Evaluating the Scalability and Elasticity of Function as a Service Platform

Kim Long Ngo
kimlngo@yorku.ca
York University
Canada

Joydeep Mukherjee
jmukherj@calpoly.edu
California Polytechnic State University
United States

Zhen Ming Jiang
zmjiang@cse.yorku.ca
York University
Canada

Marin Litoiu
mlitoiu@yorku.ca
York University
Canada

Abstract

Function as a Service (FaaS) is a new software technology with promising features such as automated resource management and auto-scaling. Since these operational aspects are transparent, software engineers may not fully understand the scaling characteristics as well as limitations of this technology and this lack of information can lead to undesired performance results. To address these concerns, we perform a study to characterize FaaS' scalability with intensive workloads on three popular FaaS cloud platforms, namely Amazon AWS Lambda, IBM and Azure Cloud Function. We also study a workload smoother design pattern to examine if it enhances FaaS overall performance. The results show that different FaaS platforms adopt distinct scaling strategies and by applying a workload smoother, software engineers can achieve 99 - 100% success rates compared to 60 - 80% when FaaS' system is saturated.

CCS Concepts

• **Software and its engineering** → **Software infrastructure**.

Keywords

serverless, function as a service, software performance, scalability, elasticity, workload smoother.

ACM Reference Format:
Kim Long Ngo, Joydeep Mukherjee, Zhen Ming Jiang, and Marin Litoiu. 2022. Evaluating the Scalability and Elasticity of Function as a Service Platform. In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22), April 9–13, 2022, Bejing, China.* ACM, Beijing, China, 8 pages. https://doi.org/10.1145/3489525.3511682

1 Introduction

Software architecture and technology have evolved rapidly in the last decade, with software deployment transitioning from monolithic architecture operating on Virtual Machine (VM) to Microservice architecture running on light-weight containers such as Docker. To relieve the software engineers from operational tasks such as resource management and capacity planning, a new technology called Function as a Service (FaaS) was popularized since 2014. FaaS is an event-driven cloud platform that enables software engineers to focus on business logic and leave the infrastructure management to cloud providers [2]. FaaS implementation are snippets of source code developed in supported programming languages and executed inside a container on a cloud platform, they are usually referred to as cloud functions. When a cloud function is triggered, the cloud provider will automatically create a new container and load the source code for execution into this container [4].

One of the promising FaaS features is resource auto-scaling, which is the automatic provisioning and de-provisioning of computing resources in response to workload changes [3]. This feature, on one hand, simplifies the software engineering operational aspect, but on the other hand, poses a challenge to performance planning. Software engineers may need to know in-depth about how each cloud provider scales their cloud functions, how reactive the scaling is or what are the limitations when scaling up or down. Hence, a load and performance benchmark is essential to justify this technology's auto-scaling feature.

Most previous studies on FaaS focused on evaluating the performance by running heavy CPU, Memory, I/O and Network benchmarks on different FaaS platforms. There were only a few research studies that focused on the auto-scaling aspect of FaaS [11, 14]. Those studies only used a small number of concurrent clients and low intensity workloads.

In this paper, we describe an empirical study of FaaS' auto-scaling and report the results of our systematic load and performance experiments. The results show auto-scaling strategy is implemented differently by different cloud providers. Then, we examine the use case when FaaS has spawned all the allowable cloud function instances to serve the traffic (i.e., the upper concurrency capacity limit has been reached). Next, we evaluate the effects of introducing a workload smoother which is a software component located in-front of a target system to smooth the workload and prevent overloading. Our studies were conducted on three popular cloud providers, namely Amazon AWS Lambda, IBM and Azure Cloud Function. The contributions of this paper are:

- We simulate intensive workload scenarios, evaluate and report FaaS' auto-scaling characteristics on different cloud platforms.
- We propose and show the advantages of a workload smoother which implements the *Queue-Based Load Leveling* microservice design pattern [24]. Our prototype implementation shows that it can achieve a 99 - 100% success rate compared to 60 - 80% when FaaS' system is saturated.

The paper is organized as follows. Section 2 introduces the background and related work for FaaS' scalability. Section 3 presents the research questions and methodology used in our study. Section 4 describes our experimental setup. Section 5 presents the evaluation results. Section 6 outlines potential threats which might impact

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE'22, April 9–13, 2022, Beijing, China
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9143-6/22/04…$15.00
https://doi.org/10.1145/3489525.3511682

our results. Finally, section 7 discusses and concludes our research work.

2 Background and Related Works

In this section, we first provide an overview of the FaaS in Section 2.1. Then we discuss the related work in Section 2.2.

