FaaSdom: A Benchmark Suite for Serverless Computing

Anonymous Author(s)
Submission Id: 38

ABSTRACT

Serverless computing has become a major trend among cloud providers. With serverless computing, developers fully delegate the task of managing the servers, dynamically allocating the required resources, as well as handling availability and fault-tolerance matters to the cloud provider. In doing so, developers can solely focus on the application logic of their software, which is then deployed and completely managed in the cloud.

Despite its increasing popularity, not much is known regarding the actual system performance achievable on the currently available serverless platforms. Specifically, it is cumbersome to benchmark such systems in a language- or runtime-independent manner. Instead, one must resort to a full application deployment, to later take informed decisions on the most convenient solution along several dimensions, including performance and economic costs.

FaaSdom is a modular architecture and proof-of-concept implementation of a benchmark suite for serverless computing platforms. It currently supports the current mainstream serverless cloud providers (*i.e.*, AWS, Azure, Google, IBM), a large set of benchmark tests and a variety of implementation languages. The suite fully automatizes the deployment, execution and clean-up of such tests, providing insights (including historical) on the performance observed by serverless applications. FaaSdom also integrates a model to estimate budget costs for deployments across the supported providers. FaaSdom is open-source and available at https://github.com/faasdom/benchmark-suite-serverless-computing.

KEYWORDS

serverless, function as a service, benchmark suite, open-source

1 INTRODUCTION

The serverless computing paradigm is an emerging approach for developing cloud-based applications [11, 12, 48]. IBM [29] defines it as *"an approach to computing that offloads responsibility for common infrastructure management tasks (scaling, scheduling, patching, provisioning,* etc.*) to cloud providers and tools, allowing engineers to focus their time and effort on the business logic specific to their applications or process"*.

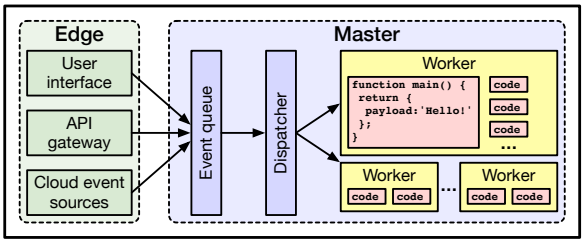


Figure 1: Typical FaaS architecture. The main aspects to benchmark are: performance of a worker (*i.e.*, execution speed of a function) and quality of the auto-scaling mechanism (*e.g.*, allocation of new VMs, how to handle flash-crowds [30]).

Serverless computing requires less expertise than other self-managed approaches. Users do not manage directly the infrastructure and runtime of the system, but instead delegate its operations to the cloud provider. Additionally, cloud providers can deploy finer-grain billing policies (*e.g.*, on a per service-call basis) for any of the offered services [10, 43], generally leading to reduced costs for developers.

One can distinguish between various serverless paradigms: *(1)* FaaS (*function as a service*, and focus of our work) implemented for instance by AWS Lambda [4], *(2)* DBaaS (*database as a service*), as available through Microsoft Azure for PostgreSQL [35]; and *(3)* STaaS (*storage as a service*), via Google Cloud [23]. FaaS can be considered an hybrid between Platform as a Service (PaaS) and the Software as a Service (SaaS) service model: data and infrastructure are fully managed by the provider, while the application is handled by the user. Figure 1 illustrates a typical FaaS infrastructure. In the typical FaaS approach, developers bypass the setup, maintenance and management of a compute node (*i.e.*, bare metal, virtual machines, or even containers). Instead, users provide the application code for specific *functions* to be deployed to the cloud. Specific events (*e.g.*, HTTP requests, storage or database conditions) trigger their execution, typically implementing data processing [4, 16]. The provider then handles the load, as well as availability and scaling requirements. Despite the convenience of the approach, it is currently hard to decide on a specific FaaS provider based on criteria such as performance, workload adaptability or costs.

This paper introduces FaaSdom, a testing suite that tackles this problem. In a nutshell, application developers can

use the FaaSdom suite to automatically deploy several performance tests a set of FaaS providers. The results can then be easily compared along several dimensions. Based on this information, deployers can evaluate the "FaaS-domness" of the providers and decide which one is best adapted for their applications, in terms of performance, reliability or cost.

While few efforts exist to benchmark serverless computing [13, 32–34], they are relatively limited in terms of supported providers, comparison metrics, diversity of benchmarks (*e.g.*, user-defined functions), or operating modes (*e.g.*, stress-test vs. continuous monitoring). Similarly, studies on benchmarking cloud platforms lack an in-depth evaluation of serverless computing platforms [14].

In this paper, we introduce FaaSdom, a modular and extensible benchmark suite for evaluating serverless computing. FaaSdom natively supports major FaaS providers (AWS, Azure, Google, IBM) but can be easily extended to benchmark additional platforms. It includes a wide range of workloads, including user-defined functions, and implementation languages. FaaSdom uses several metrics to compare FaaS platforms against multiple dimensions, in terms of latency, throughput and operating costs. It provides a Web-based interface that allows users to perform benchmarks in a fully automatic way (including deployment, execution and clean-up of the tests), and keeps track of historical data. In that way, FaaSdom can be used both for one-shot benchmarks and for continuous monitoring over time of the providers. FaaSdom is the first system to support such in-depth, comprehensive and extensible benchmarking of serverless computing.

The remainder of this paper is organised as follows. We first introduce background concepts (§2) and the supported frameworks (§3). We then describe the FaaSdom architecture (§4) and its different benchmarks (§5). We present and discuss evaluation results (§6), before concluding with a summary of lessons learned (§7) and open perspectives(§8).

2 BACKGROUND

This section provides technical details about the four major mainstream serverless providers, namely Amazon Web Services (§2.1), Microsoft Azure (§2.2), Google Cloud (§2.3) and IBM Cloud (§2.4). We compare the performance of all of them in our evaluation (§6).

2.1 Amazon Web Services Lambda

AWS Lambda [4] was released in November in 2014 [7]. AWS Lambda spans 18 geographical regions, plus China [9]. At the time of writing, it supports six different runtime systems and seven different programming languages [8]. Depending on the region where the function is deployed, Lambda supports up to 3,000 instances to serve the user functions [6]. The memory allocated to a function instance can vary from 128 MB up to 3,008 MB in steps of 64 MB [5]. The CPU power increases linearly with its memory allocation. For instance, according to the documentation, at 1,792 MB the function will get 1 vCPU [5].

As observed in [49], Lambda executes functions using two different CPUs, namely Intel Xeon E5-2666 clocked at 2.90 GHz and Intel Xeon E5-2680, clocked at 2.80 GHz.

2.2 Microsoft Azure Functions

Microsoft Azure Functions [36] was released publicly in November 2016 [31]. It supports five runtime systems and seven different programming languages [42]. In contrast with the other three cloud providers, Azure offers three different hosting plans [39]: Azure Functions offers billing plans that adapt to the load and popularity of the deployed function (*"Consumption"* plan), plans with finer-grain control over the computing instance size and pre-warming support (*"Premium"* plan), and a billing plan customized on a given application needs (*"App Service"* plan). This work only considers the consumption plan (generation 2.0 [38]), as it is the only one to be fully managed by the cloud provider and the most similar in terms of features to the plans from alternative providers.

