An Evaluation of FaaS Platforms as a Foundation for Serverless Big Data Processing

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

Karim El Tal
Careem
Berlin, Germany
karim.eltal@careem.com

Stefan Tai
ISE, TU Berlin
Berlin, Germany
st@ise.tu-berlin.de

Preprint, to appear at IEEE/ACM 12th International Conference on Utility and Cloud Computing (UCC '19), December 2–5, 2019, Auckland, New Zealand; final version availible at https://doi.org/10.1145/3344341.3368796

ABSTRACT

Serverless computing, or Function-as-a-Service (FaaS), offers a new alternative to operate cloud-based applications. FaaS platforms enable developers to define their application only through a set of service functions, relieving them of infrastructure management tasks, which are executed automatically by the platform. Since its introduction, FaaS has grown to support workloads beyond the lightweight use-cases it was originally intended for, and now serves as a viable paradigm for big data processing. However, several questions regarding FaaS platform quality are still unanswered. Specifically, the impact of automatic infrastructure management on serverless big data applications remains unexplored.

In this paper, we propose a novel evaluation method (SIEM) to understand the impact of these tasks. For this purpose, we introduce new metrics to quantify quality in different big data application scenarios. We show an application of SIEM by evaluating the four major FaaS providers, and contribute results and new insights for FaaS-based big data processing.

KEYWORDS

Serverless; Benchmarking; Big Data Processing; Cloud Computing

1 INTRODUCTION

Traditionally, servers have always played a central role in cloud-based big data processing. Typically, big data is processed in large clusters: jobs consisting of individual tasks are executed across a set of machines, in stream or batch processing models. New advances in the field have sufficiently addressed the performance and scalability needs of such deployments, e.g. through distributed computing frameworks[1][2][3][4]. Still, the costs associated with deploying, configuring, and maintaining these systems continue to be significant, as does the knowledge required to manage such a cluster.

Research has recently explored new models and alternative architectures to address these shortcomings. One approach that has gained traction in both industry and the scientific community is serverless big data processing, based on Function-as-a-Service (FaaS) platforms. FaaS platforms let developers run code to be executed and scaled as needed, without provisioning or managing servers.

Even though it was originally intended for lightweight use-cases like API-serving or small backend services [20], FaaS has since proven to be a viable paradigm for heavier workloads. Among them, FaaS is being used for big data processing [7, 10, 30] and information retrieval [6] workloads. The potential of FaaS for big data processing was also recognized by major cloud providers including AWS [5], and the first FaaS-based Big Data Computing Frameworks are starting to emerge [4, 24]. In the context of big data, FaaS promises faster startup times, lower cost and individual task based scaling, and thus can be considered a new relevant tool in the big data processing toolbox.

To support FaaS-based big data processing, some open challenges [11] are already being addressed. Specialized storage solutions have been developed to allow for intermediate ephemeral storage between functions [14, 26]. Approaches to support function orchestration are also beginning to emerge: [22], AWS Step Functions[6], and Fission Workflows[7]. However, there are still several open questions regarding FaaS platform quality [29]. Making knowledgeable design decisions in serverless big data processing requires detailed understanding of these platforms.

In particular, it remains unclear how common operational tasks usually executed by computing frameworks impact application qualities, e.g., scalability, consistency, and cost-effectiveness. These qualities are relevant for common big data application scenarios, including ad-hoc or explorative analytics. In explorative analytics, queries can change constantly and the complexity of these queries is not known beforehand. Consequently, this can lead to the execution of many operational tasks: the code in the deployment package may change often, and the unpredictable computational requirements may lead to frequent changes in function configuration.

[1] http://spark.apache.org
[2] http://hadoop.apache.org
[3] https://flink.apache.org
[4] http://www.tensorflow.org
[5] https://github.com/awslabs/lambda-refarch-mapreduce
[6] https://aws.amazon.com/step-functions/
[7] https://github.com/fission/fission-workflows

Here, we address the following research question: **Do operational tasks impact quality in FaaS platforms and is the impact relevant for serverless big data processing applications?**

To answer this, we build on our extensive expertise and previous work on quality-driven design and evaluation of cloud-based systems [2, 3, 15, 16]. We present serverless big data application scenarios in the context of smart grid management and show the importance of operational task quality. To measure the quality impact, we propose a novel evaluation method (SIEM) and introduce new metrics. Overall, we answer the research question by means of experimental evaluation. From the results, we extract generic insights for the design of future serverless big data applications.

The remainder of this paper is organized as follows: The necessary background on FaaS is laid out in section 2. Section 3 refines our initial problem statement. Following that, we discuss relevant related work in section 4. The evaluation method and the experiment design are introduced in section 5 and section 6, respectively. Results are presented in section 7. A subsequent discussion in section 8 synthesizes generic insights and considers limitations. Finally, we close our paper with a conclusion and future work (section 9).

2 BACKGROUND

FaaS platforms are systems that facilitate general serverless computing. These platforms enable a developer to define a computation only by the code that is to be executed and the events that trigger the execution[11]. The developer is relieved from resource management tasks like provisioning, scaling and logging, which are now handled by the FaaS platform.

