Benchmarks Elasticity of FaaS Platforms as a Foundation for Objective-driven Design of Serverless Applications

Jörn Kuhlenkamp  
ISE, TU Berlin  
Berlin, Germany  
jk@ise.tu-berlin.de

Sebastian Werner  
ISE, TU Berlin  
Berlin, Germany  
sw@ise.tu-berlin.de

Maria C. Borges  
ISE, TU Berlin  
Berlin, Germany  
mb@ise.tu-berlin.de

Dominik Ernst  
TU Berlin  
Berlin, Germany  
de@ise.tu-berlin.de

Daniel Wenzel  
TU Berlin  
Berlin, Germany  
dw@ise.tu-berlin.de

Preprint, to appear at The 35th ACM SIGAPP Symposium on Applied Computing (SAC’20), March 30-April 3, 2020, Brno, Czech Republic; final version available at https://doi.org/10.1145/3341105.3373948

ABSTRACT
Application providers have to solve the trade-off between performance and deployment costs by selecting the “right” amount of provisioned computing resources for their application. The high value of changing this trade-off at runtime fueled a decade of efforts by industry and research to develop elastic applications. Despite these efforts, the development of elastic applications still demands significant time and expertise from application providers.

To address this demand, FaaS platforms shift responsibilities associated with elasticity from the application developer to the cloud provider. While this shift is highly promising, FaaS platforms do not quantify elasticity; thus, application developers are unaware of how elastic FaaS platforms are. This lack of knowledge significantly impairs effective objective-driven design of serverless applications.

In this paper, we present an experiment design and corresponding toolkit for quantifying elasticity and its associated trade-offs with latency, reliability, and execution costs. We present results for the evaluation of four popular FaaS platforms by AWS, Google, IBM, and Microsoft, and show significant differences between the service offers. Based on our results, we assess the applicability of the individual FaaS platforms in three scenarios under different objectives: web serving, online data analysis, and offline batch processing.

CCS CONCEPTS
- Applied computing → Event-driven architectures; Service-oriented architectures; Software and its engineering → Cloud computing; Software performance; Computer systems organization → Cloud computing.

KEYWORDS
Serverless, Benchmarking, Experimentation, Elasticity

1 INTRODUCTION
Serverless Computing, or Function-as-a-Service (FaaS), offers a new alternative to develop cloud-based applications and has since emerged as a popular choice among application providers. Instead of using the cloud infrastructure directly, developers provide short-running code in the form of functions to be executed by a FaaS platform provider. This simplified programming model lets developers create applications without the burden of server-related operational tasks such as provisioning, managing, or scaling of resources.

However, by relinquishing control of the infrastructure, developers have to trust that the FaaS platform provider will manage the application in accordance with their application objectives. FaaS platforms often claim that the provided services can scale dynamically under volatile workloads. However, without knowledge of the underlying architecture or the provider’s load balancing strategies, it is hard for developers to verify this claim. Without explicit information, application developers must select and use a FaaS platform based on gut feeling and intuition. We argue that this significantly impairs the objective-driven design of serverless applications; thus, the dissemination of serverless applications in general.

Consequently, the need to benchmark different characteristics and qualities of FaaS platforms has been recognized by research and industry [19, 21, 26]. Elasticity has been highlighted by Jonas et al. [11] to be of particular interest, due to the unpredictable performance of cloud functions observed to date.

Here, we advocate for a comparison between FaaS platforms under volatile workloads. Such a comparison will allow developers to assess the impact of such workloads on specific objectives set during the design phase. With our work, we provide the first contributions to answer the research question:

Does the quality of a FaaS platform change from a client-side perspective under volatile workloads, and is this change relevant in an application context?

We build on our extensive expertise and previous work on quality-driven design and evaluation of cloud-based systems [2, 3, 14, 17] and propose a new experiment design and corresponding toolkit for application developers and researchers to use in the experimental evaluation.

Further, based on the results of the experiments, we discuss the current applicability of the various FaaS Platforms for different application scenarios. Thus, in our work, we present the following contributions over the state-of-the-art:

- C1: A novel experiment design and corresponding toolkit for the assessment of FaaS platform quality under volatile workloads from a client-side perspective.
• C2: An experiment-based evaluation of four major FaaS platform providers based on C1.
• C3: A discussion of the applicability of FaaS platforms in different application scenarios based on results from C2.

The remainder of the paper is structured as follows: we present relevant background (sec. 2), give a refined problem statement (sec. 3), discuss related work (sec. 4), present our experiment design (sec. 5) and results (sec. 6), discuss applicability of FaaS platforms in different scenarios (sec. 7) and conclude in sec. 8.

2 BACKGROUND
We present background on FaaS platforms and experiment-driven evaluation of cloud-based systems and applications.