Azure Functions can use as many as 200 instances and up to 1.5 GB memory [39]. The service can run either on Windows or Linux hosts, and is offered in 28 out of 46 publicly accessible regions [41]. Note that the consumption plan is only available in 11 regions for both Linux and Windows, hence we restrict our deployment to those in our experiments. Computing nodes can be characterized by their Azure Compute Unit (ACU), with 100 ACU roughly mapped to 1 vCPU. According to our investigations, we believe Azure Functions to be executed by virtual machines of type *Av2*.[1] These VMs use three different CPUs: Intel Xeon 8171M at 2.1 GHz, Intel Xeon E5-2673 v4 at 2.3 GHz and Intel Xeon E5-2673 v3 at 2.4 GHz [40].

2.3 Google Cloud Functions

Google Functions [16] was released on July in 2018 [19] and is available through seven out of the twenty Google regions [21]. It currently only supports three programming languages [20], namely Node.js, Python and Go. While there is not a maximum number of allocated instances per single function, it only allows up to 1,000 functions to be executed concurrently [18]. Table 2 summarizes the options for CPU and memory combinations supported by the platform [17]. While the official documentation lacks details on the exact CPU models, a quick inspection of `/proc/cpuinfo` unveils certain details, such as `vendor_id`, `cpu_family` and `model`. We only identified Intel-based processors during our experiments.

[1] https://docs.microsoft.com/de-ch/azure/virtual-machines/av2-series

Table 1: Runtime systems supported by mainstream FaaS providers ([2,3]: generation of Azure Functions; [†]: deprecated; [β]: beta; [#]: only in *App Service* or *Premium* plans).

		Node.js				Python				Go		.NET Core			Java		Ruby		Swift	PHP	Docker
		6.x	8.x	10.x	12.x	2.7	3.6	3.7	3.8	1.11	1.13	2.1	2.2	3.1	8	11	2.5	2.7	4.2	7.3	
AWS		✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓	✗	✗	ECS
Azure	Linux	✗	✓[2]	✓[2,3]	✓[3]	✗	✓[2,3]	✓[2,3]	✓[3]	✗	✗	✗	✓[2]	✓[3]	✓[2,3]	✗	✗	✗	✗	✗	✓[#]
	Windows	✗	✓[2]	✓[2,3]	✓[3]	✗	✗	✗	✗	✗	✗	✗	✓[2]	✓[3]	✓[2,3]	✗	✗	✗	✗	✗	✗
Google		✓[†]	✓	✓[β]	✗	✗	✗	✓	✗	✓	✓[β]	✗	✗	✗	✗	✗	✗	✗	✗	✗	Cloud Run
IBM		✗	✓	✓	✗	✓	✓	✓	✗	✓	✗	✗	✓	✗	✓	✗	✓	✗	✓	✓	✓

Table 2: Memory/CPU configurations supported by Google Cloud Functions [17].

Memory	128 MB	256 MB	512 MB	1,024 MB	2,048 MB
CPU	200 MHz	400 MHz	800 MHz	1.4 GHz	2.4 GHz

2.4 IBM Cloud Functions

IBM Cloud Functions [25] is built on top of Apache OpenWhisk, an open source serverless cloud platform using Docker containers [3]. As such, in addition to Docker, it supports eight additional runtime systems [26]: Node.js, Python, Swift, PHP, Go, Java, Ruby and .NET Core. The service is restricted to 1,000 concurrently active executions (including those enqueued for execution) per namespace [28]. Memory allocation can be set from 128 MB to 2,048 MB in steps of 32 MB. IBM Cloud Functions is available in five out of the six IBM regions [27].[2] Our experiments revealed that some of the instances supporting the execution of the functions run on top of Intel Xeon E5-2683 v3 at 2.0 GHz.

3 RUNTIME SYSTEMS AND LANGUAGES

This section describes the runtime systems and programming languages supported by the serverless providers described in §2. For the sake of comparisons and fairness of benchmarking, we are interested in those supported by multiple cloud providers.

Table 1 highlights that Node.js is supported by all cloud providers. We explain this by the fact the peculiar features of the language and the event-driven nature of the runtime make it particularly fit for serverless computing, as well as by its popularity among developers and its productivity advantages. Python is similarly well supported, except by Azure on Windows. Microsoft's `.NET Core` lacks support from Google. In the remainder of this paper, we focus our comparison on the most supported runtime systems and languages, namely Node.js, Python, Go and .NET Core. We briefly introduce them next.

Node.js. [46] is a JavaScript runtime built on Chrome's V8 JavaScript engine [44]. The Node.js framework and the Node Package Manager (NPM) have greatly contributed to making JavaScript a popular language to implement all kinds of applications. Table 1 shows the versions of Node.js supported by each cloud provider. Because of its vast support from all providers, FaaSdom will deploy all Node.js applications using version 10.x.

Python is supported in multiple versions. At the time of this writing, all cloud providers support version 3.7, hence we rely on this version for our benchmarking results.

Go [15] is supported by the FaaS providers for two recent releases of the language. We use version 1.11 in our evaluation.

.NET Core is not uniformly supported by all the cloud providers. Hence, FaaSdom uses 2.1 on AWS and 2.2 on Azure and IBM. All `.NET Core` functions are implemented in the C# dialect of the `.NET` framework.

[2]One region (Tokyo) lacks support for Cloud Foundry [24] and as such cannot be used to deploy functions via the Command Line Interface (CLI). We therefore did not include it in our evaluation.

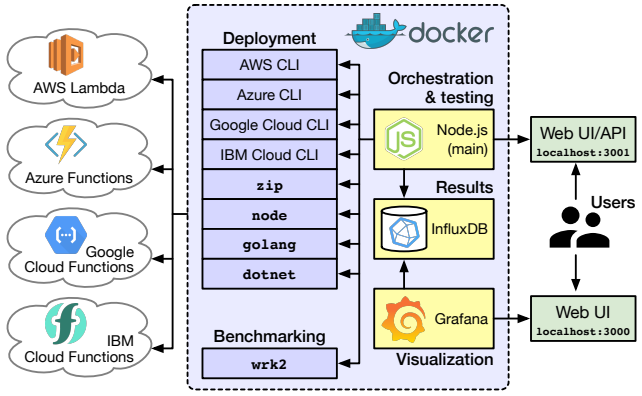


Figure 2: FaaSdom architecture.

4 ARCHITECTURE

This section describes the architecture of the FaaSdom prototype, depicted in Figure 2, as well as providing some additional implementation details (§4.5). The architecture includes the supported clouds and the corresponding serverless services (left), the involved Docker images (middle) and their interface with the system (right). We detail each component next.

