Comparison of FaaS Platform Performance in Private Clouds

Marcelo Augusto Da Cruz Motta[a], Leonardo Reboucas De Carvalho[b],
Michel Junio Ferreira Rosa[c] and Aleteia Patricia Favacho De Araujo[d]
Department of Computing Science, University of Brasilia, Brasilia, Brazil

Keywords: FaaS, Function-as-a-Service, Private Cloud, OpenWhisk, Fission, OpenFaaS.

Abstract: Since its appearance in mid-2014, there has been notable growth in the adoption of cloud services via the Function-as-a-Service (FaaS) model, with several public cloud providers offering this model in their catalog. Various papers have mostly evaluated the performance of these models in public cloud environments. However, the implementation of this model in a private cloud environment has not been explored enough by the academic community. This article presents a two-level factorial design-based assessment of the main open-source operating platforms (Such as OpenWhisk, Fission, and OpenFaaS) currently available to serve as a comparative tool that can be used in decision-making processes. Results showed that regarding the stress test the OpenWhisk platform has greater reliability. On the other hand, regarding the processing of the Matrix, Factors, and Filesystem functions, Fission remains at similar behavior at all concurrency levels.

1 INTRODUCTION

Since the launch of the Function-as-a-service (FaaS) (Schleier-Smith et al., 2021) cloud service model in 2014 by AWS (Amazon Web Services) as AWS Lambda (Amazon Web Services, 2021), several providers have also added FaaS products to their catalogs. Initially used in tasks of low complexity, the model is quite lenient and soon came to expand its range of applicability. From this point on, the growth of this cloud service model has been quite representative, and it has been appointed as a candidate to become the predominant model adopted by users.

In this context, some academic papers such as (Carvalho and Araújo, 2019), (García López et al., 2018), (Malawski et al., 2020), have evaluated the performance and behavior of this model, using different types of metrics. However, a greater interest in the literature for the implementation of this service model in public clouds have been noticed, even considering the possibility of using FaaS technologies in private clouds. Given this scenario, this paper presents a performance analysis based on a factorial design considering the main open-source FaaS platforms currently available, which are: Fission (Fission, 2021), OpenFaaS (OpenFaaS, 2021) and OpenWhisk (Apache OpenWhisk, 2021) in order to identify the best performing and lowest cost tool, aiming its use in real world operations in private clouds.

An equivalent environment containing these tools were provisioned in Dataprev's [1] private cloud infrastructure to allow a comprehensive analysis on the quality and on the efficiency. To achieve this goal, corresponding workloads were planned and executed on each platform using the Apache JMeter (Halili, 2008) tool and some metrics were collected from them, allowing comparative analysis under different conditions. Results showed that the OpenFaaS platform obtained a more consistent response level in several tests regarding the network-bound function. When processing the Matrix function, The OpenWhisk platform reached a more consistent level at the beginning, being outperformed by Fission in large activations.

This paper is divided into six sections, this being an introductory section. Section 2 presents the concepts regarding Function-as-a-Service and the main platforms of this model. In Section 3, related papers are presented. Section 4 presents the methodology used in this paper, while Section 5 presents the results. Finally, Section 6 presents a conclusion and future work.

[a] https://orcid.org/0000-0002-4114-4735
[b] https://orcid.org/0000-0001-7459-281X
[c] https://orcid.org/0000-0002-0860-1834
[d] https://orcid.org/0000-0003-4645-6700

[1] Social Security Data Processing Company, a Brazilian state-owned company. https://www.dataprev.gov.br/

DOI: 10.5220/0011116700003200
In *Proceedings of the 12th International Conference on Cloud Computing and Services Science (CLOSER 2022)*, pages 109-120
ISBN: 978-989-758-570-8; ISSN: 2184-5042

2 FUNCTION-AS-A-SERVICE

Also called Serverless (Chapin and Roberts, 2017), or Backend-as-a-Service, (BaaS) (Castro et al., 2019) when used specifically to act as APIs, the Function-as-a-Service (FaaS) model suppliers give the users interfaces (graphical or command line) to allow the input and the manipulation of source code written in any of the supported programming languages, supervising the application processing using pre-configured triggers. These triggers can originate eighter from the provider's services, or through an API (Schleier-Smith et al., 2021).

Even if the fluctuation in the load exerted on the service is considered, it is expected that the provider guarantees that it can respond to the triggers demands, eventually making available more computational resources to contemplate the effective processing promptly, even in overload situations. Likewise, in situations of a low load, the provider must deallocate additional resources to minimize the operational cost of the platform transparently for its users.

Normally, the charging part in this service model is carried out through a policy based on the number of resources that the function has allocated, considering its execution time, thus making it possible to not charge for inactivity as occurs in traditional models of cloud services. Additionally, given its elastic characteristic, FaaS deals with the application's operational overloads without the need for user intervention. Because they do not share resources at runtime, FaaS services usually have limitations related to allocable resources per function (RAM, CPU, etc.) and maximum execution time. In scenarios where maintaining state is needed, it will also be necessary to use other resources from the provider for storage purposes.

Today's largest public cloud providers offer service options oriented to the FaaS model. AWS, a pioneer in this segment, offers the AWS Lambda (Amazon Web Services, 2021). The Google Cloud Platform also offers a FaaS service, Google Cloud Function (Google, 2021). Microsoft has Azure Functions (Microsoft, 2021) as its product and other providers like Oracle, IBM and Alibaba also have their FaaS model solutions. Each provider has its own strategy to operationally sustain these FaaS-based services. Google uses the Borg (Verma et al., 2015) container orchestrator to render its cloud services, including Cloud Functions. Borg was the basis for the development of Kubernetes[2] (Burns et al., 2016). AWS uses the Firecracker platform (Amazon, 2021) to orchestrate the microVms consumed by AWS Lambda. Microsoft Azure has adopted their own solution to manage the virtual machines that process the executions of its FaaS solution in Azure Functions.