2.1 FaaS Platforms
FaaS are platforms that facilitate general serverless computing. FaaS is available from all major cloud providers, with offerings such as AWS Lambda or Google Cloud Functions. These providers manage and run serverless infrastructure in their massive data centers. Besides these commercial offerings, many open source FaaS solutions exist (e.g., Fission, Knative), which can run on commodity hardware.

All these implementations share the following properties: First, the unit of computation, a function, is defined by a developer with a deployment package. The deployment package contains all the code and libraries necessary to run the function. The FaaS platform provides and manages the operating system and the function program runtime, e.g., Python or NodeJS used to execute that function.

Second, the function configuration defines the environment and runtime constraints of a function. The function configuration contains the events that activate the execution of a function in the platform, referred to as a function trigger. Furthermore, the function configuration also describes the function memory and function timeout, which governs the maximum amount of resources a function can use.

Lastly, most FaaS platforms support many different event triggers, such as HTTP requests or platform events such as the insertion of a new value in a managed database. The platforms, therefore, have strict limitations on the input and output interfaces, or handlers, of a function.

FaaS platforms also have similar lifecycles regardless of implementation. Generally, a function is only provisioned and run when an event triggers an execution. After execution, the function runtime is kept alive only for a short period before it is destroyed again. A function runtime, however, is reused, should a new event arrive before the runtime is destroyed. On the other hand, it is also common that no runtime is provisioned, and therefore no resource is used.

2.2 Experimentation
Our experiment methods are based on a mix of (i) the combination of system performance evaluation and Design of Experiments (DoE) proposed by [12] and (ii) concepts proposed in [3, 23].

The goal of an experiment is the evaluation of a system under test (SUT). An experiment typically involves two roles. An experiment is requested by a decision maker and executed by an experimenter.

An objective is a non-functional property or quality of a SUT that is of interest to a decision-maker. The term experiment environment describes all assumptions about the environment of a SUT that are not under the control of a decision-maker. The experiment environment is encoded in environment variables. An experiment is executed for a well-defined time, also called a run.

During a run, the SUT is observed. Observations that are persistent are referred to as measurements. Based on measurements, high-level metrics are calculated that characterize one or more objectives. During a run, the experimenter executes one or more treatments, a controlled change of a SUT. Treatments are encoded in treatment variables. In contrast to environment variables, treatment variables are under the control of a decision-maker.

3 PROBLEM STATEMENT
Application developers design applications to fulfill application objectives. Examples of such application objectives are targets for throughput, latency, reliability, variable execution costs for single computations, and fixed deployment costs.

Software architecture is about deferring decisions by providing options [6]. These options allow changing an application objective in the future under known conditions. We argue that these options should have no impact on other application objectives or well-defined impacts on other application objectives that are acceptable within a specific application context.

For example, using a NoSQL database with a configurable consistency level such as Cassandra allows changing consistency in the future, thus giving the option to increase consistency objectives according to future demands with a quantifiable negative impact on execution latency [12].

FaaS platforms promise the core capability to execute code without requiring an application provider to manage resource allocations [11]. Thus, a cloud automatically provisions resources to execute application code with incoming events. In an application architectural context, this capability implies that a FaaS platform provides an option for throughput-based application objectives. We refer to this option as elasticity.

Elasticity is highly valuable for many types of applications with unknown or changing throughput objectives — for example, web serving, explorative data analysis, and periodical batch processing. While the available choices for fully managed FaaS platforms and open-source FaaS platforms are considerable and continue to increase, and more applications migrate to FaaS platforms, application developers do not have a clear understanding of a core option that drives architectural design decisions: the option on variable throughput objectives.

We argue that such a clear understanding of elasticity includes: (i) quantifying the amount by which a throughput objective can change in a particular time, and (ii) quantifying the impact of changing throughput objectives on other application objectives such as latency, reliability, and execution costs. In conclusion, we define our problem statement as follows:
• P1: Do FaaS platforms deliver on the promise to process volatile target throughput?
• P2: What are the impacts of volatile target throughput on latency, reliability, and execution costs?
• P3: What are the implications of P1 and P2 on the applicability of FaaS platforms in different scenarios?

4 RELATED WORK
This paper addresses the experiment-driven evaluation of FaaS platform elasticity. We discuss related work from the perspective of benchmarking elasticity of traditional cloud platforms based on virtual machines and benchmarking FaaS platforms in general.

There are many publications on the design objectives of system benchmarks [3, 4, 15]. These benchmarks all share common well-established properties, regardless of the difference in objectives. Namely, all benchmarks need to be relevant, repeatable, fair, understandable, and portable [3, 7, 25]. These properties must also apply to the evaluation proposed in this paper.