Main application. The FaaSdom core component is implemented in JavaScript and leverages the Node.js framework. It manages all user input and executes the actions

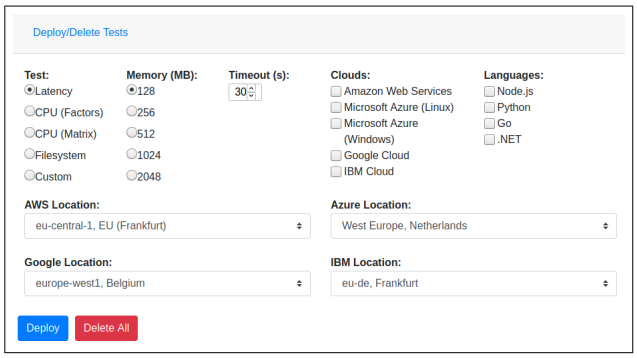


Figure 3: FaaSdom Web-based GUI.

or delegates them to other components. This application is packaged and executed as Docker containers. It manages the following main tasks: *(1)* deployment to the clouds; *(2)* execution of tests benchmarks; and *(3)* computing price estimations. Users access it through a Web-based GUI or via a REST API. Once started, the set of configured tests are deployed for execution (some examples are given in §5).

Time series DB. FaaSdom uses a time series database (Time Series Database (TSDB)) to store all the results from tests. These results are subsequently used by graphical interfaces and pricing calculation. Our prototype uses the InfluxDB [2] TSDB.

UI. The FaaSdom architecture provides an API to easily integrate visualization tools. Our prototype integrates with Grafana [1], an open source tool to display, plot and monitor data stored in a database. The results gathered by the tests and stored in InfluxDB are then displayed in Grafana.

CLIs. With the exception of AWS, all cloud providers offer a Docker image for their CLI. Resources can be deployed, deleted and managed completely by the CLI.

Runtime and languages (Node.js, Go, .NET). In addition to the source code of the function(s) to execute, a prebuilt and packaged zip file is commonly required to successfully complete the deployment. The FaaSdom architecture allows developers to ship runtime images, necessary for instance to install packages and build/run the corresponding code.

Workload Injectors. The FaaSdom architecture provides hook points for plugin workload injectors. The evaluation results shown in §6 uses `wrk2` [45], a stress tool for REST services, to issue requests toward services and gather throughput/latency results.

4.1 Deployment

Once the main application has started, the client can deploy the desired tests, for instance via the Web-based interface shown in Figure 3. Functions can be parametrized with memory allocated for each instance, timeouts, list of providers and runtime versions to use for the benchmarks, as well as geographical regions (as supported by the cloud provider) where to deploy the tests. Once deployed, the interface continuously reports on the progress and potential issues (*e.g.*, timeouts, access problems, *etc.*). The deployment process is parallelized, including the creation of the required cloud resources, the build and packaging of the functions, and their upload over the cloud. The deployment flow is slightly different for each provider, with some providers requiring significantly more operations than others. Figure 4 illustrates the various steps involved in the deployment for the providers supported by FaaSdom.

Upon cleanup, all deployed functions are permanently deleted, as well as all the configurations and resource groups created at deployment time.

4.2 Testing

When testing a serverless execution, FaaSdom will send every five seconds a request to the functions previously deployed, with the purpose to gather baseline results under low, constant load. Figure 5 illustrates a screenshot of Grafana displaying the results of a latency test in Node.js.

4.3 Benchmarking

For evaluating the performance of a deployed function, FaaSdom's benchmarking component injects specific workloads toward the function exposed by a given serverless provider. Our design is modular and currently supports `wrk2` [45], a constant throughput, correct latency HTTP benchmarking tool. Additional tools are easy to integrate by providing a Docker container and a simple REST interface to exchange parameters and results. The FaaSdom architecture allows

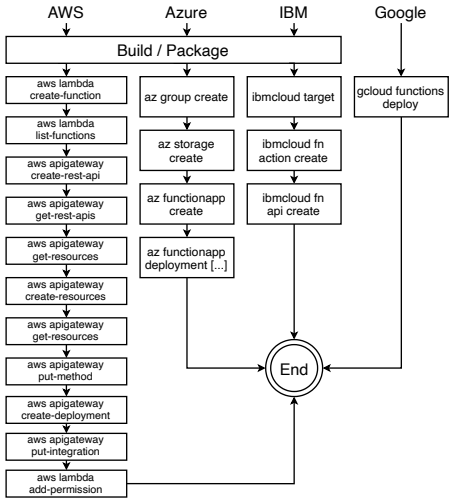
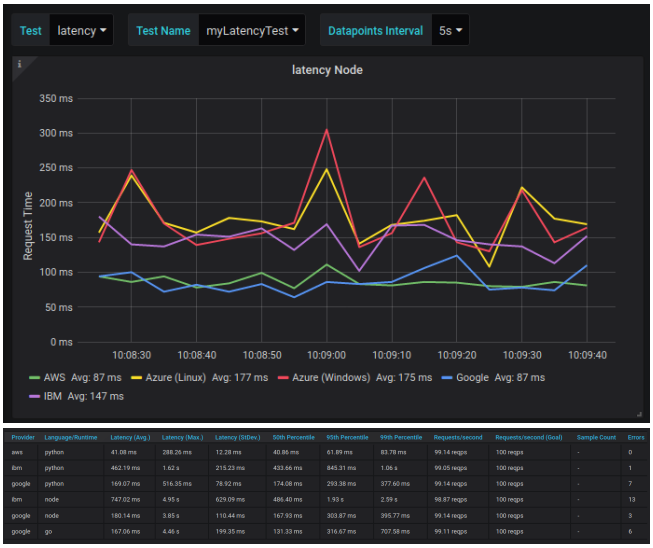


Figure 4: Deployment steps for the different cloud providers supported by FaaSdom.



Provider	Language/Runtime	Latency (Avg.)	Latency (Max.)	Latency (StDev.)	50th Percentile	95th Percentile	99th Percentile	Requests/second	Requests/second (Goal)	Sample Count	Errors
aws	python	41.08 ms	288.26 ms	12.28 ms	40.86 ms	61.89 ms	83.78 ms	99.14 reqps	100 reqps	-	0
ibm	python	462.19 ms	1.62 s	215.23 ms	433.66 ms	845.31 ms	1.06 s	99.05 reqps	100 reqps	-	1
google	python	169.07 ms	516.35 ms	78.92 ms	174.08 ms	293.38 ms	377.60 ms	99.14 reqps	100 reqps	-	7
ibm	node	747.02 ms	4.95 s	629.09 ms	486.40 ms	1.93 s	2.59 s	98.87 reqps	100 reqps	-	13
google	node	180.14 ms	3.85 s	110.44 ms	167.93 ms	303.87 ms	395.77 ms	99.14 reqps	100 reqps	-	3
google	go	167.06 ms	4.46 s	199.35 ms	131.33 ms	316.67 ms	707.58 ms	99.11 reqps	100 reqps	-	6

Figure 5: Visualization of performance results in Grafana.

users to directly visualize and analyze these results, using Grafana (Figure 5) or other tools that can fetch the result data from the TSDB.

