving it a higher degree of reliability.

Figure 5b presents the latency averages calculated in each repetition of the use cases Lambda to DynamoDB and GCF to Firestore in the three scenarios, since both use cases refer to solutions that relate FaaS to DBaaS. In Figure 5b the role of providers in increasing the scale of environments is clearly visible, both in the AWS use case and in the GCP use case, since the average calculated in the first repetition is the highest in the series, while from the second repetition onwards the averages drop drastically and from the third repetition onwards the oscillations reflect the variations in the transmission rate of the data network.

Analyzing Figure 5b it is possible to notice that the fastest execution of the use case Lambda to DynamoDB occurred in Scenario 3 where the framework was running inside the provider in which the use case is deployed (AWS), although this is also the case in Scenario 3, the use case Lambda to DynamoDB has the highest failure rate at 31.60%, while in Scenarios 1 and 2 the failure rates were at 12.58% and 25.31%, respectively. These high error rates demonstrate that the AWS provider, in these scenarios, experienced difficulties in handling concurrent requests. These problems may be related to temporary instabilities faced by the network during the testing period, to improper configurations of the use cases, or even to a problem in the provider's platform. The tests were repeated several times in order to compose the mass of data, but an instability could have covered the entire period of the tests. Regarding the use cases, they were created using the "default" configurations indicated by the manufacturer, and can be easily adjusted to generate new use cases with different configurations in order to evaluate the same scenarios against different configurations. All this is thanks to the flexibility provided by the Orama framework.

The CGF to Firestore use case had the lowest average latency in Scenario 3, although in this scenario the Orama framework was deployed on the AWS provider. This indicates that the network layer of the AWS IaaS service managed to be more efficient in transmitting data than the respective layer in GCP, while in Scenario 2, where the framework was deployed in a GCP IaaS service, the average latency was even higher than in Scenario 1, in which the framework was running outside the clouds, in a local environment, which reinforces the perception that the network layer of the GCP IaaS service contributed to these results.

It is noteworthy that the use of the Orama Framework allowed these findings with a much reduced effort, since the implementation of test cases, as well as the conduct of benchmarks occurred in an automated manner.

Figure 5c shows the results of the use case related to object storage in the three scenarios. In this figure, it is possible to observe a large variation in latency averages between repetitions in Scenario 1 for both use cases and this is due to the intrinsic variations of the data network, since in Scenario 1 the Orama framework is deployed outside the clouds.

The peaks that occur in all scenarios for the CGF to Cloud Storage use case are noteworthy, these peaks may represent the actuation of the providers when reduce the scale of the environment after the test execution immediately before. Another fact that stands out is the presence of the same behavior perceived in Figure 5b. Once again the GCP-related use cases ran faster in scenarios where the Orama Framework was deployed outside of it. As previously mentioned, this fact may be related to the characteristics of the GCP's internal network that supports the IaaS environment where the Orama framework was installed. It is possible to view the average times of all repetitions in each use case for each scenario separately in Table 3.

Table 2: T-test results.

Use case type	Scenario								
	1 - Local			2 - GCP			3 - AWS		
	Difference	T-test	Confidence	Difference	T-test	Confidence	Difference	T-test	Confidence
Calculator	72.26	Passed	80%	235.75	Passed	97.50%	299.32	Passed	99.95%
Database	878.15	Passed	99.95%	924.57	Passed	99.95%	939.57	Passed	99.95%
Object Storage	705.31	Passed	99.95%	933.04	Passed	99.95%	1136.58	Passed	99.95%

Figure 5c also shows that although AWS-related use cases process requests faster than GCP use cases, their error rates are not much higher, especially in those scenarios where the framework was deployed in the cloud, where error rates reached about 31%.

An important feature offered by the Orama Framework is factorial design analysis. Figure 6 shows the results of the confrontation between the use cases in each scenario. This figure shows, in a pie format, the percentages of the effects calculated by the Factorial Design for the factors: provider (in green), concurrence (in blue), the relationship between provider and concurrence (in yellow) and the sampling error (in red).

The effect of sampling error within a factorial design indicates the existence of some factor interfering with the results beyond those mapped (provider and concurrence), since this effect is calculated from the difference between the values calculated in the samples and the mean of these samples. As the benchmarks use an approach that massively executes requests over the data network, especially over the internet, this factor interferes with the results and this can be corroborated by analyzing that the portions of the sampling error effects in Scenario 1 are higher than those portions determined in Scenarios 2 and 3 where, one of the use cases was tested from an installation of the framework within its own provider, promoting a reduction in data traffic.

Table 3: Average latencies in each use case (milliseconds).