2.1 Background

2.1.1 FaaS Overview: Since its introduction in 2014, FaaS has gained the attention of both academic researchers and industry practitioners for its auto-scaling feature which is automatically adding more container instances to elevate processing capacity. While providing auto-scaling, all FaaS cloud platforms impose an upper concurrency limit, for example, by default AWS Lambda restricts the maximum concurrency at 1,000 concurrent clients. Beyond this limit, requests will be throttled with "Too Many Request" error (HTTP-429). Software engineers can reduce AWS Lambda concurrency level by using *"reserved concurrency"* option. Similarly, IBM cloud function also caps the maximum concurrent executions to 1,000 but does not provide reducing option. Under basic Consumption Plan, Microsoft Azure allows a cloud function to scale up to 200 instances and claims that each cloud function instance can serve multiple concurrent requests [15].

2.1.2 Scalability and Elasticity: Herbst et al. [8] defined scalability as the ability of the system to sustain the increasing workload by using of additional resources. Hence, scalability of a platform can be measured as the number of additional resources it can provide to sustain a workload burst.

Elasticity, on the other hand, is described as the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an automatic manner [8]. Elasticity is mainly related to "the change in resources and the reaction time is of importance" [12]. Consequently, elasticity of a system can be evaluated as the speed of scaling up a system from under-optimal to optimal states.

2.1.3 Cloud Design Patterns Overview: Cloud design patterns are documented solutions pertaining to specific software engineering problems. Each cloud design pattern includes the discussion about the context, problem, solution, issues and considerations. Software engineers and architects apply cloud design patterns to ensure that the software system is reliable, scalable and secured. Taibi et al. [24] reviewed 32 patterns that can be used for FaaS, some of these patterns, such as queue-based load leveling, have been used with microservices [17]. We implemented the queue-based load leveling pattern for FaaS to determine if this pattern can improve the performance and by how much.

2.2 Related Works

When benchmarking FaaS, most previous research focused on CPU, Memory, I/O, Network and only a few studies concentrated on the scalability. Kuhlenkamp et al. [11] presented an elasticity benchmark for AWS Lambda, IBM, Azure and Google Cloud. Their study used Node.js and a variety of evaluation metrics such as reliability, request-response latency, throughput, execution cost. The benchmark was designed to send slow ramping requests (60 seconds). Martins et al.[14] defined a suite of seven tests to benchmark the raw performance of AWS Lambda, Azure, Google and IBM cloud functions using latency and throughput measurement. The results showed that AWS Lambda, Azure and Google cloud functions exhibited almost linear throughput increase when the number of concurrent requests accelerated. Our study is different from above research in the following ways: we evaluate FaaS' scaling characteristics using a high intensity workload [23]; we use Java, which is a very popular programming language [20]; we investigate the response to burst workloads, where requests ramp up in short periods; we also followed well-established methodologies conducted by Cooper et. al. [6] to examine the software system at saturation point.

3 Research Questions and Methodology

In this section, we describe our Research Questions (RQ) and corresponding methodologies.

3.1 Research Questions

We formulate two following RQs:

- **RQ-1 (Scalability)**: What are the FaaS' scalability and elasticity characteristics under heavy load test?
  In **RQ-1**, we focus on FaaS' scalability and elasticity with different workload intensities. Based on the scalability and elasticity definition presented in Section 2.1.2, we chose to test with multi-concurrent clients.
- **RQ-2 (FaaS Under Saturation and Performance Improvement Patterns)**: What is FaaS' performance under saturation? Does workload smoother pattern help to improve the performance?
  In **RQ-2**, we inspect FaaS' performance at saturation level (i.e., when the throughput of a system stops increasing [6]) and apply the workload smoother design pattern to verify the performance improvement.

3.2 Methodology

3.2.1 RQ-1 Methodology: We used JMeter client [5] to send multi-concurrent requests to testing cloud functions. We note that previous studies experimented with concurrency level at 1, 5, 10, 20, and 30 concurrent requests [14, 19]. To simulate intensive workload, we decided to test the platforms with 100 concurrent clients.

In addition, the workload intensity was also controlled by the following JMeter parameters:

- **Ramp Up Time** refers to how long JMeter takes to create 100 testing threads. We noted that this parameter was also used by Somu et al.[21]. We used four discrete intervals which are one (1s), three (3s), six (6s) and ten seconds (10s).
- **Total Number of Requests** refers to the size of the examining workload. We designed three workload sizes which were 1000, 3000 and 7000 requests representing low, medium and high levels.

We evaluate FaaS' scalability and elasticity based on the following four metrics which were inferred from the definitions discussed in Section 2.1.2:

- *The number of function instances spawned*: In this metric, we counted the number of cloud function instances provisioned based on a unique cloud instance identifier (discuss in Section 3.2.3). This metric helps to discover the scaling patterns adopted by each cloud platform.
- *The ramp duration to expected number of instances*: In this metric, we measured the time it took cloud platforms to scale

up to required number of instances. The ramp duration was calculated as the time difference between the first response and the response which was first served by the lastly added instance. For example, if a test requires cloud platform to provision 100 instances, the ramp duration would be measured as the time difference between the first response's timestamp and the response's timestamp which was first served by the 100th instance. This metric indicates how quickly a cloud platform can add more instances. The shorter it takes, the better the elasticity is [8].