4.4 Billing Costs Calculator

FaaSdom provides a pricing calculator component, which can be used to evaluate beforehand the cost of executing a certain workload on the supported cloud providers. Developers provide the planned workload (*e.g.*, number of function invocations, execution time per call, size of the returned data and allocated memory, *etc.*). The FaaSdom prototype produces an overview of the billing costs across the various serverless providers. In future work, we envision this module to be able to forecast the billing costs even for full-fledged serverless applications, by applying machine-learning techniques to the system traces produced from sample executions serving real-world workloads.

4.5 Container Implementation

The implementation of FaaSdom extensively relies on Docker containers. Specifically, the main container invokes other containers using a technique called *Docker-in-Docker*.[3] We do so by granting the main container access to `/var/run/docker.sock` and mounting it as a volume. During our evaluation, we did not observe any particular performance degradations using this approach.

The implementation of FaaSdom is freely available to the open-source community at https://github.com/faasdom/benchmark-suite-serverless-computing. Its core components consist of about 2,000 lines of Node.js code.

[3]https://www.docker.com/blog/docker-can-now-run-within-docker/

5 THE FAASDOM BENCHMARK SUITE

The FaaSdom benchmark suite consists of a collection of tests targeting different system aspects. Each test is implemented in several programming languages in order to support the variety of the available runtime systems. While there are some subtle differences across the code variants to adapt to the specificities of individual cloud providers, the code executed for each programming language is largely the same for all. All functions are implemented using an HTTP trigger, as supported by all the serverless providers. Currently, FaaSdom includes the followings benchmarks:

`faas-fact` and **`faas-matrix-mult`** are two CPU-bound benchmarks to respectively factorize an integer and multiply large integer matrices.

`faas-netlatency` is a network-bound test that immediately returns upon invocation, with a small JSON payload (HTTP body of 79 bytes and HTTP complete response including headers of ~500 bytes). This test is used to verify the roundtrip times for geographically distributed deployments

`faas-diskio` is an IO-bound benchmark to evaluate the performance of a disk. Perhaps surprisingly, each serverless cloud provides a temporary file-system that functions can use to read and write intermediate results. Other functions executed by the same instance share this file-system to access its content.

`faas-custom` is provided by FaaSdom to allows developers to test custom functions. The suite provides templates for the currently supported cloud providers and the supported implementation languages. In the long term, we expect the number of custom functions to greatly outnumber those provided by the suite itself.

6 EVALUATION

We present in this section the results of our evaluation using FaaSdom of the different FaaS providers supported by our current implementation. We carried out these tests over a period of three weeks (Jan. 13–Feb. 4, 2020). These experiments specifically aim at answering the following questions: *(1)* what is the most effective programming language to use for serverless applications; *(2)* what is the most convenient provider; and *(3)* which provider yields the most predictable results.

6.1 Call Latency

We begin by measuring the latency (round-trip) for all cloud providers and corresponding regions, using the `faas-netlatency` benchmark. Unless specified otherwise, we deploy Node.js using 128 MB. We note that AWS functions

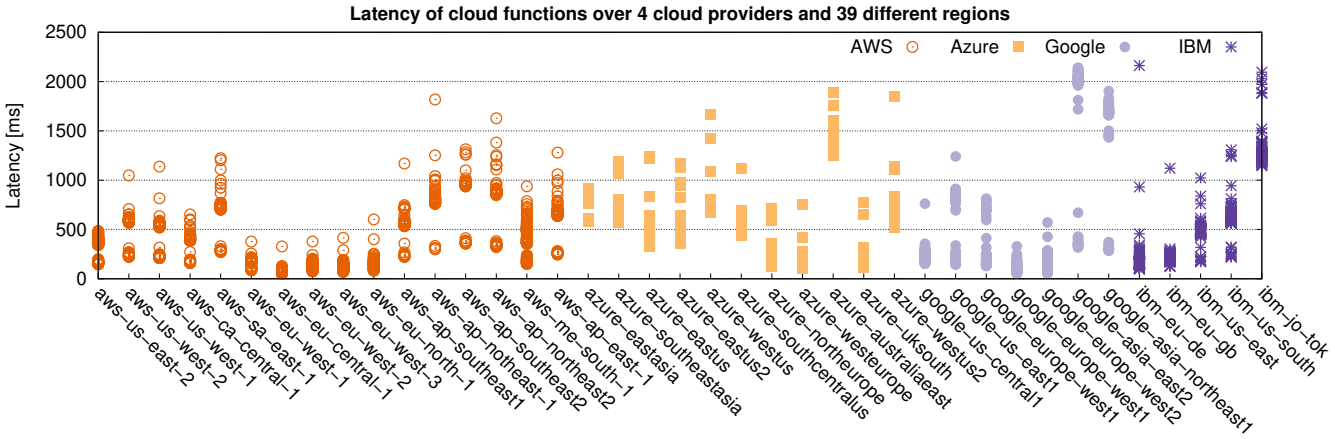


Figure 6: Latency test scatter plot. Since `azure-west-us` had issues with Node.js, we resort to `.NET` for that zone.

run under a custom kernel 4.14.138-99.102.amzn2.x86_64;[4] whole host configurations are not disclosed for Azure, Google or IBM deployments. A request is sent every 5 seconds to each cloud and region, up to 100 samples per function. Figure 6 reports our results across 39 different configurations. The average latency over all clouds and regions is 538 ms. We achieve the fasted result on AWS Lambda (`eu-central-1`) at 80 ms, and the slowest toward Google (`asia-east2`) at 1,770 ms. Overall, the best performing cloud provider is Amazon, with slightly lower latency for the same geographical locations of other providers.

6.2 Cold Start

A cold start happens when a function takes longer than usual to execute. Most of the time, this occurs when the invocation is scheduled right after the completion of its deployment, or when the function has not been used in a while, *e.g.*, after 10 minutes without invocations. Specifically, there are several steps to perform before a function can execute: allocate a server, set up a worker with the associated code, packages and extensions, load the function in memory, and finally run it. These steps contribute clearly to the cold start effect that we observe in our results. When the function is *warm*, it is ready in memory and can immediately serve requests. A function can also be de-allocated (*i.e.*, garbage-collected) when a new instance needs to be provisioned for scaling purposes. When this happens and an invocation is scheduled for the just garbage-collected function, the cold start effect is clearly measurable with degrading effects on the measured latency. Figure 7 shows the execution workflow for a cold start on Azure, as opposed to the warm case. Similar workflows (and effects on latency) exist on the other cloud providers.[5]

Our results are averaged over ten executions for each configuration runtime/cloud. Functions are executed using 512 MB of memory and without external packages, to reduce the baseline footprint and the corresponding loading time to the bare minimum. We executed these tests on the following regions: `eu-central-1` for AWS, `west-europe` for Azure, `europe-west1` for Google and `eu-de` for IBM. Note however that the chosen region has no impact on the cold start latency. We compute the cold start latency as the total request time minus the normal average latency.

Figure 8 summarizes our results using a box-and-whiskers plot. It stands out that AWS is overall the fastest, with an average cold start latency of only 335 ms for Node.js, Python and Go. Results with .NET are worse, with latencies up to 1,739 ms, likely due to the nature of the runtime and the

[4]https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html

[5]Refer for instance to https://cloud.google.com/functions/docs/bestpractices/tips for information and best practices from Google.