Furthermore, research is doing much work with regards to cloud service benchmarking [2, 3, 14]. Analyzing cloud services comes with additional challenges, such as less accessible measurement points and intransparent changes of services over time. Therefore, benchmarking FaaS platforms must take the lessons learned in cloud service benchmarking into account. Thus, we base our experiment design on these established design guidelines for benchmarks.

In literature, a body of work addresses elasticity benchmarking of traditional, VM-based cloud platforms [5, 8, 13, 27]. All experiment designs assume that clients have control over the provisioning of computing resources. This assumption does not hold in the context of FaaS platforms; thus these approaches are not applicable for benchmarking FaaS platforms.

Industry and research already started to use experiment-based evaluations to analyze FaaS platforms [9, 18, 20–22]. Most of these efforts focus on performance [16]. However, the evaluation of elasticity in FaaS platforms has largely been ignored in literature. A first proposal by Lee et al. [19] evaluates latency changes in response to volatile throughput. The proposal motivates our work, and we significantly extend its scope.

5 EXPERIMENT DESIGN
In this section, we describe our experiment design, which entails a detailed description of experiment goals, SUT setup, and workloads. Associated goals with this detailed design are understandability, verifiability, and reproducibility of results by interested readers or third parties.

5.1 Experiment Goal
The goal of our experiment design is to quantify the elasticity of FaaS platforms from a client-side perspective. More precisely, that is the ability of a FaaS platform to process volatile target workloads, i.e., the number of events that are processed, considering a provisioned function, associated impacts on other client-observable qualities, and execution cost. Based on our results, we want to guide application developers to account for FaaS platform-specific behavior to enable objective-driven design decisions.

5.2 System Under Test
5.2.1 FaaS Platforms. We evaluate four fully managed FaaS platforms in different data centers in London: Amazon Web Services Lambda (AWS), Google Cloud Functions (GCF), IBM Cloud Functions (ICF), and Microsoft Azure Functions (MAF).

5.2.2 Memory Size. Through preliminary experiments, we observed out of memory errors for smaller memory configurations than 1024 MB in ICF. Thus, we set the memory size of function handlers to 1024 MB for AWS, GCF, and ICF. MAF manages memory allocations implicitly; thus, clients are not able to configure memory size explicitly.

5.2.3 Handler Timeout. Overall application scenarios, we consider a client-side request-response latency of 30 seconds as the tolerable maximum. Thus, we configure a hard handler timeout of 30 seconds.

5.2.4 Handler Implementation. The target programming platform for all handler implementations is Node.js in the major version 10. The function handler takes a natural number n as handler input and returns true or false as handler output depending on whether n is a prime number based on the Sieve of Eratosthenes algorithm. We use a non-optimized basic implementation, which has a time complexity of \(O(n \log \log n)\) and a memory requirement of \(O(n)\) for the two-dimensional array indicating which numbers are prime. Consequently, the function handler can be characterized as CPU and Memory intensive. The handler implementation does not cache any results.

5.2.5 Function Trigger. We trigger all functions via a public HTTPS endpoint. Thus, we use the corresponding auxiliary services for each FaaS platform to provision the HTTPS endpoint. For example, we use the API Gateway service for AWS. Each auxiliary service is configured through the serverless framework using default settings.

5.3 Workloads
Our workload model uses three sequential phases (P0-P2): warmup phase (P0), scaling phase (P1), and cooldown phase (P2). Besides, we split each phase into one-second intervals. In each one-second interval, clients send a well-defined request target to the SUT. We refer to this target as target requests per second (trps). In our workload design, a single client request results in a single event that a FaaS platform has to process.

The warmup phase P0 lasts for 60 seconds and ensures that a SUT is in a well-defined state before introducing any changes. During P0, we apply a constant number 0 or 60 trps. The reason behind this choice is that a FaaS platform must initialize new execution environments which can result in visible impacts for clients - a phenomenon introduced in literature as "cold starts" [22]. With a value of 0 trps in P0, we try to capture this effect in P1.

Similar to the warmup phase P0, the scaling phase P1 lasts for 60 seconds. During the scaling phase P1, we increase trps for each interval by a constant factor of 0.5, 1, or 2. Thus, we increase trps for 60 seconds linearly.

In response to a scaling phase P1, a SUT can potentially change its behavior over a long period. Thus, the duration of the cooldown phase (P2) is 180 seconds. P2 allows monitoring a SUT for a more extended period after changing its environment in P1. We argue
that a FaaS platform should be able to respond to workload changes within 3 minutes. Similar to P0, we apply a constant number of trps in P2. The number of trps equals the trps of the last interval of the scaling phase. In summary, we expose a SUT to a stable environment (P0), change the environment in a well-defined way (P1), and continue to observe the SUT in a stable environment (P2).