- *The system's throughput*: System throughput is a commonly used performance metric. It is measured as the number of requests processed in a unit of time. We calculated the throughput as follow:

$$Throughput = \frac{Total\ Number\ Of\ Request}{Execution\ Time\ In\ Second} * 60 \quad (1)$$

Throughput shows how many requests have been served in a unit of time (i.e., minute). The higher the throughput, the better the system performance is.

- *The median response time*: We measured the median response time which is also a commonly used performance metric. This response time shows the duration a system needs to produce the result. Compare to average response time, median response time is known to be less sensitive to short-term fluctuations such as cold-start or unexpected timeout [10]. The lower the response time is, the better the performance delivers.

The experiments were conducted over extended period of April 5th - 12th, 2021.

3.2.2 RQ-2 Methodology: As discussed in Section 2.1.1, all FaaS platforms impose upper concurrency limit. Hence, in RQ-2, we aim to study the system's behavior when the workload has reached FaaS' maximum concurrency limit. We also study if a workload smoother design pattern can help to improve the system's performance. Among the proposed design patterns [17, 24], we choose to implement a workload smoother based on the *Queue-Based Load Leveling Pattern* because this pattern is effective when the target system intermittently experiences the high load.

We followed the strategy discussed by Cooper et al. [6] about measuring the response time as throughput increases until the point at which it stops increasing, i.e., the system is saturated. We first sent the workload to FaaS system until a point the throughput reached a consistent level. Then, we stopped and recorded the total number of requests (including the passed and failed). Next, we re-sent the same number of requests to a system with the workload smoother and recorded similar performance metrics. This methodology helps us to evaluate if a workload smoother can improve FaaS' performance.

We chose to experiment the workload smoother with AWS Lambda and Azure Cloud Function since these cloud platforms offer configurable concurrency limits. We did not examine IBM Cloud Function because a concurrency level of 1,000 with no configurable option is beyond the scope of this study. We configured AWS Lambda to have a maximum concurrent executions of 100 using *"reserved concurrency"*. To prevent overloading, we configured the workload smoother so that there were at most 100 concurrent requests sending to AWS Lambda functions. JMeter client was set to test the system with 150, 200 and 250 threads ramped in 10 seconds. These workloads were equivalent to 1.5x, 2x and 2.5x times of the FaaS' capacity and thus would help us to assess the system at saturated level. Microsoft Azure claims that each Azure function instance can serve multiple concurrent requests hence to ensure the same experimental setup, we set Azure function instance to only serve at most one concurrent request. Experiments showed that although the maximum number of instance can reach 200, we only observed about four instances spawned in all experiments. Consequently, we configured our workload smoother so that it would send at most five concurrent requests to Azure cloud function and set JMeter to create five, seven and ten threads (equivalent to 1.25x, 1.75x and 2.5x of FaaS' capacity) ramped in 50 seconds.

We conducted experiments for each cloud provider twice to avoid cloud instability. AWS Lambda systems were assessed on May-5th and May-25th, 2021 and Azure cloud platform was evaluated on May-28th and June-5th, 2021. In these experiments, we measured median response time, coefficient of variation (CV) which is the ratio in percentage between standard deviation over mean response time, the number of passed, failed, total requests, success rate which is the ratio in percentage between passed requests over the total requests, throughput and the number of instances provisioned.

3.2.3 Function Instance Identification: For **RQ-1** and **RQ-2**, it is essential to quantify how many cloud function instances were provisioned hence uniquely identifying the cloud function instance is important. To achieve this goal, we used the following approaches:

For AWS Lambda, we used the unique *"logStreamName"* which is tied to the function executing instance and extracted from the execution context.

For Azure Cloud Function, we queried the Monitoring - Log to retrieve requests' *"customDimensions"* which contained the executing instance identifier.

For IBM Cloud Function, we followed the self-generate mechanism presented by Lloyd et al. [13] to create a universally unique identifier (UUID) for the function instance. When a cloud function is invoked, it checks the local file *"/tmp/host.txt"* for the UUID. If this file does not exist, it means this is a new instance and a UUID is created and stored in this file. To ensure thread safety, we implemented the synchronization with double lock mechanism while creating and storing the UUID. When the *"/tmp/host.txt"* exists, subsequent execution can retrieve the UUID without acquiring the lock to avoid performance degradation.