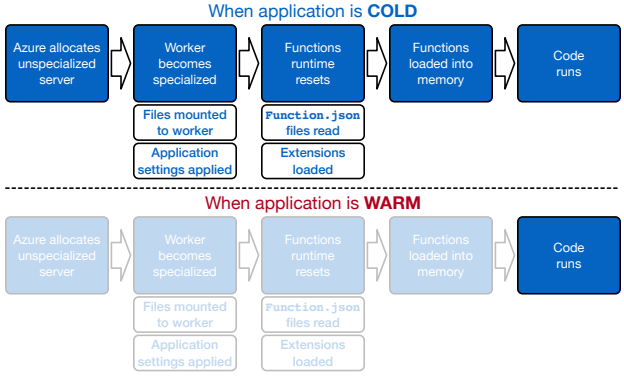


Figure 7: Workflow for cold and warm start on Azure [47].

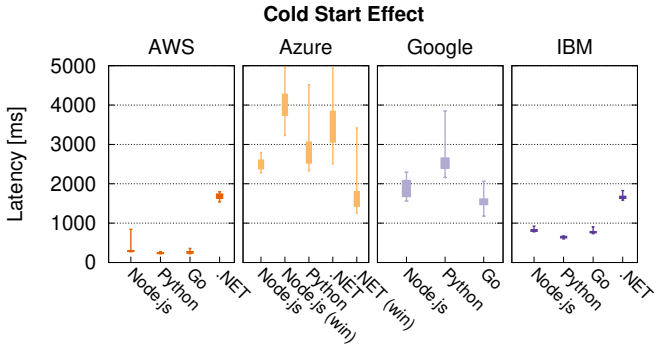


Figure 8: Cold start latency.

compilation of C#. On Azure, cold start latency is strictly more than 2 seconds and up to 5 seconds, with the exception of the combination .NET on Windows, which averages at 1,917 ms. IBM compares similarly to AWS. Finally, Google Cloud consistently yields higher cold start latencies, in the 2-3 seconds range. IBM exhibits a performs similarly to AWS across the spectrum of supported languages, although the cold start latency is around 600 ms higher for Node.js, Python and Go but similar for .NET.

On AWS and IBM, it took usually around 10 minutes of no activity for the computing instance to be recycled by the provider and re-inserted into the pool of available ones, and up to 20 minutes on Azure. On Google, this time varied from 10 minutes up to 10 hours.

6.3 CPU-bound Benchmarks

This benchmark evaluates how CPU-bound workloads behave across the different cloud providers. A function (`faas-fact`) is invoked every five seconds, using as parameter a large integer triggering a sufficiently challenging computation. We collect 100 results for every configuration (runtime, memory, cloud provider). We report these results for all cloud configurations and languages: Python (Figure 9), Node.js (Figure 10), Go (Figure 11) and .NET (Figure 12).

We note a few interesting facts. On AWS Lambda, the standard deviation is low, with a minimum value of 50 ms using .NET and 2,048 MB of memory, with very consistent execution times across all the different configurations. As expected, every doubling of the allocated memory results in halving the execution time, *i.e.*, performance scales linearly with allocated memory [5].

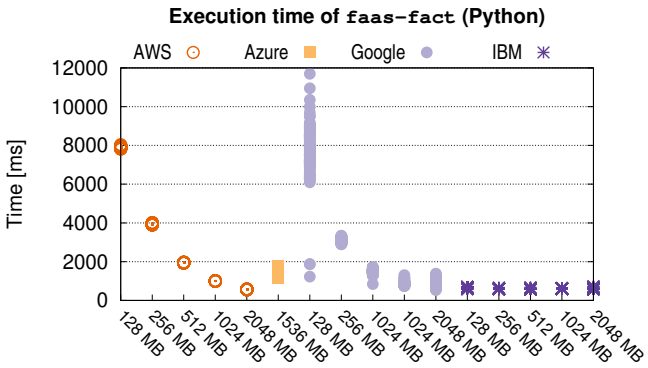


Figure 9: Execution time of `faas-fact` in Python.

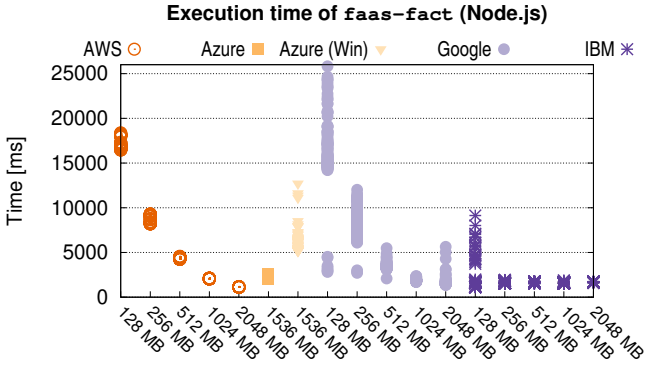


Figure 10: Execution time of `faas-fact` in Node.js.

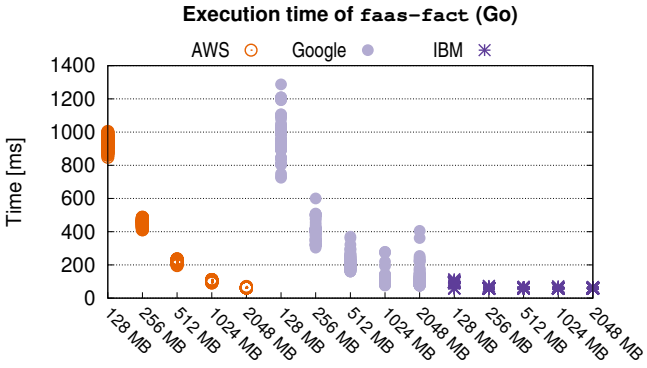


Figure 11: Execution time of `faas-fact` in Go.

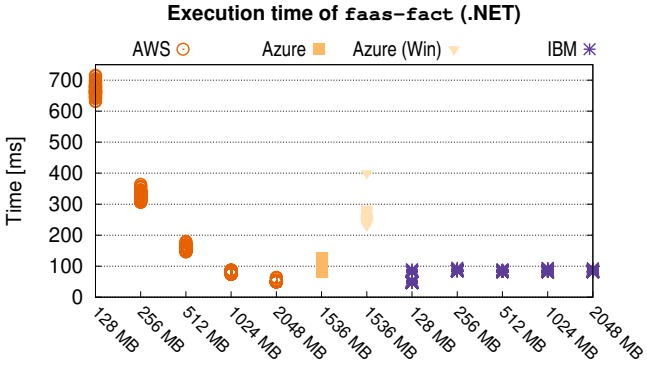


Figure 12: Execution time of the `faas-fact` in .NET.