A multitude of FaaS platforms exist already, some are commercial like AWS Lambda[8], Google Cloud Functions[9], and some are available as open-source platforms, e.g., OpenWhisk[10], KNative[11], and Fission[12]. All these platforms require a developer to specify a *deployment package*. A deployment package contains the code and all libraries that are needed to run that function. The underlying runtime, e.g., a python environment, is provided by the platform. A developer also has to specify the *function configuration*. The function configuration parameterizes trigger events as well as runtime settings. Runtime settings can, for instance, specify the memory available to the function or the timeout before a function is forcefully terminated. Both function configuration and deployment package are controlled through *operational tasks*. In general, a operational tasks is an infrastructure management task that is executed automatically by a cloud infrastructure provider. Operational Taskss are prompted by changes performed by a user, and executed by the platform on their behalf. The FaaS platform will create a *service function* when an event triggers a run. Depending on the platform, that deployment can either be a container or a lightweight VM. The developer-defined code will run once a service function is available. Most platforms reuse service functions in order to reduce the initialization time for subsequent events. Furthermore, most platforms remove a service function after a fixed period of time, if no new events occur.

[8] https://aws.amazon.com/lambda/
[9] https://cloud.google.com/functions/
[10] https://openwhisk.apache.org/
[11] https://cloud.google.com/knative/
[12] https://fission.io/

3 PROBLEM STATEMENT

In order to refine our initial problem statement, we present two application scenarios of serverless big data processing. In these scenarios, the need for frequent execution of operational tasks is highlighted. Finally, we show the importance of operational task quality for making knowledgeable design decisions in serverless big data processing.

3.1 Application Scenarios

We use two application scenarios in the context of smart grid management, smart grid management is center to many of our past and ongoing research projects[13][14]. Each scenario makes use of serverless big data processing. The first application [30] periodically executes offline analysis using matrix multiplication. We refer to the first application as `Batch`. The second application [15] conducts online analysis for smart grid management based on smart grid metering data. We refer to the second application as `Streaming`. The main quality objectives of `Batch` comprise correctness and fault-tolerance of analyses. In contrast, the main quality objectives of `Streaming` are high availability and low latency. Both require good cost-efficiency and high scalability with the size input matrices or amount of metering data per time period, respectively.

3.2 Operational Tasks

We identify cases that require to execute operational tasks based on the two example applications (see section 3.1).

We do not assume completeness for the list of cases. However, we argue that the depicted cases are highly relevant because they occur in many applications that utilize big data processing. For `Batch` ...

- **C1 - Job Deployment**: First approaches to port existing big data processing frameworks to FaaS platforms result in large deployment packages that suffer from significant cold start times [25], rendering these approaches effectively unusable for many application scenarios. To bypass this, new native frameworks can change the deployment package of an existing function with minimal code required for a specific job.
- **C2 - Job Recovery**: During the execution of a job, a single processing task can continuously fail if a service function exceeds available memory or the maximum execution time defined by the cloud provider. To recover from this fault, a compute framework can change the function configuration to increase its allocated compute resources. The task can then be re-executed, avoiding the redeployment of the complete job.
- **C3 - Objective Tuning**: Fully managed data analysis services allow to customize cost and performance objectives of jobs that are periodically executed [30]. This customization commonly results in sizing the compute resources of service functions accordingly, changing the function configuration.

For `Streaming` ...

[13] https://www.ise.tu-berlin.de/PEN/parameter/en/
[14] https://www.ise.tu-berlin.de/BloGPV/parameter/en/

- **C4 - Rollback**: A rollback might be required after delivering a faulty version of a service function. A rollback results in changing the deployment package of a service function back to an earlier version.
- **C5 - Changing Service Objectives**: Similar to the case objective tuning, changing service objectives can require changes to the configuration of an execution environment of a service function.

We can synthesize all five cases C1-C5 into two relevant operational tasks OP1-OP2 that are executed in FaaS platforms.

- **OP1 - Function Configuration Change**: A *function configuration change* modifies the configuration of the execution environment of a service function from an old configuration c to a new configuration c'. Examples of configuration changes are limiting access rights or increasing the amount of computing resources for a service function during execution. A configuration change does not alter the deployment package of a service function.
- **OP2 - Deployment Package Change**: A *deployment package change* replaces the deployment package of a service function from an old package p to a new package p'. For example, a developer commits a new version of a function handler implementation. A deployment package change does not alter the configuration of the deployment environment of a service function.

We argue that operational tasks are so commonly executed in serverless big data processing applications and that developers must explicitly account for them.

3.3 Quality of Operational Task

Function configuration changes and deployment package changes are commonly executed in FaaS-based applications. Thus, it is highly relevant to characterize the quality of executing operational tasks, and other underlying behavior in different FaaS platforms. To illustrate this point, we discuss the cases C1-C5 in section 3.2 in light of the different service objectives for the two application scenarios 3.1.

We assume that the execution of an operational task is not instant but requires some finite time before the change becomes visible to a client of a service function. Thus, depending on the internals of a FaaS platform, the same service function might be offered with different environments (OP1) or functionality (OP2) at the same time. For example, a long running job uses a service function that switches to a different function handler at an unknown point in time and, thus, compromises correctness and verifiability of the job result.