Table 1 summarizes all five workloads WL0-WL4 used in our experiments.

Table 1: Target Requests per Second (trps) as a Function of Elapsed Time $t$ for the Three Phases P0 (Warmup), P1 (Scaling), and P2 (Cooldown) for the Five Workloads WL0-WL4.

<table>
<thead>
<tr>
<th>Name</th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>WL0</td>
<td>0</td>
<td>$0 + [0.5 \times t]$</td>
<td>15</td>
</tr>
<tr>
<td>WL1</td>
<td>0</td>
<td>$0 + 1.0 \times t$</td>
<td>60</td>
</tr>
<tr>
<td>WL2</td>
<td>0</td>
<td>$0 + 2.0 \times t$</td>
<td>120</td>
</tr>
<tr>
<td>WL3</td>
<td>60</td>
<td>$60 + [0.5 \times t]$</td>
<td>75</td>
</tr>
<tr>
<td>WL4</td>
<td>60</td>
<td>$60 + 1.0 \times t$</td>
<td>120</td>
</tr>
</tbody>
</table>

With each request, we want to trigger an execution that should take around 1-2 seconds. Thus, the function handler input is 7.5m plus a random number that is drawn uniformly from a small range of [0,1000]. We use the range to avoid potential caching mechanisms within a SUT to improve the fairness of the workload.

5.4 Measurement Points

A measurement point observes a SUT or the experiment environment during an experiment run. Table 2 summarizes all measurement points that are observed during a run.

5.4.1 Result. We create a unique id (RId) for each request on the client. For each response, we record the HTTP status code (RCode) and the result of the handler (RResult) to determine if a request was successful.

5.4.2 Timestamps. For each request, we measure up to four timestamps with millisecond precision; the start of a request (RStart) and the result of the handler (RResult) to determine if a request was successful.

5.4.3 Execution Environment. We track the execution environment (container) and the host of the execution environment (host) to quantify the internal state of a FaaS platform. As this is not trivial from a client-side perspective, we use an approach based on Lloyd et al. [20]. Each execution checks for a global field that contains a unique identifier for the container (CId). If the field does not exist, i.e., for the first execution in a container, we create the field. Similarly, we obtain a unique host identifier (HId), by inspecting /proc/uptime. Because this is not possible for MAF, we inspect the network interfaces and hostname to obtain the HId.

5.4.4 Internal Validation. We measure the operating system (COs) and programming platform (CPlat) of a container for internal validation of an experiment.

5.5 Metrics

Based on our measurements, we calculate two sets of high-level metrics. We use client-side metrics to analyze impacts on application objectives. We use platform-side metrics to characterize the internal behavior of a FaaS platform to develop mitigation strategies for application designers.

5.5.1 Reliability. We consider a request successful if the RCode is 200, and the RResult is correct. For all other cases, we denote a request as failed. We terminate open requests on the client-side after 30 seconds and count the request as failed. For each one-second interval of an experiment run, we report the total number of successful requests RSuccess and failed requests RFailed. We argue that the ratio of RSuccess and RFailed represents an indicator of reliability application objectives.

5.5.2 Request-Response Latency. For each request, we calculate the request-response latency RLat as the difference between ELat and RStart. For each one-second interval of an experiment run, we report the total number of successful requests RSuccess and failed requests RFailed. We use ELat as an indicator of latency-based performance application objectives.

5.5.3 Request Throughput. For each one-second interval of an experiment run, we report the total number of successful requests RSuccess to characterize the throughput of a function. RSuccess is an indicator of throughput-based application objectives.

5.5.4 Execution Cost. To characterize the impacts on application cost objectives, we calculate the execution cost of each request. The pricing models of all four SUTs are a linear function of execution time ELat - the difference between EE and EStart. Table 3 summarizes the different pricing models. For each one-second interval of an experiment run, we report the total number of successful requests RSuccess and failed requests RFailed based on all successful requests as an indicator of execution costs. While mean latency is generally a non-significant metric

---

**Table 3: Pricing Model for Execution Costs. MAF is an Estimate due to Limited FaaS Platform Exposure.**

<table>
<thead>
<tr>
<th>Provider</th>
<th>EVar</th>
<th>EFix</th>
<th>ECost</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS^1</td>
<td>16.67</td>
<td>0.2</td>
<td>(16.67 \times \text{ELat} + 0.2)</td>
</tr>
<tr>
<td>GCF^2</td>
<td>16.50</td>
<td>0.4</td>
<td>(16.50 \times \text{ELat} + 0.4)</td>
</tr>
<tr>
<td>ICF^3</td>
<td>17.00</td>
<td>0.0</td>
<td>(17.00 \times \text{ELat} + 0.0)</td>
</tr>
<tr>
<td>MAF^4</td>
<td>16.00</td>
<td>0.2</td>
<td>(16.00 \times \text{ELat} + 0.2)</td>
</tr>
</tbody>
</table>

5.6 Plots

In our result section, we make heavy use of two types of plots to visualize an experiment run. We refer to the two plots as client-view and platform-view. Due to their non-trivial nature, we discuss each plot in detail. We do not show measurements for the last 20 seconds of each run (see sec. 6 for explanation).