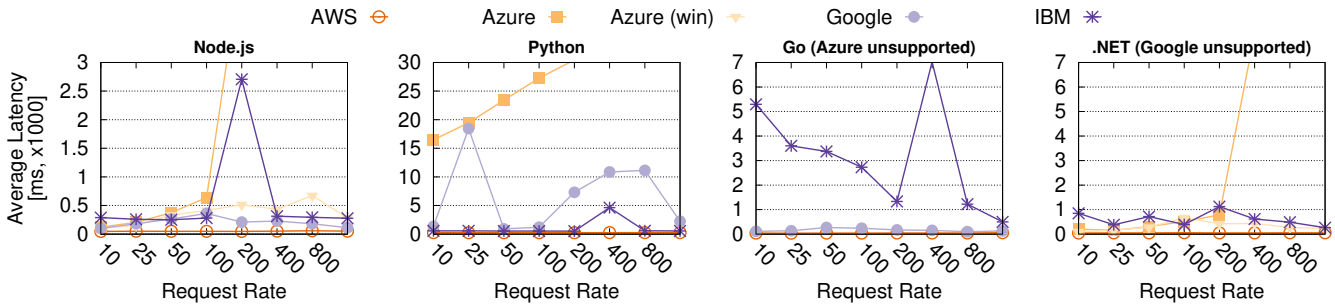


Figure 13: Throughput/latency benchmark using `wrk2` [45] and FaaSdom's `faas-matrix-mult` stress function.

Azure's memory configuration of 1,536 MB suggests its performance to be in the 1,024-2,048 MB range. Surprisingly, it is slower than 1,024 MB instances of the other clouds. Google Cloud Platform performs similarly to AWS, although more scattered for 128 MB of memory. Google achieves better results than AWS for several memory configurations (128, 256, 512 and 1,024 MB). Note however that the performance scaling behaves differently on the Google platform [17]. The results we gathered on the IBM platform suggests that memory allocation does not correlate with CPU allocation in any way, with five different configurations performing very closely. This is remarkable since the pricing model of IBM only accounts for the GB-seconds used.

6.4 Throughput/Latency

To understand the saturation point of the deployed services, we rely on `wrk2` [45], a constant throughput/exact latency HTTP-based benchmarking tool. We configure this benchmark to issue function call invocations at increasingly high rates, from 10 up to 1,000 requests per second. For each of the configurations, `wrk2` reports the average latency to handle the requests (*i.e.*, the functions). Between each configuration, the benchmark waits for sufficient time to process any request still possibly enqueued. Figure 13 shows our results for the `faas-matrix-mult` function and the 4 different programming languages under test.

We observe that AWS achieves stable response latencies for all the tested workloads. Azure, on the other hand, shows more surprising results. Whereas it can tolerate high loads on Windows OS, it performs very inconsistently on Linux. The response latency grows almost linearly with the total requests per second injected and quickly saturates. That is a strong indication that none (or too few) new instances were allocated to handle the load. We investigated this behaviour further through the Azure portal Live Metrics Stream (not shown). When reaching 1,000 requests per second, only a total of 12 instances were deployed to serve .NET functions. Additionally, only 500 requests per second were actually served, leaving a large percentage of requests in the waiting queue.

Table 3: Latency: goal/target, results in the long-tail (below 90% of the target requests/sec).

Cloud	Runtime	Req/s (goal)	Req/s (actual)	Req/s (%)
Azure	Node.js	200	179	89.50%
Azure	Node.js	400	227	56.75%
Azure	Node.js	800	278	34.75%
Azure	Node.js	1000	327	32.70%
Azure	Python	10	6	60.00%
Azure	Python	25	12	48.00%
Azure	Python	50	18	36.00%
Azure	Python	100	25	25.00%
Azure	Python	200	30	15.00%
Azure	Python	400	15	3.75%
Azure	Python	800	14	1.75%
Azure	Python	1000	15	1.50%
Azure	.NET	400	324	81.00%
Azure	.NET	800	407	50.88%
Azure	.NET	1000	512	51.20%
Google	Python	25	12	48.00%
Google	Python	200	168	84.00%
Google	Python	400	297	74.25%
Google	Python	800	575	71.88%
IBM	Go	25	21	84.00%

For Node.js and .NET, response latencies spike up to 24 s and 18 s, respectively. Using Python, Azure managed to handle up to 200 requests per second, beyond which none of the requests could be served correctly. Google can manage the load generally well, although latency increases slightly for Node.js and Go. However, Python functions deployed Google suffer from poor performance, in particular at higher request rates. Results suggest poor capacity in adapting quickly to flash-crowd requests, with the average request duration growing from 1,356 to 18,448 ms when increasing the rate from 10 to 25 requests per second. We investigate further (Figure 14) by analyzing the scaling behaviour for instances running Google Functions. The plot shows the number of instances allocated during the load test on Google, suggesting indeed that the auto-scalability features provided by the infrastructure might lead to poor results as requests start saturating the deployed instances.

On IBM Cloud Functions, Node.js performs particularly poorly when compared to the other providers, while Python

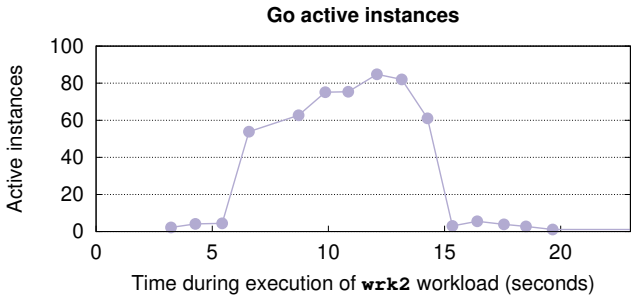


Figure 14: Google Go: number of active compute instances during the wrk2 load test.

Table 4: Pricing calculation: cost of the CPU-bound benchmarks computed by FaaSdom according to the pricing models of each cloud.

Memory	Exec. time	Invocation	GB/sec	GHz/sec	BW	Total
AWS						
128 MB	8000 ms	2.00$	166.67$	—	3.43$	172.10$
256 MB	4000 ms	2.00$	166.67$	—	3.43$	172.10$
512 MB	2000 ms	2.00$	166.67$	—	3.43$	172.10$
1 GB	1000 ms	2.00$	166.67$	—	3.43$	172.10$
2 GB	600 ms	2.00$	200.00$	—	3.43$	205.43$
Azure						
128 MB	1267 ms	2.00$	25.34$	—	3.32$	30.66$
Google						
128 MB	7700 ms	4.00$	24.06$	154.00$	4.58$	186.64$
256 MB	3200 ms	4.00$	20.00$	128.00$	4.58$	156.58$
512 MB	1600 ms	4.00$	20.00$	128.00$	4.58$	156.58$
1 GB	900 ms	4.00$	22.50$	126.00$	4.58$	157.08$
2 GB	800 ms	4.00$	40.00$	192.00$	4.58$	240.58$
IBM						
128 MB	700 ms	—	14.88$	—	—	14.88$
256 MB	600 ms	—	25.50$	—	—	25.50$
512 MB	600 ms	—	51.00$	—	—	51.00$
1 GB	600 ms	—	102.00$	—	—	102.00$
2 GB	700 ms	—	238.00$	—	—	238.00$

functions perform on par with AWS. Go functions achieve far worse than for other providers, contrasting with the cold start results that suggested similar behaviour. Further investigation is needed as part of future work to understand the reasons of this behaviour. Table 3 reports for which cases the target throughput set by `wrk2` could not be reached within a 10% margin, *i.e.*, cases achieving at most 90% of the target throughput (right-most column).