In conclusion, we define our problem statement as follows: **Does a relevant quality impact exist for executing code or configuration changes in fully managed FaaS platforms?**

4 RELATED WORK

This paper addresses the experiment-driven evaluation of FaaS platforms. We discuss related work from the perspective of benchmarking cloud platforms and benchmarking FaaS platforms in general as well as work on serverless big data processing frameworks.

Design objectives of system benchmarks is a relevant and ongoing problem in research [1, 2, 5]. 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, 8, 28]. These properties must also apply to the evaluation proposed in this paper.

Furthermore, cloud service benchmarking research [2, 3, 15] has many parallels to work presented in this paper. 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.

Industry and research already started to use experiment-based evaluations to analyze FaaS platforms [9, 18, 21, 23, 25]. Most of these efforts focus on performance [17]. However, the evaluation of operational tasks in FaaS platforms has largely been ignored in literature. A first proposal by Lee et al. [19] evaluates the performance impact during function code changes. The proposal motivates our work and we significantly extend its scope. We were unable to verify or reproduce the original experiment due to missing information [17]. Moreover, Lee et al. do not discuss the impact of operational tasks on `Streaming` or `Batch` serverless big data applications specifically.

A second line of potential related work are evaluations of new systems in support of serverless big data processing: [4, 10, 12, 30]. However, the proposals do not specifically address evaluations of operational tasks.

Towards that end, we propose an evaluation method in the next section. This method allows serverless data processing frameworks to evaluate sensitivities of operational tasks.

5 EVALUATION METHOD

We approach our research question with an experiment-driven evaluation of FaaS-platforms (see sections 6 and 8). For this purpose we introduce the Serverless Infrastructure Evaluation Method (SIEM). SIEM builds on our extensive previous work on quality-driven design and evaluation of cloud-based systems [3, 13, 15, 16] and the structured review of community-driven evaluations [17]. We describe goals, roles, and tasks of SIEM.

5.1 Goals

Experiments should be reproducible, fair, portable, understandable, and relevant [3]. We use SIEM in support of reproducablity and understandability. Furthermore, we identified three additional design goals.

- **Coordination**: Experimentation is difficult, expensive, time-consuming, and error-prone. Thus, experiments are typically not conducted by a single person, but by a group of people that must coordinate their efforts. Therefore, a method should reduce coordination efforts. For example, our experiments are conducted by a group of students, research associates, application providers, and cloud providers that participate differently.
- **Extensibility**: The scope of specific experiments is always limited. Thus, experiments often motivate additional experiments with different configurations for the environment,

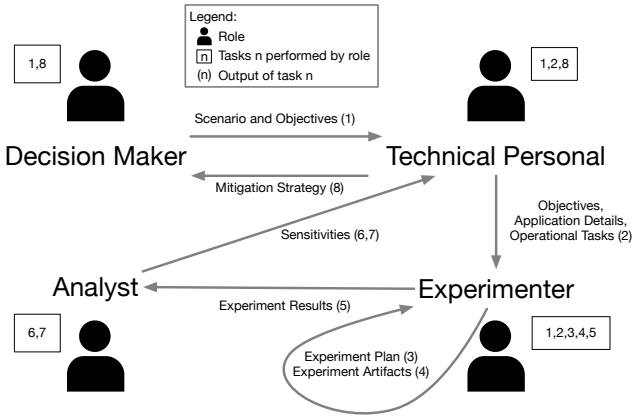

Figure 1: Overview on Roles, Tasks, and Outputs of the Serverless Infrastructure Evaluation Method (SIEM) inspired by ATAM and ETEM

workload, or observations. A method should help to identify extension points while fostering resuse.

- **Organizational Context**: It is often challenging to identify and implement need for experiment-driven evaluations in a more general organizational context. A method should help to overcome this challenges.

To address all design goals, we present SIEM. SIEM extends ETEM a generalized method for evaluating cloud-based applications [13]. While promising, we found that ETEM does not explicitly account for roles, steps, and artifacts that occur in FaaS platform benchmarking that we could identify during our review of existing FaaS platform benchmarks [17].

5.2 Roles

SIEM distinguishes between four roles (see figure 1).

- The *Decision Maker* (DM) is a management role that provides high-level information about the application scenario and objectives. The DM agreees on mitigation strategies that the TP and Analyst design or negotiate.
- The *Technical Personal* (TP) is a system architect, software engineer, or system operator. The TP provides detailed information on application details, e.g., architecture, serverless infrastructure, workloads, operational tasks. In addition, the TP provides mitigation strategies for operational tasks with undesirable impacts on objectives by design or negotiation with a cloud provider.
- The *Experimenter* designs an experiment plan, executes experiments, and checks validity of raw experiment results.
- The *Analyst* aggregates raw experiment results, identifies sensitivities of operational tasks, and assesses the need for mitigation strategies to handle operational tasks that negatively impact objectives.

5.3 Steps

Our method includes 8 tasks that are typically executed in sequence, see also section 6. The method allows for feedback loops and iteration between different tasks, thus, it is possible to jump back to prior tasks and start again.

Step I: Define Scenarios and Objectives: During Step I, the DM and TP specify an application scenario. This includes high-level information on the application architecture including software services, workloads, and application objectives. Examples of application objectives are targets for the number of requests in a time window that fail or exceed a request request-response latency.