4 Experimental Setup

Cloud Providers: Based on Eismann et al.[7] FaaS' study, we chose to evaluate on three most popular cloud providers, namely AWS Lambda (AWS), IBM Cloud Function (IBM) and Microsoft Azure Function (Azure) because they occupied majority of FaaS use cases (80%, 10% and 7% respectively). Google Cloud Function only accounted for a small use case percentage (3%) and therefore not generalized enough.

Runtime: We focused on Java in this paper because Java and Node.js are known to be the most popular studied language in FaaS software industry research [20]. Among different versions of Java, we used Java version 8 since it is the commonly supported version on all participating platforms.

Tools: We used Apache JMeter (version 5.2.1) to conduct multi-concurrent client load tests. JMeter is a pure Java application designed to load test functional behavior and measure performance. While benchmarking the cloud functions, we cross-verified how many threads were spawned by JMeter using VisualVM tool [25].

Testing Function: We developed a CPU-intensive cloud function which calculates the factorial of a number similar to Somu et al.[21] experiments. However, our cloud function is a single function compared to their multiple chained cloud functions. We followed the FaaS Principles and Best Practices suggested by AWS Lambda, IBM and Azure cloud providers which recommend that each cloud function should perform only one action and not to make the functions call each other [1, 9, 16]. Our cloud function was designed to compute the factorial using loop iteration and no cache storage was used to ensure performance consistency. We chose 4000 as the testing parameter as this number is large enough to simulate intensive processing. Our source code is published for reproduction purpose [26].

For AWS Lambda and IBM cloud function, we allocated 512MB memory since this is appropriate for the factorial calculation. On Microsoft Azure, we deployed the function on a standalone function app which ran on Linux Operating System. We exposed all testing cloud functions to have HTTPS endpoints for JMeter invocation. Fig. 1 illustrates our **RQ-1** experiments.

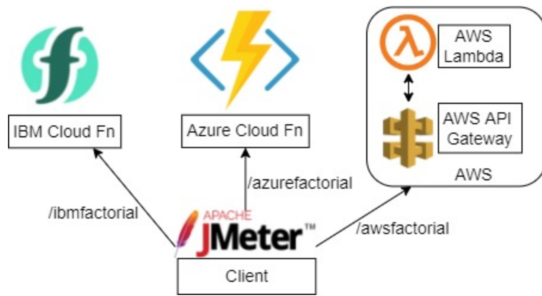

Figure 1: Load and Performance Experiment Benchmark With JMeter On Three Cloud Platforms

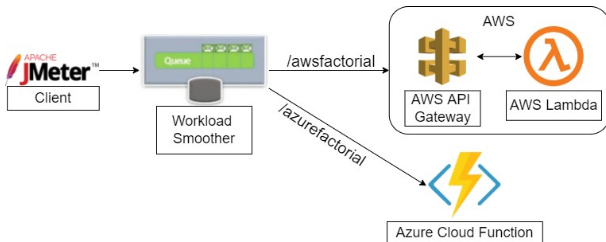

Figure 2: Cloud Functions With Workload Smoother

For **RQ-2**, we re-used the **RQ-1** structure as depicted in Fig. 1 to experiment a system without a workload smoother introduced. Next, we developed a workload smoother application powered by Spring Boot Technology [22] as shown in Fig. 2. Internally, the workload smoother has two fixed-size thread pool executors which execute the submitted tasks using one of the available threads in the pool [18]. Each thread pool executor corresponded to one target FaaS system, i.e., AWS Lambda and Azure Cloud Functions. The thread pool size was configured to align with the maximum concurrent capacity that each FaaS system could handle. Furthermore, each thread pool has implicit unbounded queue which automatically en-queues the request when all the threads are busy. Once a thread becomes available, the request will be de-queued and forwarded to the FaaS system. The workload smoother also exposed two corresponding HTTPS endpoints for AWS Lambda and Azure cloud function and was deployed on U.S-East region (Ohio) EC2 t2.large instance (2 vCPU and 8GB Memory). We used t2.large instance because this instance type can sustain high CPU and network requirements therefore eliminate potential performance bottleneck. It should be noted that deploying the workload smoother on Amazon AWS EC2 environment did not introduce performance bias to AWS Lambda because each service in the system was built as a standalone component and communicated to via HTTPS APIs.

5 Evaluation Results

We describe the experimental results with respect to research question **RQ-1** (Section 5.1) and **RQ-2** (Section 5.2).

5.1 RQ-1: Scalability of FaaS

In this section, we present the scalability and elasticity characteristics of FaaS under different intensity levels.