Use case type		Scenario		
		1 - Local	2 - GCP	3 - AWS
Calculator	AWS	1288.92	1473.44	966.58
	GCP	1361.18	1709.19	1265.89
Database	AWS	1274.43	1564.43	1008.14
	GCP	2152.58	2489.00	1947.71
Object Storage	AWS	1328.46	1593.24	987.21
	GCP	2033.77	2526.28	2123.79

In Figure 6 it is possible to verify that in the comparison between Lambda Calc and GCF Calc in the factorial design results there is a predominance of the concurrence effect (blue slices) to the detriment of the other factors. In the comparison between Lambda to DynamoDB and GCF to Firestore there is a greater in-

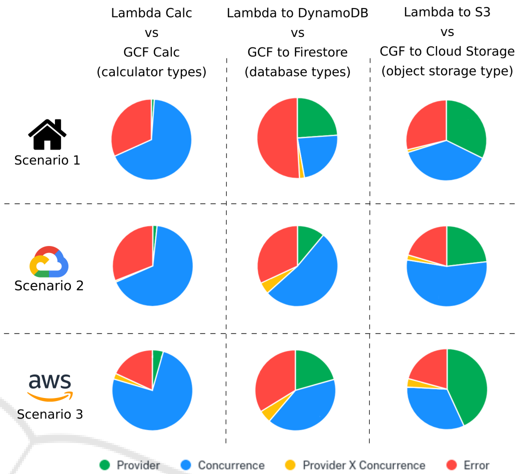


Figure 6: Factorial designs results.

fluence of the provider effect, but in Scenarios 2 and 3, where the sampling error has less influence, it is possible to see that the provider influence loses impact and once again the factor concurrence exerts greater influence on the results obtained. When analyzing the confrontation between S3 Lambda use cases such as GCF to Cloud Storage, the effects shift their influence between scenarios. While in Scenario 1 the provider and concurrence effects have practically the same influence, in Scenario 2 the concurrence effect outweighs the provider effect. In Scenario 3, it is the provider that exerts the greatest influence.

The results of factorial designs show that for the use cases tested, in general, the results are mostly impacted by the concurrence effect. Table 2 shows the results of the t-tests between pairs of use cases of the same type, where it is possible to find a difference between the average latencies. It is possible to notice that in all tests the difference was considered significant, once the test was passed. Most of the differences managed to pass the test with 99.95% confidence level, only in the use cases related to the calculator for Scenarios 1 and 2 the degree of confidence with which the difference passed the t-test was 80% and 99.75%, respectively. The results of the experiment shown in this section are available in detail on GitbHub[4].

[4]https://github.com/unb-faas/orama-results

7 CONCLUSION

Considering the results shown, it is possible to conclude that the Orama framework proposed in this work is capable of running benchmarks in several FaaS scenarios, allowing comparative analyzes between benchmarks both in terms of absolute performance and through statistical analyzes such as factorial design and t-test.

The pre-configured use cases that are part of the Orama Framework are tools that can help the community beyond what they propose. They can serve as a basis for building other similar benchmarks with different configurations or even for different benchmarks, with services that have already been addressed by the original use case. In addition, Orama use cases can also serve as a starting point for a solution in a productive environment, since it has a defined and robust orchestration process.

In the experiment, the framework was installed in different positions in relation to the target clouds of the use cases. Installation, provisioning and test execution are automated, allowing quick and easy analysis from different points of view. Overall, the results of experiments on AWS and GCP providers showed that use cases running on AWS managed to be faster than equivalent use cases running on GCP. However, there was a high occurrence of errors in the executions in AWS, while in GCP the use cases did not present errors.

In future work, use cases may be incorporated to increase the scope of testable solutions. Thanks to the decoupled structure of the Orama Framework, incorporating new use cases is a relatively simple process. Thus, it is possible, any out further work, for example, in the investigation of performance problems commonly found in FaaS architectures, varying different configuration possibilities in order to find those parameters that exert the greatest influence on the results.

REFERENCES

AWS (2021). AWS lambda. https://aws.amazon.com/en/lambda. [online; 11-Aug-2021].

Cloud, A. (2021). Alibaba cloud function. https://www.alibabacloud.com/product/function-compute. [online; 11-Aug-2021].

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.

Google (2021). Cloud functions. https://cloud.google.com/functions/. [Online; 10-Aug-2021].

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

Halili, E. H. (2008). *Apache JMeter*. Packt Publishing Birmingham.

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

IBM (2021). IBM cloud functions. https://cloud.ibm.com/functions/. [online; 11-Aug-2021].

Jain, R. (1991). *The art of computer systems: Techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons,.

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

Malawski, M., Gajek, A., Zima, A., Balis, B., and Figiela, K. (2020). Serverless execution of scientific workflows: Experiments with hyperflow, AWS lambda and Google Cloud Functions. *Future Generation Computer Systems*, 110:502–514.

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

Microsoft (2021). Azure functions. https://azure.microsoft.com/pt-br/services/functions/. [online; 11-Aug-2021].

Nupponen, J. and Taibi, D. (2020). Serverless: What it is, what to do and what not to do. In *2020 IEEE ICSA-C*, pages 49–50.

Oracle (2021). Oracle cloud function. https://www.oracle.com/cloud-native/functions/. [online; 11-Aug-2021].

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. *Commun. ACM*, 64(5):76–84.

Wen, J., Liu, Y., Chen, Z., Ma, Y., Wang, H., and Liu, X. (2021). Understanding characteristics of commodity serverless computing platforms.