5.6.1 Client-View Plot. The design goal of the client-view plot is a combined view of all client-side metrics for a workload run. On the x-axis, the client-view plot shows the elapsed time of an experiment run in one-second intervals and the three workload phases. The plot shows the total number of trps, RFailed and minimum, median, and maximum RLat and ELat.

5.6.2 Platform-View Plot. The design goal of the platform-view plot is a combined view of (i) execution environments that are available to process incoming events and (ii) the heat of individual execution environments. Similar to the client-view plot, the platform-view plot shows the elapsed time of an experiment run in one-second intervals and the three workload phases.

We indicate the value CHeat or HHeat with different intensities of red. The highest intensity indicates a HHeat \(\geq 12\) for a one-second interval. If the min ELat \(\geq 1\), a CHeat or HHeat \(\geq 3\) indicates parallel executions. Therefore, executions are potentially less isolated and can cause noisy neighbor effects.

With increasing values for RFailed, less data is available to generate the specific heat map. Thus, corresponding plots must be interpreted carefully.

5.7 Experiment Artifacts

The functions were deployed using the serverless framework (v. 1.3.5). On Lambda, GCF & ICF, the calculation was executed on the main thread. MAF computes multiple requests in the same container concurrently. To leave the main thread unblocking, calculations were performed in other threads. Requests were sent using the artillery.io framework. We deployed the workload clients on a TU Berlin virtual machine with an Intel Xeon E3-1220 CPU and 64 GB Memory, located in Germany. The experiment is fully automated through a single script, available on GitHub^7.

All measurements are available in an aggregated relational dataset to enable easy custom analysis for other researchers and practitioners. The code to execute all analysis and generate figures is available as Jupyter-Notebook^3.

^7https://ipython.org/notebook.html
To increase understandability, verifiability, and reproducibility, all code artifacts and results are available at GitHub\textsuperscript{7}. For easy referencing in publications, all measurements and figures are available at Zendoo\textsuperscript{6}.

6 EXPERIMENT RESULTS

We present and discuss results for experiments based on the experiment design presented in section 5.

6.1 Individual FaaS Platforms

We present our experiment results for the four FaaS platforms AWS, GCF, ICF, and MAF, and the five workloads WL0-WL4. We visualize select experiment runs using client-view and platform-view plots (see section 5.6). The complete set of figures and raw measurements for all runs are available (see section 5.7). Measurements for RLat and ELat vary significantly for different FaaS platforms. Thus, we use different ranges for the latency y-axis in the client-view plots for each FaaS platform provider. All reported numbers are based on requests that are started before 270 seconds. This is due to a configured client-side timeout of 30 seconds and no more workload generation after 300 seconds. Thus, requests started after 270 seconds are potentially partially executed in a changed environment.

Table 4 shows aggregated statistics for each platform aggregated over all workloads WL0-WL4.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{Results for WL2 and WL4 on AWS}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{Results for WL2 on GCF}
\end{figure}

6.1.1 AWS Lambda. For all workloads, we measure 844 failed requests out of a total number of 88452 requests that indicate a failure rate of 0.954\%. We measure a 99-percentile RLat of 2.051s that significantly increases for the worst-case (to 17.082s). Because RLat measurements take network latency between client and SUT into account, we argue that the 99-percentile is significant. We measure similar values for median ELat of 1.635s and 90-percentile ELat 1.679s with a worst-case of 2.393s.

We continue to analyze results for the individual workloads WL2 and WL4. Both workloads show no failed request before the 240th second. During the scaling phase, RLat varies between (i) 1.462s and 2.400s with a median of 1.716s for WL2 and (ii) 1.560s and 2.410s with a median of 1.721s for WL4. Values for median RLat and ELat between a one-second interval differ within a small band. After sending the first requests, the median RLat und ELat increase for under 10 seconds by less than 0.5s. We observed the same effect for the first 10 seconds of (i) the scaling phase of WL0-WL2 and (ii) the warmup phase of WL3-4. Our results indicate that AWS can handle sudden trps increases of different sizes equally well.

AWS does not start new hosts after the end of the scaling phase. The start of new hosts correlates with increasing trps. Visual inspection of the platform-view plot for WL4 (see figure 2) reveals that the platform successfully starts executions on 55 different hosts within the first second of the warmup phase.