# of Request	Ramp Up Time											
	AWS				IBM				Azure			
	1s	3s	6s	10s	1s	3s	6s	10s	1s	3s	6s	10s
1000	99	82	54	29	100	108	101	109	4	4	3	4
3000	100	86	67	43	107	108	112	113	5	6	5	5
7000	97	88	70	66	113	105	120	123	7	7	7	7

Table 1: Number of Instances spawned by different Ramp Up Times and Workload Levels

5.1.1 The Number of Function Instances Spawned:

Table 1 shows the number of instances spawned by three cloud providers across different ramp up times and workload levels. When the workload became more intensive by reducing the ramp up time, AWS increased the instances spawned accordingly. This pattern was consistent across three testing workload levels. In addition, when the workload expanded from 1000 to 7000 requests, AWS Lambda exhibited two different patterns. For long ramp up time (6s and 10s), the number of instances increased in accordance with the increase in workload (54 - 70, 29 - 66 instances respectively). However, for short ramp up time (1s and 3s), the number of instances only changed by a small number (99 - 97, 82 - 88 instances).

On the contrary, IBM cloud platform shows a similar number of 100 to 120 instances spawned across the ramp up times. There was no clear pattern in the relationship between changing the ramp up time and more instances getting provisioned. Nevertheless, when workload extended from 1000 to 7000, the number of instances spawned increased accordingly in most cases (three out of four cases except for the ramp up time of 3s).

Microsoft Azure cloud function shows that the number of instances spawned was almost constant across different ramp up times. When the workload level increased from 1000 to 7000, we noticed that Azure cloud platform nearly doubled the number of instances from four to seven.

In this metric, we observed that IBM cloud platform scaled their instance fleet to around 100 - 120 instances in all testing configurations. These values were quite close to 100 concurrent clients

Number of Requests	Ramp Up Time											
	AWS				IBM				Azure			
	1s	3s	6s	10s	1s	3s	6s	10s	1s	3s	6s	10s
1000	900	3,329	3,725	3,728	37	1,050	2,156	9,023	32,341	36,202	25,291	30,735
3000	652	2,638	6,120	6,790	122	8,333	17,062	10,355	43,293	51,583	43,208	42,056
7000	718	2,593	7,915	10,942	26,589	15,359	24,664	33,319	68,036	63,018	62,689	62,349

Table 2: Ramp Duration To Expected Number of Instances (in ms)

Number of Request	Ramp Up Time											
	AWS				IBM				Azure			
	1s	3s	6s	10s	1s	3s	6s	10s	1s	3s	6s	10s
1000	14,648	10,870	8,408	5,578	4,193	4,467	4,056	1,354	1,679	1,369	1,741	1,938
3000	20,258	25,564	18,036	13,349	2,850	3,748	7,047	5,712	3,174	3,229	3,737	3,840
7000	34,291	34,440	27,299	19,592	13,338	12,812	14,596	9,810	5,667	5,837	5,866	6,089

Table 3: Cloud Function Throughput (requests per minute)

Number of Request	Ramp Up Time											
	AWS				IBM				Azure			
	1s	3s	6s	10s	1s	3s	6s	10s	1s	3s	6s	10s
1000	219	204	180	113	325	355	415	376	2,644	2,561	2,926	2,878
3000	125	116	98	88	384	380	360	354	1,346	1,241	1,165	1,125
7000	97	89	90	98	303	296	318	320	768	754	773	754

Table 4: FaaS Median Response Time (ms)

which leads us to the conclusion that IBM cloud platform provisioned the number of instance similar to the number of concurrent clients. AWS Lambda exhibited similar characteristic only when the workload was intensive (i.e., 1 second). Other less intensive ramp up times show a smaller number of instances spawned. Azure Cloud Function scaled their fleet to only a small number of four to seven instances, this could be due to each Azure cloud function instance can process multiple concurrent requests hence a small number of instances is adequate to process the workload [15].

5.1.2 The Ramp Duration To Expected Number of Cloud Function Instances: Table 2 shows the duration cloud providers ramped their fleet to expected number of instances. For AWS Lambda, we noted that when the ramp up time reduced, the time taken to increase the fleet decreased three to 15 times (3,728ms vs 900ms, 10,942ms vs 718ms). This pattern was observed on all workload levels. In the best scenario, we recorded that AWS Lambda could scale their fleet to 100 instances within 652ms.

Similar to AWS Lambda, IBM cloud platform slowly provisioned and deployed the function instances when the ramp up time was long (6s and 10s). However, when this duration was shorter (1s and 3s), IBM cloud platform increased the instances 84 to 243 times faster (9,023ms vs 37ms, 10,355ms vs 122ms). In one best scenario, we observed this platform ramped 100 instances in just 37ms. Nevertheless, we also noticed that the IBM performance was inconsistent. In 1s-ramp up time and 7000-request level, it took IBM more than 26 seconds to increase the fleet capacity. We hypothesize that this might be due to the VM cold provisioning which takes more time to provision a new instance [13].

Different from AWS Lambda and IBM Cloud Function, Azure cloud platform required 30 - 60 seconds to deploy more instances to the optimal level. This might be due to the characteristic that each Azure instance can process multiple requests concurrently hence Azure cloud platform only considers to add extra instances at a later stage of the test.