Step II: Collect Application Details: The TP identifies select serverless infrastructure services, e.g., FaaS platforms, storage services, and gateway services, and corresponding operational tasks. The experimenter shares workloads and documentations for infrastructure services with the experimenter.

Step III: Create Experiment Plan: The *Experimenter* identifies experiments of interest. For each experiment, he creates an experiment plan that contains a detailed description of the corresponding experiment design. To ensure reproducability and verifiability of results, the experiment plan describes at least the system under test (SUT), workload, observations, derived metrics, and controlled variables.

Step IV: Collect Experiment Artifacts: The *Experimenter* collects all required experiment artifacts, e.g., workload generator. To reduce time and complexity and improve reproducibility of results, the *Experimenter* should make use of experiment automation [2, 13].

Step V: Conduct Experiments: The *Experimenter* runs experiments based on the experiment plan. He checks the internal and external validity of the obtained results. Experiments with low validity should result in multiple experiment executions or iterative refinement of the experiment plan and experiment artifacts. Valid results are shared with the Analyst.

Step VI: Identify Sensitivities: The *Analyst* checks the validity of results. Then, he analyzes how sensitive objectives are to operational tasks of different infrastructure services. Appropriate methods include visual inspection and the definition of select metrics.

Step VII: Assess Sensitivities: The *Analyst* characterizes and evaluates trade-offs [13] and identifies operational tasks that require mitigation strategies. Mitigation strategies can become necessary to handle undesirable impacts on objectives. For example, the analyst decides that a strategy is necessary to mitigate high numbers of failing requests for five minutes after triggering a code change. The *Analyst* shares the assessment with the TP.

Step VIII: Design or Negotiate Mitigation Strategy: In Step VIII, the TP and the *Analyst* identify and implement desired mitigation strategies. Mitigation strategies are (i) changes to the application design by the TP, (ii) migration to another cloud provider or (iii) negotiating a solution with the cloud provider. Additional iterations of SIEM, should evaluate the implemented mitigation strategy. The DM approves select mitigation strategies proposed by the Analyst and TP.

In the problem statement (see section 3) we defined scenarios and objectives analog to Step I of SIEM, Step II-VIII follow in the next sections.

6 EXPERIMENT DESIGN

In this section we present the experiment design based on the problem statement (see section 3). Specifically, we present Step II, III and IV of SIEM (see section 5).

6.1 Application Details, Objectives, and Operational Tasks

We summarize the initial input that is available to an experimenter for creating an experiment plan. The experiment design is based on two applications `Batch` and `Streaming` (see section 3.1). `Batch` priorities the application objectives correctness, fault-tolerance, and cost-efficiency. `Streaming` priorities the application objectives high availability, low latency, respectively. `Batch` changes deployment packages (OP2) of a service function after submission of a new job (C1). Moreover, it changes the function configuration (OP1) during job execution to increase resources to recover single failing tasks from OOM or timeout exception (C2). `Streaming` alters deployment packages (OP2) due to continuous delivery of new features (C1) and due to rollbacks after fault detection (C4). The experiment plan must allow to investigate the impact of operational tasks on application objectives.

6.2 Experiment Plan

As specified in SIEM, the experiment plan consists of a description of the system under test,the workload,the treatments and the metrics. In the following we present these details.

6.2.1 System Under Test. We investigate four fully managed FaaS platforms: Amazon Web Services Lambda (AWS), Google Cloud Function (GCF), IBM Cloud Function (ICF), and Microsoft Azure Functions (MAF). Due to restricted availability, We provision service functions in different data centers in different regions: Frankfurt for AWS and IBM, Belgium for Google, and the Netherlands for Azure.

We configure the execution environment of each service function as follows. All four FaaS platforms use a Linux-based execution environments. In addition, MAF uses a Windows-based execution environment, respectively. The memory size is 512 MB. The timeout is 300 seconds. The trigger of each service functions is a HTTP event trigger.

All function handlers target the compute platform Node.js version 8, as it is available among all investigated FaaS platforms. We want to ensure are verifiable and reproducible usage of compute resources for each execution that approximates our applications. Thus, a function handler calculates Fibonacci numbers recursively. A function handler must not make use of caching.

6.2.2 Workload. We use a closed-loop workload model [27] that emulates 10 client machines. To isolate effects from operational tasks, we use a constant load profile associated with each requests. Precisely, we return the 35th Fibonacci number for each request.

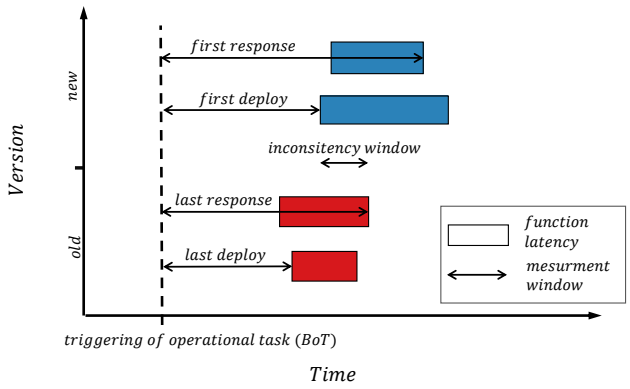

Figure 2: Representation of relevant measurement points.

6.2.3 Treatments. A treatment is a controlled interaction with an experiment environment. We use two treatments in support of our experiments.