Our results indicate that a single host rarely executes events in parallel. Due to our measurement approach, our results could even indicate that a single host executes events sequentially. Our results do not indicate that executions cause noisy neighbor effects.

6.1.2 Google Cloud Functions. Across all workloads, we measure 2324 failed requests out of a total number of 83698 requests, which indicates a failure rate of 2.777\%. ELat varies between the median of 1.391s and a maximum of 2.616s in a range of 1.224s. In contrast, RLat varies between the median and 90-percentile by 2.836s and median and 99-percentile by 19.815s. Thus, our results indicate that the execution time and execution cost for executions are stable, but the time between starting a request and beginning an execution can vary significantly.
We conduct an indepth analysis for WL0 and WL2 (see figure 3). In response to increasing trps, the median and maximum RLat temporarily increase significantly. For WL4, we observe increases in RLat up to 30 seconds, which results in client-side timeouts. Our results indicate that temporary increases in RLat correlate with temporary trps increases. We continue to analyze this behavior based on the visual inspection of the corresponding platform-view plots (see figure 3). The GCF platform continues starting new hosts after the scaling phase up to \( t = 240 \)s, around 2 minutes into the cooldown phase. These late starts indicate that GCF initially underestimates the target workloads. Moreover, we observe HHeat < 2. Thus, our results indicate that multiple executions run on the same host at the same time. However, the combination of high HHeat and stable ELat indicates the isolation of different executions on the same host.

The combined visual inspection of the client-view and platform-view plots for WL2 shows that RLat increases occur with high HHeat values on multiples hosts. Around \( t = 170 \)s HHeat and RLat suddenly drops on multiple hosts accompanied by a sudden drop in maximum RLat. Our results indicate that the platform queues incoming events and only a well-defined number of events execute on available hosts; thus, preventing volatile execution latency.

### Table 4: Comparison of FaaS Platforms Based on Client-side Qualities Aggregated Over Workloads WL0-WL4.

<table>
<thead>
<tr>
<th></th>
<th>Reliability</th>
<th>RLat [s]</th>
<th>ELat [s]</th>
<th>ECost [$]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Req [#]</td>
<td>Failed [#]</td>
<td>Failed [%]</td>
<td>Med</td>
</tr>
<tr>
<td>AWS</td>
<td>88452</td>
<td>844</td>
<td>0.954</td>
<td>1.711</td>
</tr>
<tr>
<td>MAF</td>
<td>86657</td>
<td>83170</td>
<td>95.976</td>
<td>20.002</td>
</tr>
</tbody>
</table>

### 6.1.3 IBM Cloud Functions. We measure 2457 failed requests out of a total number of 88534 requests, which indicates a failure rate of 2.775%. We measure a median RLat of 2.775s and a median ELat of 3.193s. RLat and ELat significantly increase for high percentiles up to 29.898s and 14.943s, respectively. Our results indicate that clients observe significant variations in performance and execution costs.

We continue to discuss WL2 and WL4 in more detail (see figure 4). Except for WL3, we do not observe any failures during experiment runs. The failures in WL3 occur after 1 minute into the cooldown phase. The failure rate RFailure is around trps, thus resulting in white vertical bars in the corresponding platform-view plot.

We observe quite stable values for median RLat and ELat for different one-second intervals in the same run. We observe a difference of around 2 seconds in median ELat and around 6 seconds in median RLat for cold workloads WL0-WL2 and warm workloads WL3-WL5. Furthermore, our results show significant variations in maximum ELat over 2 seconds and RLat over 6 seconds throughout complete runs. The variations in maximum ELat in combination with high values for HHeat indicate noisy neighbor effects caused by different events of the same function. The ICF platform does not start new hosts after the end of the scaling period. Thus, our results indicate that the platform underestimates the initial target workload.

### 6.1.4 Microsoft Azure Functions. We measure 83170 failed requests out of a total number of 86657 requests that indicate a failure rate of 95.976%. Over 80% of these failures are due to the client-side timeout of 30 seconds; the other failures were all related to service errors (HTTP 500). These failure rates are highly significant. Because of these high failure rates, we do not show any platform-view plots. We want to stress that RLat and ELat statistics only represent less than 5% of the total number of started requests. Thus, they are less representative compared to the previous three platforms. However, we observe a short median ELat of 0.729s. From a client-side perspective, we observe a median RLat of 20.002s. Visual inspection of individual runs (see figure 5) shows that failure rates increase up to 100% for the warm workloads WL3-WL4.

Our results indicate that the FaaS platform is not able to stabilize within 300 seconds of an experiment run.