In this metric, we noted AWS Lambda and IBM cloud function demonstrated similar behavior. When the workload arrives at intensive pace, these platforms quickly provisioned more cloud function instances in short period which shows good elasticity. Azure cloud function operated differently in which each instance could serve multiple concurrent requests and hence the duration to scale up the fleet was longer.

5.1.3 The System's Throughput: Table 3 presents the throughput measured in requests per minute on all cloud platforms. We observed that AWS Lambda produced a consistent throughput increase when the workload became more intensive. This pattern demonstrates a good performance since more resources added to the system should result in more requests processed and consequently increased the system's throughput.

IBM cloud platform on the contrary, exhibited an inconsistent throughput changing pattern. The results fluctuated because in some cases, we noticed a number of requests returned after 30 - 35 seconds. We hypothesize that these calls might be served by newly provisioned VM and it would require more time to complete. As a result, the overall execution duration increased and further reduced the system's throughput.

Azure cloud function showed a consistent throughput across different ramp up times. When the workload level increased from 1000 to 7000, more cloud function instances were added to the system leading to a boost in throughput.

5.1.4 The Median Response Time: Table 4 displays the median response time recorded on all testing platforms. For AWS Lambda, at low and medium level workload (i.e., 1000, 3000-request), when the ramp up time decreased, the response time increased around 42% - 93% (88ms vs 125ms, 113ms vs 219ms). Nonetheless, when the workload level was high (i.e., 7000-request), there was no major difference between the response times. This result might be due to the cold-start, where FaaS system needs to provision and initialize the function instances before execution. This activity introduces

additional delay in response time for early requests (i.e., cold-start requests), but subsequent requests are processed faster (i.e., warm-start requests). Among the three workload sizes, the 7000-request had more warm-start requests than the other two levels therefore the median response time of this workload level was less susceptible to cold-start and hence delivered stable median response time.

IBM cloud platform produced a consistent median response time across the experiments. This could be due to the fact that IBM cloud platform provisioned a similar number of instances in all testing scenarios and therefore the workload was equally distributed, thus producing stable results.

Microsoft Azure cloud function exhibited a consistent median response time across different ramp up times. Nevertheless, when the workload level increased, the median response time was greatly improved with a reduction from 2,600ms to 1,200ms then 760ms. This improvement pattern was similar to AWS Lambda mentioned above.

Summary: Through RQ-1 experiments, we conclude that though providing auto-scaling, each platform adopts different scaling strategy. AWS Lambda and IBM provision the number of instances similar to the number of concurrent requests whereas Azure platform only spawns a small number of instances since each instance can process multiple concurrent requests.

When the workload becomes more intensive, all three platforms demonstrated an increase in computing resource provisioned and hence, elevate the processing capability (higher throughput). System's median response time might be impacted if the workload size is small, however, in the long run, when the workload size is larger, median response time will be stable.

5.2 RQ-2: FaaS Under Saturation and Performance Improvement Patterns

Here we present the FaaS' performance at saturated level and the advantages gained by employing a workload smoother.

Metrics	1.5x Capacity		2x Capacity		2.5x Capacity	
	Direct	WLSM	Direct	WLSM	Direct	WLSM
Median Resp. Time (ms)	80	201	84	268	103	372
Coefficient of Variation	49.60%	56.03%	51.41%	47.50%	62.69%	39.56%
Throughput (req./min)	96,036	37,088	118,873	38,706	97,707	36,378
Number of Instances	100	95	101	98	101	96
Pass	349,206	351,600	364,624	380,400	298,005	371,750
Fail (HTTP-429)	2,288	0	15,702	0	73,558	0
Total Requests	351,494	351,600	380,326	380,400	371,563	371,750
Success Rate	99.35%	100.00%	95.87%	100.00%	80.20%	100.00%

Table 5: Performance Comparison on AWS Lambda FaaS without (Direct) and with Workload Smoother (WLSM).

5.2.1 AWS Lambda with Workload Smoother: Table 5 shows the comparison between direct invocation to AWS Lambda (Direct) and through a workload smoother (WLSM). Overall, direct invocation to AWS Lambda resulted in better median response time, it took the cloud function 80 - 103ms to calculate the result. When there were more concurrent clients added, response time fluctuated with CV increased from 49% to 63%. AWS Lambda provisioned about 100 - 101 instances as expected when we configured *"reserve concurrency"* to 100. The system throughput could reach 118,000 requests per minute. Nevertheless, a large number of requests were throttled with "Too Many Request" error in 2x- and 2.5x-capacity setting (15,000 and 73,000 requests, respectively). These results demonstrate that when AWS Lambda is overloaded, excessive requests will be throttled hence reduced the success rates.