[2]Kubernetes is an open-source container orchestrator used to automate the deployment, sizing and management of applications

However, there are cases when it is not possible to use public cloud solutions, such as the case of several agencies of the Brazilian government, which, for security reasons, need to house all the data and services they manage within their own infrastructure. Therefore, this study used open-source FaaS platforms in Dataprev's private cloud infrastructure, a Brazilian state-owned technology company that has Datacenters located in three large urban centers in Brazil (Brasília, Rio de Janeiro, and São Paulo).

2.1 Open-source FaaS Platforms

There are currently several open-source tools for FaaS. To describe the FaaS tools studied in this paper, we consider some fundamental criteria in a FaaS environment to analyze the degree of maturity of each one of the criteria such as scalability, language support, large-scale operation, community support and documentation.

In some cases, public providers choose to use open-source FaaS tools. An example would be IBM, which uses OpenWhisk (Djemame et al., 2020). Apache OpenWhisk is an open-source distributed serverless platform that executes functions in response to events, at any scale. OpenWhisk manages infrastructure, servers, and scaling using Docker containers. This platform supports the processing of several programming languages, which can be dynamically scheduled and executed in response to associated events (via triggers) from external sources (Feeds) or HTTP requests. The project also includes a REST API-based command-line interface (CLI) along with other tools to support packages, catalog services, and many popular container deployment options.

Oracle uses the Fn (Fn Project, 2021) for operational support of the Oracle Cloud Functions service. The Fn (Fn Project, 2021) project is an open-source container-native serverless platform that runs on public and private clouds. It supports all programming languages, it is extensible and has a high performance. Considering that Fn and OpenWhisk can operate in a large public provider, both currently have a considerable degree of maturity.

The Texas Advanced Computing Center - TACC uses Abaco (Stubbs et al., 2017) as its FaaS solution. TACC is a niche provider, focused on academic research. Abaco uses the Docker-compose orchestrator, which is not cluster-aware and therefore operates

Table 1: Open-source FaaS Tools.

Tool	Environments	Private provider	Supported Languages	Support - GitHub/Slack	Documen-tation	Maturity
Fission (Fission, 2021)	Kubernetes	NodeJS, Python, Go, Java, Ruby, Binary, PHP, .NET, .NET 2.0 and Perl	N/A	Yes	High	High
Fn (Fn Project, 2021)	Kubernetes	Numerous (Docker images)	Oracle	Yes	High	High
Knative (Knative, 2021)	Kubernetes	Numerous (Docker images)	N/A	Yes	High	High
OpenFaaS (Kaewkasi, 2018)	Kubernetes	Numerous (Docker images)	N/A	Yes	High	High
OpenWhisk (Djemame et al., 2020)	Kubernetes, OpenShift and Docker-compose	Go, Java, NodeJS, .NET, PHP, Python, Ruby, Rust, Scala,Swift, Ballerina and Deno	IBM	Yes	High	High
Kubeless (Kubeless, 2019)	Kubernetes	Java, NodeJS and VertX	N/A	No	Medium	Medium
Knix (Knix, 2021)	Kubernetes	Python and Java	N/A	Yes	Low	Medium
Abaco (Stubbs et al., 2017)	Docker-compose	Numerous (Docker images)	TACC	Yes	Low	Low
FuncX (Chard et al., 2019)	Specific Application	Python	N/A	Yes	Low	Low
TinyFaaS (Pfandzelter and Bermbach, 2020)	Docker	NodeJS	N/A	Yes	Low	Low

with scale constraints. Considering the characteristics of Abaco, especially its limited scale, it is possible to determine its maturity level as low for a scenario like FaaS.

In addition to the solutions used by public providers, there are other open-source tools available for the FaaS. KNIX MicroFunctions (formerly SAND) (Knix, 2021) is a high-performance open source serverless computing platform designed to minimize startup delays for function execution. This tool aims to provide support for persistent functions and optimize the use of resources. It has limited support for languages (Python and Java) and therefore its maturity level was considered medium.

Kubeless (Kubeless, 2019) is a tool that uses Kubernetes features to provide automatic scaling, API routing, monitoring, troubleshooting and more. It also has limited language support (Java, NodeJS and VertX) and therefore its maturity level will be considered medium in this paper.

FuncX (Chard et al., 2019) is a distributed FaaS platform that enables the execution of flexible, scalable, and high-performance remote functions. Unlike centralized FaaS platforms, FuncX allows users to perform functions on remote heterogeneous computers, from laptops to campus clusters, clouds, and supercomputers (Chard et al., 2019). The heterogeneity characteristic of FuncX is its great advantage over other tools. However, it only supports Python, making it limited in the general context of FaaS, and its maturity level was considered low.

Fission (Fission, 2021) is an open-source framework that runs on Kubernetes and is extensible to any programming language. In Fission, the core is written in the Go language and the specific parts of the languages are isolated in something called the lower environment. This platform maintains a pool of “hot” containers, each one with a small dynamic loader. When a function is called for the first time, that is, cold-initialized, a running container is chosen, and the function is loaded. This pool is what makes Fission fast, achieving cold start latencies of around 100 milliseconds. Thus, we can consider that Fission has a high degree of maturity.

Knative (Knative, 2021) allows developers to use an event-driven architecture. An event-driven architecture is based on the concept of decoupled relationships between event producers - that create events - and event consumers, or collectors, - that receive events. Knative’s support platform is Kubernetes, on which container image deployments are made. Due to its characteristics, Knative can be classified as having a high degree of maturity.

OpenFaaS (Kaewkasi, 2018) is an open-source tool that runs on Kubernetes and aims to make it easier for developers to implement event-driven functions and microservices. It provides a highly scalable endpoint with elasticity and automatic metrics. Considering that it works with container images, it can potentially work with all programming languages. This along with the support in Kubernetes, which is highly scalable, gives OpenFaaS a high degree of maturity.

TinyFaaS (Pfandzelter and Bermbach, 2020) is a lightweight FaaS platform for the edge environment with a focus on performance in constrained environments. Once a role is deployed to TinyFaaS, role handlers are created automatically. Additionally, TinyFaaS works on Docker containers. Considering that TinyFaaS only works with functions written in the NodeJS language and given its specific applicability, in this paper its maturity level was considered low.