### 6.2 FaaS Platform Comparison

Table 4 shows a comparison of the four FaaS platforms. Our results indicate low failure rates under 2.777% for three FaaS platforms with similar values for GCF and ICF and the lowest failure rate of 0.954% for AWS. We measure the highest failure rates of 95.976%
We measure similar ELats for AWS and GCF, indicating stable execution times and thus stable execution costs. Our results indicate dense multiplexing of different executions on single hosts for GCF under performance isolation. In contrast, AWS seems to rely on a larger number of different hosts and sequential executions. Similar to GCF, ICF places multiple executions on single hosts. However, higher variations in ELat measurements indicate less performance isolation between parallel executions compared to GCF.

Based on our results, GCF RLat is significantly sensitive to steep increases in target workloads on cold functions.

In general, our results support the claim that FaaS platforms deliver on the promise of automatically provisioning the compute resources required to execute highly volatile rates of incoming events. Execution costs of different executions with the same complexity can vary for the same FaaS platform. All FaaS platforms show sensitivities to different target workloads. Thus, depending on application objectives, there is a need to mitigate some of the platform behaviors to achieve stable and reliable client-side qualities.

6.3 Limitations

Our experiments are subject to several limitations. An experiment must make assumptions about a concrete handler implementation, handler inputs, and target workloads. However, these assumptions might not represent a specific application accurately. We address this limitation by fostering easy extensibility through a detailed description of simple experiment design and making all experiment artifacts and results permanently available.

We use Artillery as a workload generator. However, Artillery does not support an easy distributed deployment. Thus, the scalability of the target workload is limited. While the scalability was sufficient to generate our workloads, it limits extensibility. Furthermore, we found that the used version of Artillery does not provide natural capabilities for tracking client-side failures, thus impeding verification of experiment results.

We did not deploy our workload generator within the same datacenter as the FaaS platforms. While this choice improves the identification of client-side errors, it adds the latency of the network to RLat and DLat. We try to mitigate this limitation by reporting multiple high-level percentiles for latency measurements.

Compared to the other three FaaS platforms, MAF does not expose the same capabilities for configuration of memory size. Thus, MAF should be compared to other FaaS platforms carefully. Moreover, MAF does not expose the amount of memory that is used by individual executions. Thus, we could only approximation the execution costs for MAF using ELat.

The short runtime of 300s for all workloads might not seem long enough for all platforms to stabilize. A longer duration could favor specific platforms more; thus, it could be considered an unfair comparison. However, we investigate FaaS platforms for the objective-driven design of serverless applications. Thus, we argue that FaaS platforms should be able to react to workload changes within 240 seconds.

The platform-view plot requires sufficient data to represent a run accurately. Thus, its expressiveness is limited for runs with high failure rates.

7 IMPLICATIONS FOR APPLICATION DESIGN

In this section, we present the implications of our experiment results for application developers looking to build elastic serverless applications. For this purpose, we introduce three possible application scenarios and derive appropriate application objectives. Based on the results presented in the previous section, we discuss the applicability of different FaaS platforms for our three application scenarios.

7.1 Application Scenarios

We propose three application scenarios and determine suitable latency, reliability, and execution cost targets. Each of the scenarios represents a current or up-and-coming use-case for serverless computing.

7.1.1 Scenario 1 - Web Serving. The first application scenario is online web serving, perhaps the most common type of application currently deployed in FaaS platforms. Typically, web applications need to provide quick feedback to users, and the number of concurrent clients interacting with the app can fluctuate substantially.

Web applications are characterized by strict low-latency requirements. Research has shown that even small variances in response latency (>1 second) are noticeable to users, which leads to less interaction with the app [1] and can potentially impact its monetization. Besides low-latency, web serving also requires high reliability. Generally, it is recommended that the error rate stay around 1%. Lastly, execution costs should always be considered but do not play a significant role in this scenario, since web functions tend to have very low execution times.
7.1.2 Scenario 2 - Exploratory Data Analysis. The second type of application is serverless exploratory analytics, which has seen increased adoption and support by research [10] and industry [24] alike. Business analysts will often perform initial investigations on data to discover patterns or test hypotheses. Due to its infrequent and sporadic nature, exploratory analytics is also prone to volatile workloads, with many executions being triggered all at once.

This scenario has more relaxed requirements for response latency because analysts do not need instant feedback from the application. However, latency should still not surpass 20 seconds to ensure smooth coordination between functions and promote the productivity of the analyst. Because functions can be re-executed in case of an error, the reliability requirements are also not as strict. Still, dealing with errors can lead to coordination efforts and introduce more deployment costs. Finally, exploratory data analysis use-cases are more sensitive to execution cost than web applications because of their long function execution times.

7.1.3 Scenario 3 - Periodical Batch Processing. Similar to the second application type, this scenario also makes use of the serverless ecosystem for big data processing. In this scenario, jobs are executed periodically and are triggered automatically. The execution of batch processing jobs, e.g., [28], can also be characterized as a volatile workload.

The lack of user interaction lowers the latency requirements in this scenario. Latency and reliability should still be taken into account because they influence the coordination of functions and therefore cause more initial deployment costs. Like in the previous scenario, batch processing applications are also quite sensitive regarding execution costs.

7.2 Discussion of FaaS Platform Applicability

As described in section 6, we observed a failure rate of over 95% for MAF during our experiments, which indicates that the platform is not able to scale under volatile workloads. We, therefore, deem it unsuitable for all three of our application scenarios and urge interested developers to continue to monitor its performance until the platform matures. The following sections discuss the applicability of AWS, GCF, and ICF for our three application scenarios.

7.2.1 FaaS platforms for web serving. Scenario 1 assumes high reliability as well as very low and consistent latency as application objectives. All three of our remaining candidates (AWS, GCF, and ICF) performed reliably, with failure rates of under 2.77%. Of the three, AWS proved to be the most reliable with a failure rate of only 0.95%. With regard to latency, not all three candidates were able to meet our application objective. ICF’s median response latency of over 7.08 seconds is less than ideal, and the latency variance between median and 99-percentile shows an even dimmer outlook for potential users of the application. We, therefore, deem ICF unsuitable for serving web applications. Contrary, GCF, and AWS both show median latencies of under 1.72 seconds, with a slight advantage to GCF. However, the latency variance between the median and the 99-percentile of almost 20s reveals GCF’s inability to handle a sudden increase in requests. GCF’s mediocre performance during the scaling phase could potentially be improved through mitigation strategies to be investigated in future work. Therefore, we do not rule out GCF outright. AWS performs consistently and is consequently the most suitable candidate for a web serving scenario.

7.2.2 FaaS platforms for exploratory data analysis. Scenario 2 should manage failure rates of up to 10% and allows for slower response times of up to 20s. Besides this, execution costs should be minimized as far as possible.

Based on our results, all three candidates performed very reliably, with failure rates well below our 10% threshold. Regarding latency, we can see a good median performance by all three candidates too. Looking at slow requests, represented by the RLat 99-percentile, AWS clearly stands out as the fastest, while ICF’s performance of 15.33s still meets our objective. GCF falls short of our 20s target by 1.32s, which again reveals its inability to scale when many executions are triggered in a short amount of time. Still, as previously mentioned, this behavior could be tolerated or potentially improved through mitigation strategies.

With all three candidates still under consideration, we look at execution cost as a potential decider. The execution cost of each function depends on its execution latency (see also table 3). Throughout our experiments, the complexity of the execution is assumed to stay constant, so variances in ELat indicate noisy neighbors and insufficient provision of resources. Our results show low mean ELat for both GCF and AWS. ICF, meanwhile, underestimated the workload and provisioned too little resources, significantly impacting the execution time and, therefore, also the execution cost. AWS and GCF both show rather consistent ELat, but GCF outperforms AWS with a better mean execution latency, making it the cheapest out of the three candidates.

To summarize, for exploratory data analysis, we recommend AWS for application providers that value low latency, while GCF is the better option for those who want to optimize execution costs.

7.2.3 FaaS platforms for periodical batch processing. Scenario 3 is similar to the previous scenario, with relaxed reliability requirements and the goal of minimizing execution costs. However, latency does not play an important role here.

Correspondingly, neither reliability or latency outright disqualify any of the three candidates, so cost becomes the tiebreaker between the three. Execution costs are marginally lower for GCF than AWS. Still, as mentioned in section 7.1.3, inconsistent latencies, and function failures can influence the coordination of functions and therefore cause more initial deployment costs. Thus, we recommend GCF for application providers who prioritize low execution costs and AWS for those looking for a straightforward deployment.

8 CONCLUSION

We presented a novel experiment design and corresponding toolkit for measuring the elasticity of FaaS platforms and evaluated four cloud providers: AWS, Google, IBM, and Microsoft. Our results show significant quality and cost differences between all four platforms.

Based on the results, we discussed the applicability of the four FaaS platforms in three common application scenarios. Our experiment design and associated toolkit can be used to verify, reproduce, or extend our results or run custom benchmarks.
ACKNOWLEDGMENTS

The work in this paper was performed in the context of the DITAS and BlobGPV+Blossom projects. DITAS is partially supported by the European Commission through the Horizon 2020 Research and Innovation program under contract 731945. BlobGPV+Blossom is partially funded by the Germany Federal Ministry for Economic Affairs and Energy (BMWi) under grant no. 01MD18001E. The authors assume responsibility for the content.

REFERENCES