In contrast, by having a workload smoother, the system could achieve 100% success rates in all scenarios. Excessive requests were queued and later de-queued for processing hence no request was throttled. However, since the requests had to pass through one additional component and potentially stayed in the queue, certain performance metrics were lower compared to direct invocation. In particular, it took a request 200 - 370ms to complete. Although it was much longer to process a request, the system response times were less fluctuated with CV reduced from 56% to 48% then 40%. There were 96 instances spawned by cloud platform which was slightly less than the 100 concurrent clients configured in the workload smoother. This gap might be attributed to workload intensity reduction by adding the workload smoother.

Metrics	1.25x Capacity		1.75x Capacity		2.5x Capacity	
	Direct	WLSM	Direct	WLSM	Direct	WLSM
Median Resp. Time (ms)	84	151	82	154	85	187
Coefficient of Variation	59.22%	34.42%	53.98%	33.97%	156.61%	24.36%
Throughput (req./min)	2,942	1,755	3,866	2,413	3,843	2,921
Number of Instances	4	4	4	4	4	4
Pass	15,167	15,651	22,174	26,444	15,543	25,314
Fail (HTTP 429)	545	64	4,473	205	9,929	166
Total Requests	15,712	15,715	26,647	26,649	25,472	25,480
Success Rate	96.53%	99.59%	83.21%	99.23%	61.02%	99.35%

Table 6: Performance Comparison on Azure FaaS Cloud Function without (Direct) and with Workload Smoother (WLSM).

5.2.2 Azure Function with Workload Smoother: Table 6 shows the performance comparison of Azure cloud functions with workload smoother added. Similar to AWS Lambda, the direct invocation produced better median response time. On average, the task was completed in 82 - 85ms. Nonetheless, the CV increased from 59% to 156% which shows major response time fluctuation. In all experiments, we observed Azure cloud platform only spawned four instances regardless of workload intensity. The throughput of the Azure cloud functions were between 3,000 - 3,800 requests per minute. The success rates deteriorated when the workload became more intensive, 96% - 83% - 61% for 1.25x, 1.75x and 2.5x-capacity, respectively. Considering that there were only four instances spawned (which was far below the 200-instance capacity claimed by Azure)

and a large number of requests were throttled, this raised a concern about the Azure cloud platform's auto-scaling algorithm. Cloud provider did not add more instances to address the workload but throttled the excessive requests while the number of instances had not reached the maximum level.

On the contrary, when workload smoother was added, the system achieved more than 99% success rates in all three experiments. The 1% failure might be due to the mismatch between the number of threads in workload smoother and the number of instances provisioned. We configured five working threads in the workload smoother while Azure cloud platform only spawned four instances. As a trade-off for improving the success rates, median response time was almost doubled compared to direct invocation, one request was processed in 150 - 187ms. Nevertheless, the CV in all three experiments were reduced which means response time were relatively consistent (34% - 33% - 24%). Throughput was reduced in between 1,700 to 3,000 requests per minute.

Summary: RQ-2 experimental results show that when the FaaS cloud function system is saturated, excessive requests will be throttled hence causing low success rates. To address this issue, we can add a workload smoother to queue these excessive requests and thus improve the system's success rate. The more intensive the workload is, the higher the success rate can be achieved. Nonetheless, certain performance metrics such as median response time and throughput will be reduced because of the waiting time in the queue.

6 Threats To Validity

In this section, we outline the potential internal, external and construct threats that might affect our results.

6.1 - Internal Validity: Our results may be impacted by the cloud platform's memory configuration. AWS Lambda is known to allocate more powerful CPU if the cloud function is allocated higher memory.

6.2 - External Validity: Our experiments were conducted on free-tier instances hence the results may change on paid subscriptions. In addition, our findings might also not apply to other cloud platform such as Google Cloud Functions. Further research needs to be done to characterize this platform.

The testing experiments were carried out on platforms run on Linux OS hence the findings may not be the same on Windows OS. Moreover, there might be other use cases which implement the cloud function different from ours hence the finding may also vary in these cases.

6.3 - Construct Validity: There might be unexpected delay in network communication between client and FaaS function. We had conducted our experiments twice to average and mitigate these delays.

7 Discussion and Conclusion

In this research work, we characterized the scalability and elasticity of FaaS implemented on three most popular cloud platforms. Experimental results show that different cloud providers adopt distinct scaling strategies. Nevertheless, all examining cloud platforms demonstrated good auto-scaling feature which is when the workload becomes intensive, more resources are automatically added and consequently increased the system's capacity. All cloud platforms impose upper concurrency limits a cloud function can have and some platforms provide options for manual configuring the concurrency level. Furthermore, we examined the usefulness of applying workload smoother design pattern when FaaS' capacity was saturated. The prototype showed major improvement from 60 - 80% to 99 - 100% success rate in trade-off for certain performance metrics such as median response time and throughput. This improvement has emphasized the importance of having a request queue with configuring options implemented implicitly at the cloud provider's side to prevent intermittent throttling issue. Therefore, cloud providers such as AWS Lambda and IBM Cloud Function should include this feature to offer better performance achievement to cloud subscribers.