Table 1 presents the main characteristics of open-source FaaS tools that can be considered for deployment in private cloud environments. Considering Table 1, five of them presented a high degree of maturity according to the established criteria, which are: Fission, Fn, Knative, OpenFaaS and OpenWhisk. However, other criteria such as dimensioning strategies, resource consumption, configurability, among others, can impact on the performance of solutions operated on this type of platform.

Therefore, this paper carried out an experiment to allow an evaluation of these characteristics in a real operational situation. The adopted methodology will be presented in Section 4.

3 RELATED WORKS

Going through the literature, the majority of papers are predominantly focused on the evaluation of FaaS platforms used in Public clouds. In (Copik et al., 2021) the authors propose the Serverless Benchmark Suite - SEBS, which consists of specifying representative workloads, monitoring the implementation, and evaluating the infrastructure. The abstract model of a FaaS implementation ensures the benchmark's applicability to various commercial vendors such as AWS, Azure, and Google Cloud. This paper evaluates aspects such as time, CPU, memory, I/O, code size, and cost based on the performed test cases. However, it does not support testing on open-source platforms that can be deployed in a private cloud environment.

The paper (Grambow et al., 2021) presents an application-centric benchmark framework for FaaS environments with a focus on evaluatiing realistic and typical use cases for FaaS applications. It has two built-in benchmarks (e-commerce and IoT), which are extensible for new workload profiles and new platforms. In addition, it supports federated benchmark testing, in which the benchmark application is distributed across multiple vendors and supports fine-grained analytics. The authors compare three public providers and analyze the characteristics of a federated fog configuration, and this is their main contribution. However, the extension of the sample spectrum of tests carried out between public providers and the cutting-edge approach used by the authors is limited, given the use of only one of the supported platforms (TinyFaaS) as an example of a federation, rather than the other two platforms (OpenFaaS and OpenWhisk), which are considerably more robust.

In the paper (Jindal et al., 2021), an extension of the concept of FaaS as a programming interface for heterogeneous clusters and to support heterogeneous functions with diverse computational and data requirements is presented. This extension is a network of distributed heterogeneous destination platforms equivalent to content delivery networks, widely used in other computational cloud models. In this proposal, a target platform is a combination of a cluster of homogeneous nodes and a FaaS platform on top of it. In this context, metrics such as requests, CPU, activations, and response times are evaluated from functions adapted from the FaaSprofiler (Shahrad et al., 2019) benchmark. Despite using an open-source platform suitable for deployment in a private cloud environment, the study was limited to just two representatives of these tools, although there are others whose level of maturity would allow them to be part of the list of FaaS operating platforms adopted by the tool.

In (Wen et al., 2021), the authors perform a detailed evaluation of FaaS services: AWS, Azure, GCP, and Alibaba, running a test flow using microbenchmarks (CPU, memory, I/O, and network) and macro benchmarks (multimedia, map- Reduce and machine learning). The tests used specific functions written in Python, NojeJS, and Java that explored the properties involved in the benchmark to assess startup latency and resource usage efficiency. However, all platforms evaluated are deployed in public clouds and as the underlying infrastructure in these services remains unclear to the customer, the evaluation is restricted to the overall approach taken by the provider. Thus, the assessment of the architectural strategy does not consider the platform in isolation.

vHive (Ustiugov et al., 2021) aims to enable FaaS researchers to innovate in the deeply distributed software stacks of a modern FaaS platform. vHive is designed to support leading FaaS vendors by integrating the same production-grade components used by vendors, including the AWS Firecracker hypervisor, Cloud Native Computing Foundation Containerd, and Kubernetes. vHive adopts the Knative platform, allowing researchers to quickly deploy and test any serverless application that can include many func-

Table 2: Related Works.

Study	Year	Private provider	Maturity analysis	Approach	Metrics	Languages
(Maissen et al., 2020)	2020	No	No	Specific functions with multilingual	Time and latency	Python, NodeJS, Go and .NET
(Copik et al., 2021)	2020	No	No	Specific functions with multilingual	Time, CPU, memory, I/O, code size and cost	Python and NodeJS
(Wen et al., 2021)	2020	No	No	Specific functions with multilingual	Startup latency and resource efficiency	Python, NojeJS and Java
(Grambow et al., 2021)	2021	OpenFaaS, OpenWhisk and TinyFaaS	No	Realistic functions	Network traffic	JavaScript
(Ustiugov et al., 2021)	2021	Firecracker	No	Specific functions and use of snapshots	Delay on cold start	Python
(Jindal et al., 2021)	2021	OpenWhisk and OpenFaaS	No	FaaSprofiler adapted functions and heterogeneous multicenter platforms	Requests, CPU, activations and time	Python
This paper	2022	OpenWhisk, OpenFaaS and Fission	Yes	Specific functions	Time, CPU, I/O and Latency	Python

tions, running on secure Firecracker microVMs, as well as full server services. Both stateful functions and services can be deployed using OCI / Docker images. vHive empowers system researchers to innovate on key serverless features, including automatic role sizing and cold boot delay optimization with multiple snapshot engines.

The article (Maissen et al., 2020), introduces FaaS-dom, a modular and extensible set of benchmarks for evaluating serverless computing that includes a range of workloads and natively supports the leading FaaS providers (AWS, Azure, Google, and IBM). The great contribution of FaaS-dom consists of the functions developed for execution during the tests, serving as a basis for conducting this paper. Although it allows the evaluation of the IBM solution, which internally uses an open-source platform, the evaluation in FaaS-dom suppresses equity aspects between the environments and, ends up focusing only on the evaluation of the provider, and not on the strategies used by the platform.

Table 2 presents the main characteristics identified in each related work described above. This paper, on the other hand, explores tools that currently have a high degree of maturity as FaaS solutions which are available to deploy in private clouds, analyzing their behavior by applying workloads from the benchmark FaaSDom (Maissen et al., 2020), to identify the best performing platform for the following cases: the scenario of CPU-bound, I/O-bound and Network-bound functions.