- **T1**: Treatment 1 reduces the memory available to an execution environment from 512 MB to 128 MB and the timeout for an execution from 300 seconds to 30 seconds.
- **T2**: Treatment 2 updates a static value in the function handler source code of the deployment package.

Therefore, a treatment changes all instances of a service function from an old version to a new version.

6.2.4 Experiment Process. The experiment process consists of uploading the first deployment package followed by the start of the workload generator. After a warm-up phase of 30 seconds, we apply T1 or T2 and run the workload generator for another 120 seconds. We identified both durations with preliminary experiments.

6.2.5 Metrics. We select ten metrics in accordance to general requirements for quality metrics [3]. Each metric is associated with one or more application objectives. We obtained the final set of metrics after multiple iterations of Task III to Task VII. In each iteration, Analysts suggested extensions of the experiment scope to Experimenters. In the following we present the final list of metrics, figure 2 illustrates basic metrics required to derive our final metrics.

- **Beginn of Treatment** `BoT` [s]: Difference of beginning of a treatment and start of an experiment.
- **Last Deploy** `LastDpl` [s]: Difference of latest start of an execution in an old instance and `BoT`.
- **Last Response** `LastRes` [s]: Difference of the last end of an execution in an old instance and `BoT`.
- **First Deploy** `FirstDpl` [s]: Difference of the earliest start of an execution in a new instance and `BoT`.
- **First Response** FirstRes [s]: Difference of the last end of an execution in a new instance and `BoT`.
- **Stabilization Time** `StabTime` [s]: Difference of the point in time when the variance of execution latency is lower than a target value δ and `BoT`.
- **After Time** `AT` [s]: Difference of end of an experiment and `StabTime`.

Each final metric is a member of one of three groups that each address a quality: performance, availability, and deployment consistency. We describe performance with 99 percentile, 90 percentile, and maximum latency and total throughput for both `StabTime` and after time.

For availability, we report the total number of execution failures after BoT. We approximate Mean Time To Repair (MTTR) with the earliest time to detect an error (FirstRes) plus the time it takes to enact a change to a stable version (`StabTime`).

To describe deployment consistency, we define the metric *inconsistency window* `IW` [s]. `IW` describes the difference between LastRes and FirstDpl (see figure 2). We argue that `IW` is elegant because a positive value describes the time for which two versions are running in parallel. A negative `IW` value indicates the time of unavailability in response to a treatment.

6.2.6 Measurements. Most of the aforementioned metrics can be measured through the client-side workload generator. The generator tracks the time when a function is triggered and when it responds. Furthermore, the response body or a potential error message is recorded for each invocation. Each function response also contain measurements that allow us to determine the version of that function and the used function configuration. From these measurements, we can calculate all mentioned metrics.

6.3 Experiment Artifacts

The experiment artifacts and experiment results are available on github.com[15]. We utilize command-line tools for each provider to provision the experiments and apply the treatments T1 and T2. We furthermore, use Apache JMeter[16] as a workload generator, which we also control through a command-line interface. Based on this, we can use one bash script per provider to perform the benchmark.

One key metric that we need to detect for all experiment is the effect of the applied treatment, e.g., a code or configuration change.

In order to detect a code change, we added a static version name to each deployment package. This version name is returned on every request and allows the detection of code changes.

In order to detect function configuration changes, however, we had to use a different strategy for each FaaS-provider. For instance, AWS has an API that allows any function to read the current function configuration while GCF stores that information in undocumented environment variables. For ICF and MAF we used operating system information obtained to the runtime API and Linux file system.

We also generated function identifies and VM identifiers for each invocation to track the reuse of VMs and service functions, which we use for better root cause analysis.

7 EXPERIMENT RESULTS

In this section we present the results of our experiments. Specifically, we present Steps VI and VII of SIEM (see section 5). Accordingly, we report the quality impact of operational tasks on performance, availability, and consistency. Table 1 and table 2 summarize our results. Figure 3 shows detailed plots for deployment

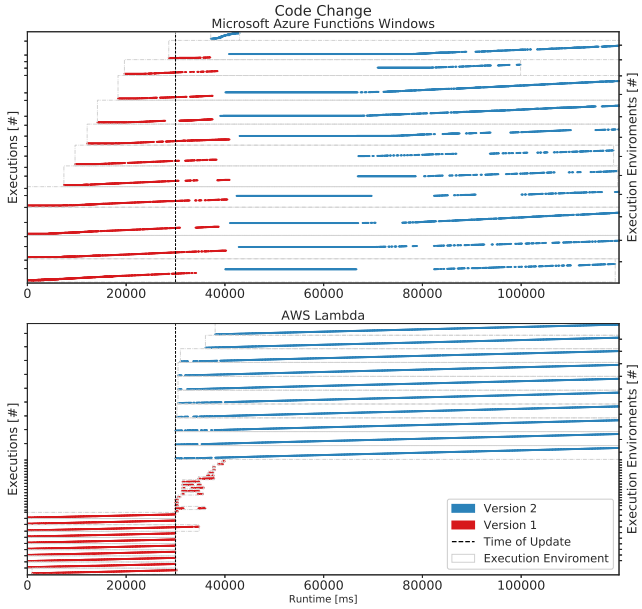

Figure 3: Deployment Package change Results for MAF and AWS. For a detailed explanation see Figure 2 and Section 7

package changes. Figure 4 shows detailed plots for function configuration, respectively. All measurements, metrics, and high-resolution figures are available in our repository[17].