References

[1] amazon.com. 2021. Best practices for organizing larger serverless applications. https://aws.amazon.com/blogs/compute/best-practices-for-organizing-larger-serverless-applications/. Last accessed: 01/25/2022.

[2] amazon.com. 2021. Building Applications with Serverless Architectures. https://aws.amazon.com/lambda/serverless-architectures-learn-more/. Last accessed: 09/08/2021.

[3] amazon.com. 2021. Lambda Function Scaling. https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html. Last accessed: 09/13/2021.

[4] amazon.com. 2021. Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/. Last accessed: 09/08/2021.

[5] apache.org. 2021. Apache JMeter. https://jmeter.apache.org/. Last accessed: 01/25/2022.

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[7] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2021. A Review of Serverless Use Cases and their Characteristics.

[8] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. 2013. Elasticity in Cloud Computing: What It Is, and What It Is Not. In *10th International Conference on Autonomic Computing (ICAC 13)*. USENIX Association, San Jose, CA, 23–27. https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst

[9] ibm.com. 2021. FaaS Principles and Best Practices. https://www.ibm.com/cloud/learn/faas. Last accessed: 01/25/2022.

[10] Zhen Ming Jiang and Ahmed E. Hassan. 2015. A Survey on Load Testing of Large-Scale Software Systems. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1091–1118. https://doi.org/10.1109/TSE.2015.2445340

[11] Jörn Kuhlenkamp, Sebastian Werner, Maria C. Borges, Dominik Ernst, and Daniel Wenzel. 2020. Benchmarking Elasticity of FaaS Platforms as a Foundation for Objective-Driven Design of Serverless Applications. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic). Association for Computing Machinery, New York, NY, USA, 1576–1585. https://doi.org/10.1145/3341105.3373948

[12] Sebastian Lehrig, Hendrik Eikerling, and Steffen Becker. 2015. Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics. In *2015 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. 83–92. https://doi.org/10.1145/2737182.2737185

[13] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 159–169.

[14] Horácio Martins, Filipe Araujo, and Paulo Rupino Cunha. 2020. Benchmarking Serverless Computing Platforms. *Journal of Grid Computing* 18 (12 2020), 691–709. https://doi.org/10.1007/s10723-020-09523-1

[15] microsoft.com. 2021. Azure Functions HTTP output bindings. https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook-output#hostjson-settings. Last accessed: 09/09/2021.

[16] microsoft.com. 2021. Best practices for performance and reliability of Azure Functions. https://docs.microsoft.com/en-us/azure/azure-functions/functions-best-practices. Last accessed: 01/25/2022.

[17] microsoft.com. 2021. Cloud Design Patterns. https://docs.microsoft.com/en-us/azure/architecture/patterns/. Last accessed: 09/30/2021.

[18] oracle.com. 2021. Java ThreadPoolExecutor. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html. Last accessed: 09/26/2021.

[19] Andrei Palade, Aqeel Kazmi, and Siobhán Clarke. 2019. An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge. In *2019 IEEE World Congress on Services (SERVICES)*, Vol. 2642-939X. 206–211. https://doi.org/10.1109/SERVICES.2019.00057

[20] Joel Scheuner and Philipp Leitner. 2020. Function-as-a-Service performance evaluation: A multivocal literature review. *Journal of Systems and Software* 170 (2020), 110708. https://doi.org/10.1016/j.jss.2020.110708

[21] Nikhila Somu, Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications. In *2020 International Conference on COMmunication Systems NETworkS (COMSNETS)*. 144–151. https://doi.org/10.1109/COMSNETS48256.2020.9027346

[22] spring.io. 2021. Spring Boot. https://spring.io/projects/spring-boot. Last accessed: 09/26/2021.

[23] Uma Tadakamalla and Daniel Menascé. 2019. *Characterization of IoT Workloads*. Springer International Publishing, 1–15. https://doi.org/10.1007/978-3-030-23374-7_1

[24] Davide Taibi, Nabil El Ioini, Claus Pahl, and Jan Niederkofler. 2020. Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review. In *Proceedings of the 10th International Conference on Cloud Computing and Services Science - CLOSER,*. https://doi.org/10.5220/0009578501810192

[25] visualvm.github.io. 2021. VisualVM Tool. https://visualvm.github.io/. Last accessed: 01/25/2022.

[26] zenodo.org. 2021. Factorial Cloud Functions. https://doi.org/10.5281/zenodo.5865550. Last accessed: 12/23/2021.