4 METHODOLOGY

The open-source FaaS platforms which were previously classified as having a high degree of maturity were implemented in a private cloud environment. Although Fn and Knative were classified in this list, due to difficulties arising from the on-premise environment, it was not possible to reach their ideal configuration for a fair comparison with other platforms, leaving only OpenFaaS, Fission, and OpenWhisk as the focus of this paper. In OpenWhisk's case, all it took was a subtle increase in the pre-configured default limits on the platform.

The Kubernetes platform was chosen to define all the tools to be compared. This allows the experiment to focus on the strategies adopted by the tool, preventing platform differences from interfering with the results. Platform installations followed the manuals offered by the vendors and it was necessary to increase the timeout of functions in the OpenFaaS and OpenWhisk's cases. Their activation limits per minute were increased in order to equalize the three analyzed platforms. No changes were made in default configuration of Fission.

The infrastructure used in this experiment was deployed on Dataprev's private cloud solution. In this cloud it is possible to provide customized configurations of memory, CPU, disk, among others. Table 3 shows the values for each resource that was used to provision the Kubernetes clusters which support each platform applied in this experiment. After the platforms were implemented, four functions were published in each of them, that have already been referenced in the literature as source codes that perform operations that allow evaluating the behavior of the environments.

Table 3: Clusters Parameters.

Number of servers	3
Operational system	RHEL 8.3
Storage	30GB
RAM memory	4GB
CPU	4 (2.5Ghz)
Physical machine	Power Egde R900 - Intel Xeon E7 4870
Hypervisor	VMware ESXi 6
Docker	v20.10.7
Kubernetes	v1.21.3
Apache JMeter	v5.4.1

For this paper, the Python functions proposed by FaaS-dom (Maissen et al., 2020) selected were: Latency (Network-bound), Matrix (CPU-bound), Factors (CPU-bound) and Filesystem (I/O-bound). According to its documentation, the *Latency* function measures the latency of a simple function, while the *Factors* function calculates the factors of a number iteratively to assess CPU performance. The *Matrix* function multiplies two *NxN* matrices iteratively also to assess CPU performance and the *Filesystem* function writes and reads *n* times *a x kB* file to the filesystem.

In order to avoid any confusion with the nomenclature of tests with the *Latency* function adapted from (Maissen et al., 2020), in this paper we will refer to it as the *Delay* function. After each function is published on each platform, URLs are made available and used to activate them. It was used five common concurrencies: 1, 2, 4, 8 and 16 simultaneous requests. The tests were performed using the JMeter (Halili, 2008) tool, which runs test batteries, collects the metrics, and generates the results for analysis.

Figure 1 shows the architecture used in the experiment. It is possible to see the individualization of each platform with its Kubernetes cluster using the same number of computational resources and the same configuration for all FaaS tools. Each test was repeated 10 times. However, due to the cold start effect inherent in the FaaS platforms, the three platforms were previously triggered in each of the functions.

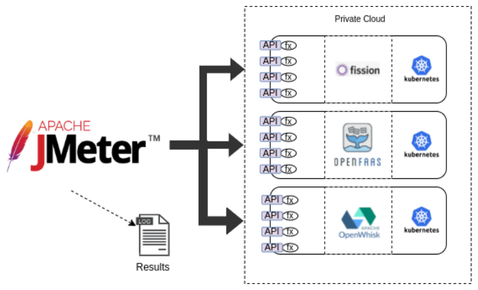

Figure 1: Experiment architecture.

A full factorial design utilizes every possible combination at all levels of all factors. A performance study with *k* factors, with the *i*th factor having n_i levels, requires *n* experiments (Jain, 1991). The advantage of a full factorial design is that every possible combination of configuration and workload is examined. It is possible to find the effect of every factor including the secondary factors and their interactions. The main problem is the cost of the study. It would take too much time and money to conduct these many experiments, especially when considering the possibility that each of these experiments may have to be repeated several times (Jain, 1991). Through the execution of a factorial design, it is possible to obtain the percentage portions of the effects of each factor in the result of the experiment, allowing the identification of the effects with a greater degree of interference in the results and thus helping decision-making processes. In factorial designs with several repetitions, it is possible to obtain the sampling error effect. High sampling error rates indicate the existence of factors not mapped in the factorial design and this is a major aspect that helps in the decision-making process.

In order to identify which of the factors exerted the greatest influence on the results after performing the tests, the factorial design 2^k was used to start the analysis of each comparison scenario between the platforms used. In each factorial design, the factors analyzed were: provider, concurrency, and function. Factorial design factors and their levels are shown in Table 4.

In order to observe the behavior of platforms in extreme situations, a stress test was designed using the Matrix function. In this test, platforms are subjected to exponential levels of Matrix function concurrency until they are unable to respond. The results of factorial designs 2^k, as well as the other results of the experiment, are discussed in detail in Section 5.

Table 4: Factorial Design Factors. Openwhisk (1), OpenFaaS (2) and Fission (3).

Factor (concurrence)	Function	Platforms		
Lower (1)	Matrix	1	1	2
Upper (16)	Delay	3	2	3
Lower (1)	Delay	1	1	2
Upper (16)	Factors	3	2	3
Lower (1)	Delay	1	1	2
Upper (16)	Filesystem	3	2	3
Lower (1)	Factors	1	1	2
Upper (16)	Filesystem	3	2	3
Lower (1)	Matrix	1	1	2
Upper (16)	Factors	3	2	3
Lower (1)	Matrix	1	1	2
Upper (16)	Filesystem	3	2	3

5 RESULTS

This section was structured according to the analyzes carried out in the tests performed on each platform.

5.1 Latency Analysis

Through the records obtained by the *JMeter* tool during the tests, latency values were generated for all levels of concurrence. In the execution without concurrency (with the request only), ten result values were generated for each function on each platform, for concurrency level two, 20 result values, for concurrency level four, 40, for concurrency level eigth, 80, and concurrency sixteen, 160 latency values for each function.

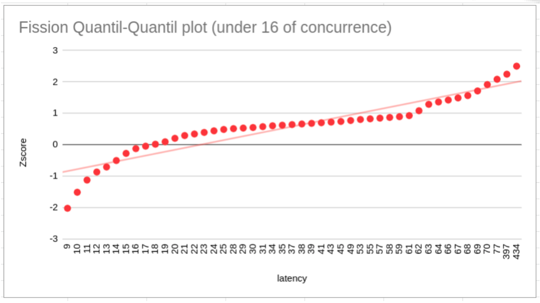

Figure 2: Quantil-quantil plot for Fission under 16 of concurrence.

In order to adequately represent the samples in each execution block and allow a consolidated analysis of the data, it was necessary, using the analysis of quantile-quantile graphs, to define a value for this representation. Figure 2 represents all executions of one of the Fission functions in 16 simultaneities (160 latency values), as well as Figure 3 for OpenFaaS, and Figure 4 for OpenWhisk.

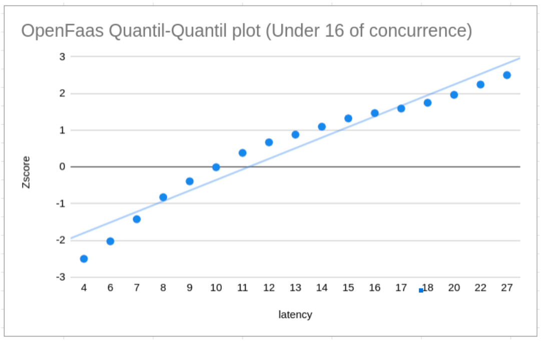

Figure 3: Quantil-quantil plot for OpenFaaS under 16 of concurrence.

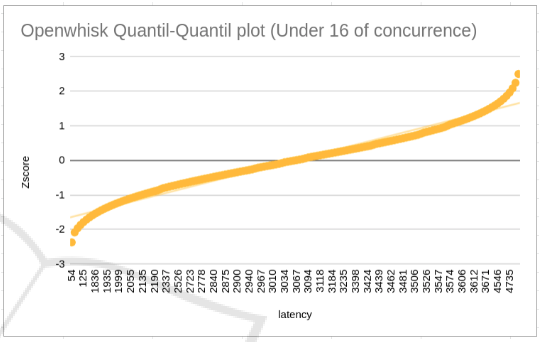

Figure 4: Quantil-quantil plot for OpenWhisk under 16 of concurrence.

In all cases, it is found that the latency values do not completely follow a normal distribution, as the shape of the graphs does not look like a straight line following a trendline. Thus, the average was adopted as a representative metric of the samples in each concurrent run, better representing the data with dispersion characteristics.

After determining the best representative metric for the latency records, we calculated the average of all simultaneous executions according to Table 5, where it is also possible to verify the Log_{10} values calculated by the average obtained in every run. Thus, the metric chosen to represent the runs was once again the average, as shown in Table 5, and they were used to build the graphs with the evolution of the execution of each function on each platform.

Figure 5 shows the evolution of Matrix function execution at each level of competition on both platforms. It is noteworthy that Figure 5 shows the averages after undergoing a logarithmic transformation to provide a better view of the information. It is possible to observe that, without concurrency, all platforms present equivalent results for latency. However, as the concurrency level increases, while Fission maintains the latency results stable, OpenFaaS make a noticeable upscale with concurrency of two and OpenWhisk

Table 5: Consolidated Delay X Matrix X Factors X Filesystem results.

Platform	Concur.	Function							
		Delay		Matrix		Factors		Filesystem	
		Run Avg.	Log$_{10}$	Run Avg.	Log10	Run Avg.	Log$_{10}$	Run Avg.	Log$_{10}$
Fission	16	30,68	1,49	889,91	2,95	17589,30	4,25	27677,16	4,44
	8	26,39	1,42	547,56	2,74	14820,28	4,17	9001,76	3,95
	4	23,80	1,38	443,53	2,65	13414,83	4,13	5332,23	3,73
	2	34,65	1,54	423,50	2,63	12781,80	4,11	4785,10	3,68
	1	80,50	1,91	1,00	446,30	12762,50	4,11	4788,40	3,68
OpenFaaS	16	10,63	1,03	3518,90	3,55	30479,75	4,48	27124,71	4,43
	8	11,84	1,07	1907,26	3,28	33404,96	4,52	24423,61	4,39
	4	10,85	1,04	896,65	2,95	38854,73	4,59	26730,43	4,43
	2	11,50	1,06	371,55	2,57	18433,90	4,27	9373,70	3,97
	1	14,10	1,15	393,00	2,59	9084,00	3,96	4617,00	3,66
OpenWhisk	16	2951,51	3,47	3974,74	3,60	23349,06	4,37	12842,34	4,11
	8	128,29	2,11	448,53	2,65	13465,26	4,13	10784,60	4,03
	4	34,90	1,54	391,60	2,59	9582,50	3,98	4573,18	3,66
	2	52,80	1,72	500,90	2,70	8890,80	3,95	5074,35	3,71
	1	71,10	1,85	332,70	2,52	8963,00	3,95	3950,80	3,60

with concurrency of eight. This demonstrates that Fission was able to process the Matrix function more efficiently than OpenWhisk and OpenFaaS.

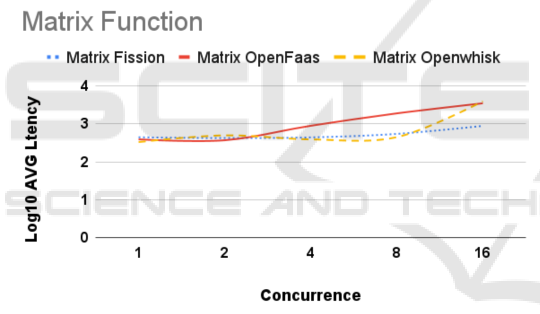

Figure 5: Matrix function results.

With the averages also logarithmically transformed, Figure 6 demonstrates the evolution of latency in the execution of the Delay function on each platform over each level of concurrency.