Overall, we see that both deployment package changes and function configuration changes for the same provider have a similar behavior. Our results show no failed requests and therefore, we see no impact on availability. Nevertheless, our experiments show differences for performance, consistency, and change-duration in each provider.

We describe the behavior of each platform in detail.

AWS. Our experiments show that AWS has a moderate `IW` of about 10 seconds. During that change, AWS has the smallest performance impact, both for code and configuration changes. AWS also starts to apply new changes quickly, but it takes time before the change is applied across the board.

GCF. Google Cloud Functions show one of the smallest `IW` and only a moderate performance impact. However, the start and duration of a change in GCF are slow, with a delay of at least 30 seconds, see Figure 4.

ICF. ICF has the smallest inconsistency window and also the smallest duration for applying a change. Additionally, we can see that ICF has only moderate performance impacts that only occur within the first seconds of a change.

MAF Windows. The Windows variant of MAF has overall a modes operational tasks profile. The `IW` is comparable to AWS. The duration and start of a change are also comparable to the other

[15] https://github.com/eltalkarim/faas-benchmark-siem
[16] https://jmeter.apache.org
[17] https://github.com/eltalkarim/faas-benchmark-siem/results

Table 1: Metrics for deployment package changes, see section 6.2.5 for a detailed explanation of metric. The value pairs for latency and throughput show the metric before, and after the stabilization time of a deployment package change.

Quality	Metric	AWS	GCF	ICF	MAF (Win.)	MAF (Linux)
Consistency	inconsistency window [s]	10.01	0.21	0.41	3.83	85.85
	last response (v1) [s]	9.96	36.63	0.22	10.96	93.26
	first deployment (v2) [s]	-0.05	36.42	-0.19	7.13	7.41
Performance	p90 latency [ms,ms]	480, 668	1749, 340	2422, 300	7906, 576	424, 871
	p99 latency [ms,ms]	549, 775	1864, 432	4214, 416	36623, 1060	1029, 1152
	max latency [ms,ms]	550, 778	1877, 450	4318, 806	37340, 10766	6224, 1512
	throughput [#/s,#/s]	16, 18	10, 29	17, 44	3, 36	12, 30
Availablity	MTTR [s]	10.00	73.18	0.70	17.96	103.78
	error detection [s]	0.50	36.87	0.96	7.96	10.82
	failed requests [#]	0	0	0	0	0

Table 2: Metrics for function configurationchanges, see section 6.2.5 for a detailed explanation of each metric. The experiment could not be performed for Azure because MAF manages memory allocation automatically. The value pairs for latency and throughput show the metric before, and after the stabilization time of a function configuration change.

Quality	Metric	AWS	GCF	ICF
Consistency	inconsistency window [s]	6.28	0.25	0.23
	last response (v1) [s]	6.25	38.36	0.07
	first deployment (v2) [s]	-0.04	38.11	-0.16
Performance	p90 latency [ms,ms]	1484, 1424	1591, 2147	3074, 457
	p99 latency [ms,ms]	1494, 1485	2405, 2932	4028, 483
	max latency [ms,ms]	1495, 1492	2498, 3223	4360, 485
	throughput [#/s,#/s]	8, 5	8, 7	8, 30
Availablity	MTTR [s]	7.31	77.43	0.46
	error detection [s]	1.36	39.36	0.67
	failed requests [#]	0	0	0

provider. However, the Windows variant showed the worst performance impact during our experiments, with a ten-fold latency increase in latency.

MAF Linux. Admittedly, MAF Linux was still in a development stage during our experiments, which explains why it shows the highest change impacts, see also Figure 3. The change duration of over 100 seconds exceeded our initial experiment time. The `IW` is similarly long, even though changes started to be applied relatively quickly.

Summary. Overall, we see that ICF and GCF have the smallest `IW`. Furthermore, we see that ICF and AWS execute on changed function quickly, and even send a request that where send before a change to a new version, (see negative FirstDpl). In terms of the overall change-delay, we see that ICF and MAF Windows perform best. Lastly, we see that AWS has the least performance impact during changes, followed closely by ICF and GFC.

8 DISCUSSION

In this section we present the implications of our experiment results on the application scenarios, fulfilling step VII and VIII of SIEM. Following that we briefly discuss SIEM and shortcomings in the experiment.

8.1 Implications for Application Scenarios

Based on our results (see section 7), we discuss the applicability of different FaaS platforms (see section 6.2.1) for the cases C1-C5 (see section 3.2) and our two application scenarios (see section 3.1).

Case C1 assumes an exploratory online analysis of a data set with frequently changing jobs. Thus, it requires a FaaS platform that completes deployment package changes fast with high consistency to provide a good user experience to analysts. Based on our results, ICF is a highly suitable candidate. AWS and MAF (Win.) are also acceptable alternatives. However, they delay job starts by 10 seconds and may introduce inconsistencies. We consider the delay of GCF and MAF (Linux) as unsuitable for this use case.

Case C2 assumes a running job and a compute framework that dynamically sizes the resources in the environments of failing tasks before resubmitting a task. Our results indicate that ICF provides a good fit. Due to the highly consistent changes, GCF allows for simple implementation that comes at the cost of a high delays of over 30 seconds. AWS allows for faster successful re-submissions but potentially requires a mitigation strategy for handling inconsistencies.

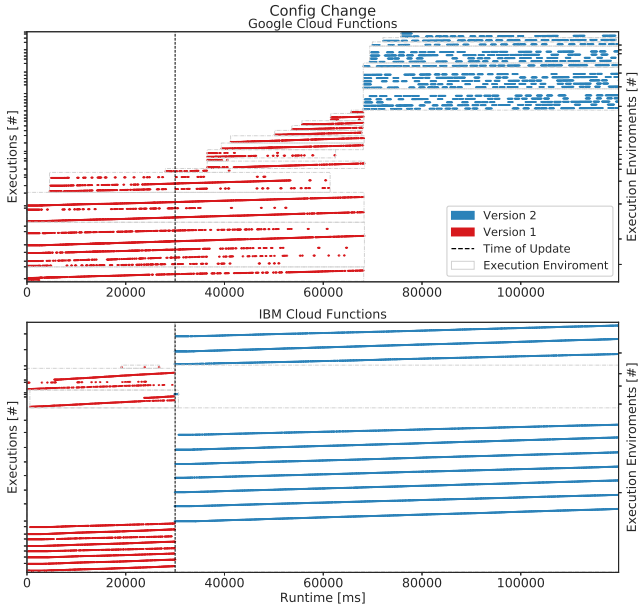

Figure 4: Function Configuration Changes in IBM and Google Cloud Functions.

Case C3 is executed offline and has no special requirements, thus, all FaaS platforms are promising candidates. If we assume that specific functions are shared between different jobs, a low performance impact becomes desirable and AWS a suitable candidate.

In the case of a rollback (C4), a fast start and completion of a deployment package change is desirable. Thus, AWS and ICF are reasonable choices. Due to their long completion time, our results indicate that GCF users should consider mitigation strategies.

Case C5 executes configuration changes to dynamically sizes computing resources of functions that realize operators in a streaming application. Low performance impacts and fast completion times are favored over low inconsistencies. Based on our results, AWS and ICF are good choices. Due to slow completion times, GCF and MAF potentially require mitigation strategies to provide the same quality. A possible solution could be additional logic for workload prediction and admission within a big data processing framework.

8.2 Experiment Limitations

Our approach is subject to a few limitations. First, the function handler used in the experiments is only a model of the real function handlers found in serverless big data processing. Observed effects might vary for function handlers with significantly different properties, e.g., very large package sizes.

Second, we model our workload as a closed-loop. We are aware that an open-loop workload model could be a more accurate representation of some serverless big data processing workloads. However, we assume that an open-loop model would only result in minor changes to performance impact measurements. Thus, the clear trends observed in our experiments would be unaffected.

8.3 SIEM

For the evaluations in this paper, we instantiate SIEM with a group of students, research associates, application providers, and cloud providers. We report on selected findings.

To interpret experiment results correctly, the Analyst role requires detailed documentation of all experiments. This requires access to detailed documentation of the experiment plan and experiment artifacts. To avoid ambiguity, each measurement point should receive an explicit label and a meaningful description. Furthermore, a standard reference point for all time-related data should be used, e.g., the start of an experiment. The same applies to the unit of time, e.g., milliseconds.

The Analyst role requested multiple small changes to the initial experiment design to conduct root cause analysis. In our experiments, we added additional measurement points to associate execution with a specific execution environment. Thus, the experiment plan, results, and analysis should be subject to careful version control. We found it very useful to associate change request from Analysts to Experimenters, e.g., requesting a new treatment or measurement point, with references to the specific sections of the experiment plan.

Our experiments showed, that the Analyst role requires deep technical expertise in experimenting, software engineering, and FaaS platforms to interpret and discuss experimental results correctly. If such expertise is not available, we suggest that (i) the Technical Personal and Analyst roles execute Step VII together or (ii) the complexity of the experiment design should be reduced.

All participants found that SIEM fosters understandability, coordination, and extensibility of experiments.

9 CONCLUSION AND FUTURE WORK

We evaluated quality impacts of operational tasks in FaaS platforms as a foundation for a new generation of emerging serverless big data processing frameworks and platforms.

In support of this evaluation, we presented SIEM, a new evaluation method to understand and mitigate the quality impacts of operational tasks. We further contributed novel metrics for characterizing operational tasks. We instantiated SIEM to evaluate deployment package and function configuration changes for four major FaaS providers. Based on our results, we synthesize new generalized design guidelines for serverless big data processing frameworks.

In future work, we will provide additional tooling in support of SIEM to lower entrance barriers, improve collaboration, and simplify integration in an enterprise context. Furthermore, we will present our new FaaS-native big data processing framework based on the insights presented in this paper.

ACKNOWLEDGMENTS

The work in this paper was performed in the context of the DITAS and BloGPV.Blossom projects. DITAS is partially supported by the European Commission through the Horizon 2020 Research and Innovation program under contract 731945. BloGPV.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

[1] Chaitanya Baru, Milind Bhandarkar, Carlo Curino, Manuel Danisch, Michael Frank, Bhaskar Gowda, Hans-Arno Jacobsen, Huang Jie, Dileep Kumar, Raghunath Nambiar, et al. 2014. Discussion of BigBench: A Proposed Industry Standard Performance Benchmark for Big Data. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer International Publishing, Cham, 44–63.

[2] David Bermbach, Jörn Kuhlenkamp, Akon Dey, Arunmoezhi Ramachandran, Alan Fekete, and Stefan Tai. 2017. BenchFoundry: A Benchmarking Framework for Cloud Storage Services. In *Service-Oriented Computing (ICSOC'15)*. Springer International Publishing, Cham, 314–330.

[3] David Bermbach, Erik Wittern, and Stefan Tai. 2017. *Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective*. Springer International Publishing, Cham.

[4] Benjamin Congdon. 2018. GitHub Repository: Corral. http://github.com/bcongdon/corral. [Online; accessed 26.02.2019].

[5] 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 (SoCC '10)*. ACM, New York, NY, USA, 143–154.

[6] Matt Crane and Jimmy Lin. 2017. An Exploration of Serverless Architectures for Information Retrieval. In *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval (ICTIR'17)*. ACM, New York, NY, USA, 241–244.

[7] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'16)*. USENIX Association, Denver, CO, 14–19.

[8] Karl Huppler. 2009. The Art of Building a Good Benchmark. In *Performance Evaluation and Benchmarking*, Raghunath Nambiar and Meikel Poess (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–30.

[9] David Jackson and Gary Clynch. 2018. An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions. In *Proceedings of the 3rd International Workshop on Serverless Computing (WoSC'18)*. IEEE, 154–160.

[10] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*. ACM, New York, NY, USA, 445–451.

[11] Eric Jonas, Johann Schleier Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzales, Raluca A. Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* arXiv preprint arXiv:1902.03383 (9 Feb 2019), 1–33.

[12] Youngbin Kim and Jimmy Lin. 2018. Serverless Data Analytics with Flint. *CoRR* abs/1803.06354 (13 Aug 2018), 5.

[13] Markus Klems. 2016. *Experiment-driven Evaluation of Cloud-based Distributed Systems*. Ph.D. Dissertation. TU Berlin.

[14] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 427–444.

[15] Jörn Kuhlenkamp and Markus Klems. 2017. Costradamus: A Cost-Tracing System for Cloud-Based Software Services. In *Service-Oriented Computing (ICSOC'15)*. Springer International Publishing, Cham, 657–672.

[16] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. 2014. Benchmarking Scalability and Elasticity of Distributed Database Systems. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1219–1230.

[17] Jörn Kuhlenkamp and Sebastian Werner. 2018. Benchmarking FaaS Platforms: Call for Community Participation. In *Proceedings of the 3rd International Workshop on Serverless Computing (WoSC'18)*. 189–194.

[18] 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.

[19] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. 2018. Evaluation of Production Serverless Computing Environments. In *Proceedings of the IEEE 11th International Conference on Cloud Computing (CLOUD'18)*. IEEE, 442–450.

[20] Philipp Leitner, Erik Wittern, Josef Spillner, and Waldemar Hummer. 2018. A Mixed-Method Empirical Study of Function-As-A-Service Software Development in Industrial Practice. *Journal of Systems and Software* 149 (18 Dec 2018), 340 – 359.

[21] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E'18)*. IEEE, 159–169.

[22] Pedro G. López, Marc Sánchez Artigas, Gerard París, Daniel B. Pons, Alvaro R. Ollobarren, and David A. Pinto. 2018. Comparison of FaaS Orchestration Systems. In *Proceedings of the 3rd International Workshop on Serverless Computing (WoSC'18)*. IEEE, 148–153.

[23] Maciej Malawski, Kamil Figiela, Adam Gajek, and Adam Zima. 2018. Benchmarking Heterogeneous Cloud Functions. In *Euro-Par 2017: Parallel Processing Workshops (Euro-Par'17)*. Springer International Publishing, Cham, 415–426.

[24] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. 2017. Serverless Execution of Scientific Workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Generation Computer Systems* (4 Nov 2017).

[25] Johannes Manner, Martin Endreß, Tobis Heckel, and Guido Wirtz. 2018. Cold Start Influencing Factors in Function as a Service. In *Proceedings of the 3rd International Workshop on Serverless Computing (WoSC'18)*. IEEE, 181–188.

[26] Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. 2017. Data-driven Serverless Functions for Object Storage. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware'17)*. ACM, New York, NY, USA, 121–133.

[27] Bianca Schroeder, Adam Wierman, and Mor Harchol Balter. 2006. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation (NSDI'06)*. USENIX Association, Berkeley, CA, USA, 18–18.

[28] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. 2015. How to Build a Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*. ACM, New York, NY, USA, 333–336.

[29] Erwin van Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, and Simon Eismann. 2018. A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE'18)*. ACM, New York, NY, USA, 21–24.

[30] Sebastian Werner, Jörn Kuhlenkamp, Markus Klems, Johannes Müller, and Stefan Tai. 2018. Serverless Big Data Processing Using Matrix Multiplication as Example. In *Proceedings of the IEEE International Conference on Big Data (Big Data'18)*. IEEE, Seattle, WA, USA, 358–365.