6.5 Pricing

We conclude our quantitative evaluation by showcasing one possible usage of FaaSdom's billing calculator (§4.4). To that end, we use it to evaluate the cost of the CPU-bound benchmarks if executed across all the serverless providers.

We assume a Python runtime and 10 M function calls/-months. The allowed memory is set to a 100 MB function and a return payload of 4 KB per call. We note that Azure supposedly charges the GB-seconds that the function actually used, rounded up to the next 128 MB step [37]. It seems therefore that the pricing model of Azure is similar to the one of IBM, delivering the same performance independent of the memory size.

Table 4 shows the costs computed by FaaSdom. We observe that AWS is particularly accurate: for every doubling of memory, execution time halves, keeping the costs constant. Despite not being the cheapest option, it remains the most predictable one. Azure is the cheapest cloud provider according to our computations. Since this example function application only uses 100 MB, Azure also only charges for 128 MB: memory is dynamically allocated and only billed according to the next 128 MB step. The costs of deploying Google Functions follows closely those of AWS, especially for mid-range configurations. For the least- and most-expensive configurations, AWS reveals to not be the best trade-off in terms of cost/performance. Finally, IBM's billing method offers predictable costs, being roughly linearly dependant of the chosen memory configurations.

6.6 Usability Considerations

We finally report on some usability considerations regarding the cloud providers and their serverless offerings.

AWS Lambda. Overall, this reveals to be the best-performing cloud provider. According to our continuous monitoring over several months, it provides consistent performance and reliability. Management via the CLI or Web-based portal is straightforward and efficient, even though setting up HTTP triggers remain convoluted.

Drawbacks: (i) lack of official Docker image for the AWS CLI; *(ii)* deleting a function requires usage of `aws lambda list-functions`, which however only works on a per-region basis; *(iii)* security counter-measures imposed by the AWS (*e.g.*, changing public APIs more frequently than 30 s) can reduce the productivity of the developers as well as the ability of quickly prototyping and testing functions.

Azure Functions. Performance and usability are worse than AWS. The cold start latency of 2 to 4 s makes it unfit for short-lived sessions. The Web-based portal provides useful insights on the real-time performance, as well as system insights from telemetry-based monitoring on the supporting computing instances.

Drawbacks: (i) there is a discrepancy between the actions available via the official CLI and the Web-based portal; *(ii)* additional tools are required also for local debugging (*e.g.*, the Azure Functions core tools,[6]), which seems counter-intuitive, and they are not shipped with any official Docker image;

[6] https://github.com/Azure/azure-functions-core-tools

(iii) the overall architecture, as well as the different generations of APIs, contribute in making the learning curve for Azure rather steep.

Google Cloud Functions. This offering leverages a simple and clearly structured Web portal, as well as a consistent CLI tool. Support for single-command operations is particularly helpful and practical. While cold start latency is higher than AWS or IBM, it remains within usable limits. Regarding billing costs, Google is the least convenient operator.

Drawbacks: (i) the monitoring measurements available to the developers lag several minutes behind the real execution, preventing prompt interventions. *(ii)* it supports a limited number of runtime systems, at the risk of being a less attractive deployment choice; *(iii)* deployers are left with very few configuration options, preventing (by design) fine-tuning operations by the clients.

IBM Cloud Functions. The current public release of IBM lacks straightforward and well-structured documentation. The cold-start performance suggests this provider to be a good candidate for quick tests, where an application can start invoking functions with short waiting times.

Drawbacks: (i) the official IBM CLI can only be used if there is a corresponding cloud foundry support for the intended region (currently the case for all public regions except for Tokyo); *(ii)* the support for the Go runtime is rather poor, resulting in the worst performance for this configuration; *(iii)* some (important) features are left undocumented, such as the limit of 3,000 request per minute authorized for normal accounts.

7 LESSONS LEARNED

This study has shown that serverless computing in the form of Function as a Service (FaaS) can deliver good performance at a reasonable price. It is fairly easy to set up, deploy and execute code in the cloud for the end user, as compared to VMs or even physical servers. With serverless computing, companies can focus on building their business logic instead of maintaining operating systems and servers on premises, which requires expertise and is not always affordable for small companies. One of the most important features of FaaS is is auto-scaling capability, which take away from the deployer the burden of dimensioning the system and allocating resources.

Through the implementation and evaluation of FaaSdom, we notably encountered the following limitations of the FaaS paradigm.

First, one is heavily limited by the offering of specific cloud providers, including available runtime systems and programming languages, supported triggers, third party services integration, quotas or maximum memory. Users cannot easily overcome such limitations as the full application stack is managed by the provider.

Second, providers can force an application to migrate from a specific runtime version to an arbitrarily different one (*e.g.*, when upgrading their system from Node.js v6 to v8). Despite backward compatibility, functions might break, forcing developers to keep up with the pace imposed by the cloud provider.

Third, developers face the risk of vendor lock-in by depending on proprietary features. This might become a serious issue when considering application upgrades and potential provider migrations.

Finally, the lack of deployment standards makes the FaaS paradigm still a largely experimental playground. Providers offer custom APIs (*e.g.*, request/response objects), which inevitably vary across cloud offerings. A common framework supported by a standard would certainly improve adoption and benefit the whole ecosystem.

8 CONCLUSION AND FUTURE WORK

Motivated by the lack of common FaaS framework or standard platform to evaluate serverless platforms, this paper presented FaaSdom, a user-space application that facilitates performance testing across a variety of serverless platform. FaaSdom is an open-source benchmark suite that is easy to deploy and provides meaningful insights across a variety of metrics and serverless providers.

We envision to extend FaaSdom along the following directions. First, we will expand the set of supported languages and runtime systems, also by integrating contributions from the open-source community. Second, we plan to maintain a continuous, online deployment of the FaaSdom, publicly accessible, in order to build a larger dataset of historical measurements that will be released to the research community. We finally intend to include native support for Docker images to ship functions, in order to facilitate integration with services such as Google Cloud Run [22].

REFERENCES

[1] 2020. Grafana - he open observability platform. https://grafana.com/ Last accessed: 26.03.2020.

[2] 2020. InfluxDB - the open source time series database. https://www.influxdata.com/ Last accessed: 26.03.2020.

[3] Apache. [n.d.]. Apache OpenWhisk. https://openwhisk.apache.org/ Last accessed: 05.11.2019.

[4] AWS, Amazon Web Services. [n.d.]. AWS Lambda. https://aws.amazon.com/lambda/ Last accessed: 22.10.2019.

[5] AWS, Amazon Web Services. [n.d.]. AWS Lambda Function Configuration. https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html Last accessed: 22.10.2019.

[6] AWS, Amazon Web Services. [n.d.]. AWS Lambda Function Scaling. https://docs.aws.amazon.com/lambda/latest/dg/scaling.html Last accessed: 22.10.2019.