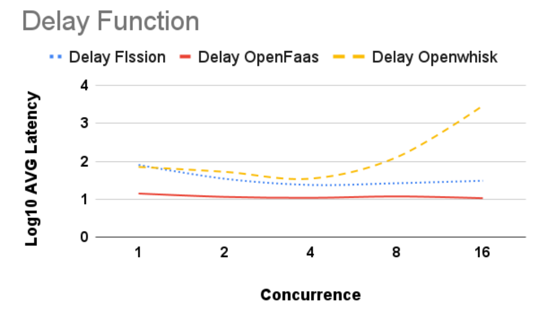

Figure 6: Delay function results.

Here we can see that when running without concurrency, OpenFaaS obtained a much lower latency result than the value obtained by Fission, followed slightly by OpenWhisk, and, as the concurrency level increases, OpenWhisk increase the distance for Fission and OpenFaaS. After superior levels, however, Fission experiences a slight decrease in latency, whereas OpenWhisk has a considerable growth. It is possible to infer that OpenFaaS was more efficient than the other platforms in processing the Latency function at all concurrency levels.

The results have shown that regarding Delay function, OpenFaaS stands out in efficiency, followed by Fission and then by OpenWhisk, whereas as to Matrix function (which requires more processing), OpenWhisk in executions with concurrency up to eight shows lower efficiency concerning latency, being surpassed by Fission after executions with concurrence ten. OpenFaaS, on the other hand, decreases its efficiency under concurrence of four.

Figure 7 demonstrates the evolution of latency in the execution of the Factors function on each platform over each level of concurrency.

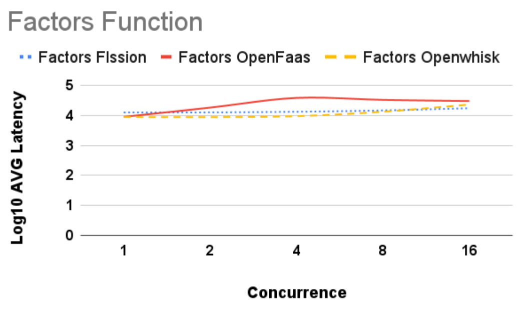

Figure 7: Factors function results.

OpenWhisk has the lowest latency record without concurrency, followed by OpenFaaS and Fission. However, as the number of concurrencies increase, OpenWhisk and Fission demonstrate consistency in their latency levels. On the other hand, OpenFaaS demonstrates an increase from tests with concurrency of two.

Also logarithmically transformed, Figure 8 demonstrates the evolution of latency in the execution of the Filesystem function on each platform at all concurrency levels.

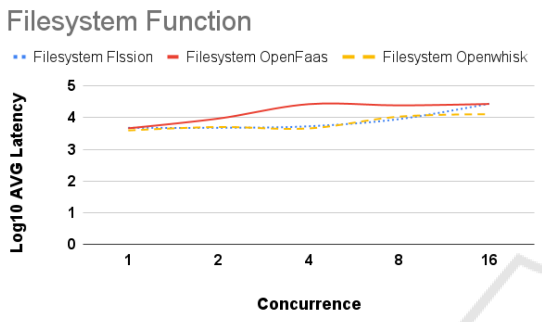

Figure 8: Filesystem function results.

When disk-related resources are used, we can see that OpenWhisk has the lowest latency record in the first test (no concurrency), followed by OpenFaaS and Fission with slightly higher latency and showing extreme similarity in the initial tests. Given the increase in concurrency, OpenFaaS experiences greater growth than other platforms already with two and four concurrency, remaining constant at eight and 16. Fission and OpenWhisk maintain their latency levels up to the competition of four concurrencies, achieving slight growth in both cases. With concurrency level 16, Openwhisk demonstrated the best records among the analyzed platforms.

However, to identify which factor has the greatest influence on the results, a factor analysis was performed using six factorial designs 2^k.

5.2 Factorial Designs

Table 6 presents the factorial planning used in both cases, and it is possible to observe the eight tests performed, as well as the effects and their interrelationships. Lower levels are denoted by -1, while upper levels are denoted by 1. All data used for average extractions according to the effects of each experiment performed in each of the ten repetitions considered in this experiment, as well as the sum of the difference between each average and the mean square and the partial calculating the sum of squared errors in this experiment for every test (on every row) are available on GitHub.[3]

Table 6: Factorial Design Planning.

Test	P	C	F	PxC	PxF	CxF	PxFxC
1	-1	-1	-1	1	1	1	-1
2	1	-1	-1	-1	-1	1	1
3	-1	1	-1	-1	1	-1	1
4	1	1	-1	1	-1	-1	-1
5	-1	-1	1	1	-1	-1	1
6	1	-1	1	-1	1	-1	-1
7	-1	1	1	-1	-1	1	-1
8	1	1	1	1	1	1	1

In each case, platforms, functions and concurrency were classified according to Tables 6 and 4, thus calculating the effects, variations, the RSS - Residual Sum of Squares, SSY - Sum of the square of Y and SST - Sum of the Total Square, allowing to obtain the value of the fractions of each factor and their relations in each result.

5.3 Factorial Design Delay-matrix

Figure 9 shows that, in the comparison between OpenFaaS and Fission platforms, the effect of the function factor is 26.35% on the results of the experiments, in relation to the concurrency factor, in the case of OpenWhisk and OpenFaaS it reaches 20% and in the case of OpenWhisk and Fission the Function factors 81.35%. Considering the scenario of the Delay and Matrix functions, it is possible to conclude that, in this case, the Function factor has the greatest effect on the results of the tests performed.

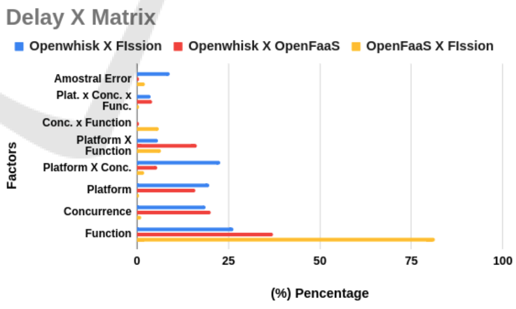

Figure 9: Factorial design results - Delay X Matrix.

5.4 Factorial Design Delay-factors

Analyzing Figure 10, regarding the Delay and Factors functions, all three comparisons between the platforms analyzed present high levels of fraction for the Function factor. OpenWhisk and Fission 76.14%, OpenWhisk and OpenFaaS 72.73% and OpenWhisk and Fission 96.25%.

[3]https://github.com/unb-faas/private-platform-benchmark

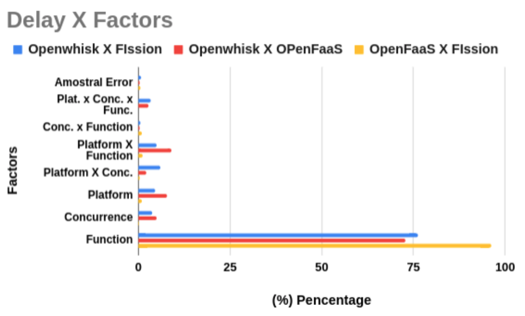

Figure 10: Factorial design results - Delay X Factors.

Like previous analysis, when we correlate the Delay and Matrix functions, the function factor has the largest representation in the results, with some distance to the second larger.

5.5 Factorial Design Delay-filesystem

The correlation between the Delay and Matrix functions calculated via the factorial plane again shows that the Function factor was predominant in this phase of the analysis, with correlation of OpenFaaS and Fission obtaining the fraction of 68.90%, OpenWhisk and OpenFaaS with 68.84% followed by OpenWhisk and Fission with 93.18%. In the competition factor, the three platforms also show similar behavior, with fractions under 10%. Again, like the two previous analyses, the function factor has a greater representation in the results in all three Platforms.

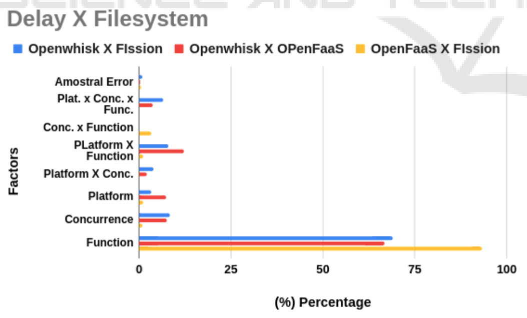

Figure 11: Factorial design results - Delay X Filesystem.

5.6 Factorial Design Factors-filesystem

At this stage, representative results were found using other functions. When analyzing the correlation between the Matrix and Factors functions in Figure 12, the Factor function is no longer as representative as seen before, reaching only the fraction of 5.21% in OpenFaaS and Fission, in OpenWhisk and OpenFaaS 6.84%, and in OpenWhisk and Fission 13.5%. In this new scenario, it is possible to see the predominance of the concurrence factor on the three platforms, with OpenFaaS and Fission 62.37%, OpenWhisk and OpenFaaS 84.95% and OpenWhisk and Fission 76.58%.

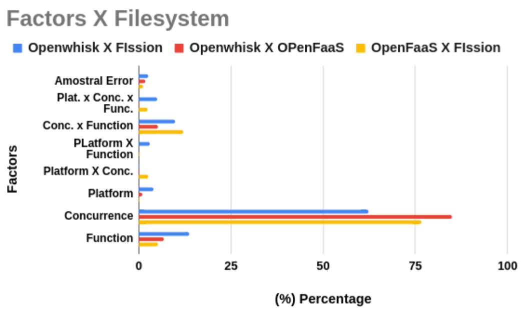

Figure 12: Factorial design results - Factors X Filesystem.

Considering the use that functions related to CPU consumption and I/O, we realized that concurrence has the predominant fraction in the results of the three correlated platforms.

5.7 Factorial Design Matrix-factors

In the analysis of the correlation between the Matrix and Factors functions, the most representative factor was again the Function. In the Figure 13 it is possible to see the fractions of 78.89% between OpenWhisk with Fission, 66.66% between OpenFaaS and Fission, and 80.77% between OpenWhisk with OpenFaaS.

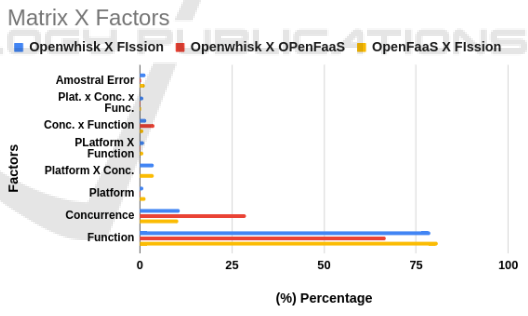

Figure 13: Factorial design results - Matrix X Factors.

Regarding concurrency, in all three scenarios analyzed for the two functions on the three platforms, its factor fraction values average from 10% to 28.8%. Considering the correlation of the functions analyzed in this case, the results again allow to infer that the function factor predominates in the of the results.

5.8 Factorial Design Matrix-filesystem

As to the correlation of the Matrix and Filesystem functions, they obtained two factors with considerable representativeness. The Function and Concurrency factors, in this case, have representative values.

In the function factor for the OpenFaaS and Fission correlation the fraction of 63.95% influences the results, whereas for the concurrency factor is 24.49%. In the case of OpenWhisk and OpenFaaS correlation, 50.71% for Function and 44.49% for concurrency. Considering OpenWhisk and Fission, 68.09% for function and 24.18% for concurrency.

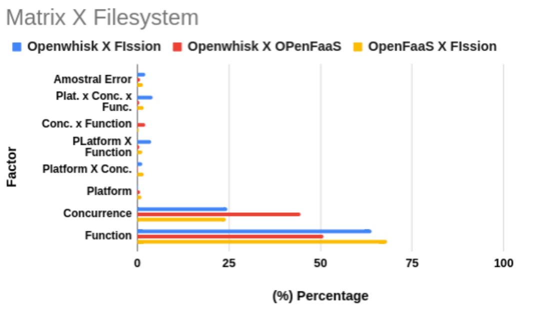

Figure 14: Factorial design results - Matrix X Filesystem.

Considering the average sampling error of only 1.19% in the 18 scenarios calculated in the experiment, it is possible to have a more detailed view of the factors that influence the results.

5.9 Factorial Design Results

In the studied cases, the influence of the Function and Concurrency factors was predominantly representative of the latency results obtained through the tests performed in the functions on each platform.

5.10 Reliability

In the stress test performed with the Matrix function, only Openwhisk was able to reach the limit of 16,384 simultaneous requests before starting to fail, followed by OpenFaas with approximately 1024 requests and Fission, with 512 requests. Figure 15 shows the degree of reliability of each platform calculated based on the percentage of satisfied requests.

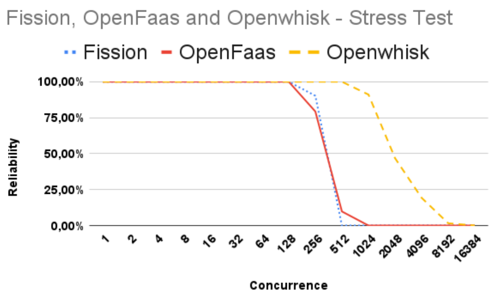

Figure 15: Reliability.

It is possible to observe that up to 128 simultaneous requests, the three platforms maintain the maximum level of reliability, fulfilling 100% of requests. At near 512 simultaneous requests, OpenFaaS starts to suffer a gradual drop in its reliability rate and, at this concurrency, Fission also shows a drop in the success percentage. Although OpenFaaS registers some level of reliability between 512 and 1024 concurrent requests, its reliability is very low and can be considered irrelevant. OpenWhisk, on the other hand, demonstrates greater reliability among the three platforms, showing an error in aproximately 10% of the requests only after a load greater than 1024 simultaneous requests.

6 CONCLUSIONS AND FUTURE WORK

In this paper, an experiment carried out in a private cloud environment to evaluate the FaaS Fission, OpenFaaS and OpenWhisk platforms under different concurrency conditions.

In case of the Delay function with a concurrency of 16, the results showed that the OpenFaaS platform performed better in comparison to latency during the tests, managing to keep its latency results at similar levels even under the first five concurrency levels, while Fission and OpenWhisk experienced linear latency variation in the tests. However, when the Matrix function was also analyzed under simultaneity of 16, the Fission platform kept its latency levels constant, while OpenFaaS and OpenWhisk showed slight growth in latency due to increased concurrency.

In the best factorial design applied to the results of the experiment, it was verified that the results obtained were influenced by 96.25% by the function factor, while the other factors were around 0.85%, and the relationships between the factors influenced between 0.01% and 1.13% each.

In future papers, considering the relevance of this factor in the results presented, other functions should be included in this experiment to provide a more realistic analysis, using real case examples in systems with micro-services architecture eligible for Dataprev. This experiment can also be carried out on Kubernetes platforms that have a greater number of resources. It would also be possible to add other solutions such as Fn (Fn Project, 2021) and KNative (Knative, 2021), establishing a customized configuration in each platform that offers equal conditions between the tools that allow an adequate comparison.

REFERENCES

Amazon (2021). Firecracker. https://firecracker-microvm .github.io/. [Online; accessed 10-August-2021].

Amazon Web Services (2021). AWS lambda.

Apache OpenWhisk (2021). Open source serverless cloud platform.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57.

Carvalho, L. and Araújo, A. P. F. (2019). Framework node2faas: Automatic nodejs application converter for function as a service. In *Proceedings of the 9th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, pages 271–278. INSTICC, SciTePress.

Castro, P., Ishakian, V., Muthusamy, V., and Slominski, A. (2019). The rise of serverless computing. *Commun. ACM*, 62(12):44–54.

Chapin, J. and Roberts, M. (2017). *What is Serverless*. Oreilly.

Chard, R., Skluzacek, T. J., Li, Z., Babuji, Y., Woodard, A., Blaiszik, B., Tuecke, S., Foster, I., and Chard, K. (2019). Serverless supercomputing: High performance function as a service for science.

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

Djemame, K., Parker, M., and Datsev, D. (2020). Open-source serverless architectures: an evaluation of apache openwhisk. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 329–335.

Fission (2021). Open source, kubernetes-native serverless framework.

Fn Project (2021). Open source. container-native. serverless platform.

García López, P., Sánchez-Artigas, M., París, G., Barcelona Pons, D., Ruiz Ollobarren, A., and Arroyo Pinto, D. (2018). Comparison of faas orchestration systems. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 148–153.

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

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

Jindal, A., Gerndt, M., Chadha, M., Podolskiy, V., and Chen, P. (2021). Function delivery network: Extending serverless computing for heterogeneous platforms. *Software: Practice and Experience*.

Kaewkasi, C. (2018). *Docker for Serverless Applications: Containerize and Orchestrate Functions Using OpenFaas, OpenWhisk, and Fn*. Packt Publishing.

Knative (2021). Knative - enterprise-grade serverless on your own terms.

Knix (2021). Knix: A high-performance, open-source serverless computing platform.

Kubeless (2019). Kubeless - the kubernetes native serverless framework: Build advanced applications with faas on top of kubernetes.

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.

Microsoft (2021). Azure functions.

OpenFaaS (2021). Serverless functions, made simple.

Pfandzelter, T. and Bermbach, D. (2020). tinyfaas: A lightweight faas platform for edge environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*, pages 17–24.

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.

Shahrad, M., Balkind, J., and Wentzlaff, D. (2019). Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 1063–1075, New York, NY, USA. Association for Computing Machinery.

Stubbs, J., Dooley, R., and Vaughn, M. (2017). Containers-as-a-service via the Actor Model. In *Gateways 2016 proceedings*.

Ustiugov, D., Petrov, P., Kogias, M., Bugnion, E., and Grot, B. (2021). Benchmarking, analysis, and optimization of serverless function snapshots. *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. (2015). Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA. Association for Computing Machinery.

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