[7] AWS, Amazon Web Services. [n.d.]. AWS Lambda Releases. https://docs.aws.amazon.com/lambda/latest/dg/lambda-releases.html Last accessed: 22.10.2019.

[8] AWS, Amazon Web Services. [n.d.]. AWS Lambda Runtimes. https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html Last accessed: 22.10.2019.

[9] AWS, Amazon Web Services. [n.d.]. AWS Region Table. https://aws.amazon.com/about-aws/global-infrastructure/regional-product-services/ Last accessed: 10.12.2019.

[10] AWS, Amazon Web Services. 2019. Serverless. https://aws.amazon.com/serverless/ Last accessed: 18.12.2019.

[11] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*, Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya (Eds.). Springer Singapore, Singapore, 1–20. https://doi.org/10.1007/978-981-10-5026-8_1

[12] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The Rise of Serverless Computing. *Commun. ACM* 62, 12 (Nov. 2019), 44–54. https://doi.org/10.1145/3368454

[13] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, and Maciej Malawski. 2018. Performance evaluation of heterogeneous cloud functions. *Concurrency and Computation: Practice and Experience* 30, 23 (2018), e4792. https://doi.org/10.1002/cpe.4792 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4792 e4792 cpe.4792.

[14] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[15] Go. 2019. The Go Project. https://golang.org/project/ Last accessed: 10.12.2019.

[16] Google. [n.d.]. Cloud Functions. https://cloud.google.com/functions Last accessed: 29.10.2019.

[17] Google. [n.d.]. Cloud Functions - Pricing. https://cloud.google.com/functions/pricing#compute_time Last accessed: 30.10.2019.

[18] Google. [n.d.]. Cloud Functions - Quotas. https://cloud.google.com/functions/quotas Last accessed: 30.10.2019.

[19] Google. [n.d.]. Cloud Functions - Release Notes. https://cloud.google.com/functions/docs/release-notes Last accessed: 30.10.2019.

[20] Google. [n.d.]. Cloud Functions - Writing Cloud Functions. https://cloud.google.com/functions/docs/writing Last accessed: 30.10.2019.

[21] Google. [n.d.]. Cloud Functions Locations. https://cloud.google.com/functions/docs/locations Last accessed: 10.12.2019.

[22] Google. [n.d.]. Cloud Run (fully managed) release notes. https://cloud.google.com/run/docs/release-notes Last accessed: 07.02.2020.

[23] Google. 2019. Serverless computing. https://cloud.google.com/serverless/ Last accessed: 18.12.2019.

[24] IBM. [n.d.]. Cloud Foundry. https://www.ibm.com/cloud/cloud-foundry Last accessed: 07.02.2020.

[25] IBM. [n.d.]. IBM Cloud Functions. https://cloud.ibm.com/functions Last accessed: 30.10.2019.

[26] IBM. 04.09.2019. IBM Cloud Functions - Runtimes. https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-runtimes Last accessed: 05.11.2019.

[27] IBM. 12.07.2019. IBM Cloud Functions - Regions. https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-cloudfunctions_regions Last accessed: 10.12.2019.

[28] IBM. 18.09.2019. IBM Cloud Functions - System details and limits. https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-limits Last accessed: 05.11.2019.

[29] IBM. 2019. Serverless computing. https://www.ibm.com/cloud/learn/serverless Last accessed: 18.12.2019.

[30] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. 2002. Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites. In *Proceedings of the 11th international conference on World Wide Web*. 293–304.

[31] Yochay Kiriaty. [n.d.]. Announcing general availability of Azure Functions. https://azure.microsoft.com/en-us/blog/announcing-general-availability-of-azure-functions/ Last accessed: 29.10.2019.

[32] Aleksandr Kuntsevich, Pezhman Nasirifard, and Hans-Arno Jacobsen. 2018. A Distributed Analysis and Benchmarking Framework for Apache OpenWhisk Serverless Platform. In *Proceedings of the 19th International Middleware Conference (Posters) (Middleware '18)*. ACM, New York, NY, USA, 3–4. https://doi.org/10.1145/3284014.3284016

[33] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. 2018. Evaluation of Production Serverless Computing Environments. https://doi.org/10.13140/RG.2.2.28642.84165

[34] Maciej Malawski, Kamil Figiela, Adam Gajek, and Adam Zima. 2018. Benchmarking Heterogeneous Cloud Functions. In *Euro-Par 2017: Parallel Processing Workshops*, Dora B. Heras, Luc Bougé, Gabriele Mencagli, Emmanuel Jeannot, Rizos Sakellariou, Rosa M. Badia, Jorge G. Barbosa, Laura Ricci, Stephen L. Scott, Stefan Lankes, and Josef Weidendorfer (Eds.). Springer International Publishing, Cham, 415–426.

[35] Microsoft. [n.d.]. Azure Database for PostgreSQL. https://azure.microsoft.com/services/postgresql/ Last accessed: 11.03.2020.

[36] Microsoft. [n.d.]. Azure Functions. https://azure.microsoft.com/en-us/services/functions/ Last accessed: 07.02.2020.

[37] Microsoft. [n.d.]. Azure Functions pricing. https://azure.microsoft.com/en-us/pricing/details/functions/ Last accessed: 22.01.2020.

[38] Microsoft. [n.d.]. Azure Functions runtime versions overview. https://docs.microsoft.com/en-us/azure/azure-functions/functions-versions Last accessed: 15.01.2020.

[39] Microsoft. [n.d.]. Azure Functions scale and hosting. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale Last accessed: 29.10.2019.

[40] Microsoft. [n.d.]. General purpose virtual machine sizes - Av2-series. https://docs.microsoft.com/en-us/azure/virtual-machines/

windows/sizes-general#av2-series Last accessed: 29.10.2019.

[41] Microsoft. [n.d.]. Products available by region. https://azure.microsoft.com/en-us/global-infrastructure/services/?products=functions&regions=all Last accessed: 10.12.2019.

[42] Microsoft. [n.d.]. Supported languages in Azure Functions. https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages Last accessed: 29.10.2019.

[43] Microsoft. 2019. Serverless computing. https://azure.microsoft.com/en-us/overview/serverless-computing/ Last accessed: 18.12.2019.

[44] Node.js Foundation. [n.d.]. Node.js. https://nodejs.org/ Last accessed: 07.11.2019.

[45] Gil Tene. 2019. wrk2 - a HTTP benchmarking tool based mostly on wrk. https://github.com/giltene/wrk2 Last accessed: 04.02.2020.

[46] Stefan Tilkov and Steve Vinoski. 2010. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing* 14, 6 (2010), 80–83.

[47] Colby Tresness. 07.02.2018. Understanding serverless cold start. https://azure.microsoft.com/en-us/blog/understanding-serverless-cold-start/ Last accessed: 22.01.2020.

[48] Erwin van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. 2017. The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures. In *Proceedings of the 2Nd International Workshop on Serverless Computing (WoSC '17)*. ACM, New York, NY, USA, 1–4. https://doi.org/10.1145/3154847.3154848

[49] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang