BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms

Martin Grambow, Tobias Pfandzelter, Luk Burchard, Carsten Schubert, Max Zhao, David Bermbach

TU Berlin & Einstein Center Digital Future
Mobile Cloud Computing Research Group
Berlin, Germany
{mg, tp, lubu, casc, mazh, db}@mcc.tu-berlin.de

Abstract—Following the increasing interest and adoption of FaaS systems, benchmarking frameworks for determining non-functional properties have also emerged. While existing (microbenchmark) frameworks only evaluate single aspects of FaaS platforms, a more holistic, application-driven approach is still missing.

In this paper, we design and present BeFaaS, an extensible application-centric benchmarking framework for FaaS environments that focuses on the evaluation of FaaS platforms through realistic and typical examples of FaaS applications. BeFaaS includes a built-in e-commerce benchmark, is extensible for new workload profiles and new platforms, supports federated benchmark runs in which the benchmark application is distributed over multiple providers, and supports a fine-grained result analysis.

Our evaluation compares three major FaaS providers in single cloud provider setups and shows that BeFaaS is capable of running each benchmark automatically with minimal configuration effort and providing detailed insights for each interaction.

Index Terms—FaaS, Benchmarking, Fog Computing, Infrastructure Automation

I. Introduction

All major cloud providers offer Function-as-a-Service (FaaS) solutions where users only have to take care of their source code (functions) while the underlying infrastructure and environment is abstracted away by the provider. FaaS-based applications are split by their business functionality into individual functions which are deployed on a FaaS platform which, e.g., handles the execution and automatic scaling. The developer does not have any direct control over the infrastructure and can only define high-level parameters, such as the region in which the function should run. This complicates an already challenging comparison of cloud providers [1], [2], as the cloud variability is further compounded by an additional, unknown infrastructure component.

Existing work dealing with benchmarking of FaaS platforms focuses on the execution of small, so-called microbenchmarks which deploy and call a simple function (e.g., a matrix multiplication [3] or a random number generator [4]). While microbenchmarks are useful for studying and comparing specific characteristics, they can give only focused and limited insights into the platform behavior that applications can expect [5]. An application-centric benchmark, in contrast, mimics the behavior of a realistic application while closely observing the platform behavior. This allows developers to better compare different service options, a strategy also taken by the

TPC benchmarks[1]. To the best of our knowledge, such an application-centric benchmark for FaaS platforms does not exist yet.

To address this gap, we here propose *BeFaaS*, an extensible framework for executing application-centric benchmarks against FaaS platforms that includes a realistic e-commerce example benchmark. BeFaaS is also the first benchmarking framework with out-of-the-box support for federated cloud [6] setups which allows us to evaluate complex configurations in which an application is distributed over multiple FaaS platforms running on a mixture of cloud, edge, and fog nodes. Beyond this, BeFaaS is focused on ease-of-use and collects fine-grained measurements which can be used for a detailed post-experiment drill-down analysis, e.g., to identify cold starts or other request-level effects.

In this regard, we make the following contributions:

- We derive requirements for an application-centric FaaS benchmarking framework.
- We propose BeFaaS, an extensible framework for the execution of application-centric FaaS benchmarks and describe our example benchmark.
- We present our proof-of-concept prototype which is available as open source and currently supports six FaaS platforms.
- We run a number of experiments and use them to compare three public FaaS offerings.

This paper is structured as follows: After outlining the related work in Section II and deriving the requirements for an application-centric FaaS benchmark in Section III, we present the design, architecture, and features of BeFaaS in Section IV. Next, we describe our implementation of BeFaaS including the built-in e-commerce benchmark in Section V which we then use to evaluate three FaaS platforms (Section VI). Finally, we discuss the current limitations and future work in Section VII before concluding in Section VIII.

II. Related Work

Existing research on benchmarking of FaaS environments has so far focused on microbenchmarks. Application-centric benchmarks that consider the overall performance of multiple

---

[1]www.tpc.org

functions, the interaction with external services, and the effects of different application load profiles are mostly still missing.

Microbenchmarks call single functions repeatedly and evaluate the resulting metrics. These functions are often designed for a specific purpose, e.g., to stress the CPU of the test system or to evaluate the test system with a disk-intensive workload. Multiple performance evaluation studies are based on microbenchmarks which compare FaaS vendors, e.g., [3], [4], [7]–[12]. Besides scaling of functions, cold start latency, and instance lifetimes, the studies also evaluate metrics such as CPU utilization, network throughput, and costs. Almost all experiments, however, focus on single isolated aspects and do not create comparability of platforms for FaaS application developers.

Some studies also consider more complex applications such as image processing [13], analyze chained functions, or deploy real world applications on serverless platforms [11]. While these papers also use application-centric workloads for experiments, their goal was not to propose a comprehensive framework for the execution of application-centric FaaS benchmarks.

PanOpticon [14] uses a deployment, workload, and metrics module to evaluate chained functions and a simple chat server on two different FaaS vendors. Although PanOpticon has similar goals as BeFaaS, it neither supports detailed drill-down analysis nor federated multi-provider setups. Also, van Eyk et al. [15] developed a high-level architecture and stated requirements for serverless benchmarking. While their project has a similar goal as BeFaaS, it unfortunately seems to still be in a vision state. Existing preliminary source code components are, according to the paper, not available online, whereas we publish BeFaaS as an open-source research prototype.

Beyond FaaS, there are a number of application-centric benchmarking frameworks in other domains, e.g., for database and storage systems [16], [17] or for virtual machines [18]. These can, however, not easily be adapted to FaaS platforms.

III. Requirements

While microbenchmarks are highly useful for studying individual features of a system-under-test (SUT), application-centric benchmarks support end-to-end comparison of different platforms and configurations. Aside from standard benchmarking requirements such as portability or fairness [5], [16], [19]–[21], an application-centric FaaS benchmarking framework needs to fulfill a number of specific requirements which we describe in this section.

R1 – Realistic Benchmark Application: The performance of a FaaS platform depends on the application that is deployed on it. For instance, an application that frequently causes cold starts through a growing request rate will be better off on AWS Lambda while an application that frequently causes cold starts through short temporary load spikes will be better off on Apache OpenWhisk due to their different request queuing mechanisms [22]. This means that the benchmark application should be as close as possible to the real application for which

the analysis is made [5], e.g., in line with the findings of [23]. A key requirement is, hence, that *a FaaS benchmark should mimic real applications as closely as possible.*

R2 – Extensibility for New Workloads: FaaS platforms are highly flexible and can be used for a wide variety of applications, so the world of FaaS applications is evolving rapidly. As such, any set of "typical" FaaS applications – and thus the workload profile for a FaaS platform – can only be considered a snapshot in time. Likewise, the load profiles of existing FaaS applications, i.e., the amount and type of requests that the application handles, are likely to evolve over time. Therefore, we argue that *a FaaS benchmarking framework should be easily extensible in terms of adding new benchmark applications and updating load profiles for existing benchmarks.*

R3 – Support for Modern Deployments: FaaS is often used as the "glue" between cloud services, web APIs, and legacy systems. Thus, a benchmarking framework must also consider these links and support external services. Furthermore, today's applications are often distributed over cloud, edge, and fog resources [24]–[26]. Here, for example, hybrid clouds can keep sensitive functions on premises while non-critical functions are hosted in a public cloud; similar setups exist for edge and fog computing use cases [27]–[29]. As such, assuming a single-cloud deployment is unrealistic for benchmarks aiming to be as similar as possible to realistic applications. *A benchmarking framework needs to support external services and federated setups in which application functions are deployed on one or more FaaS platforms distributed across cloud, edge, and fog.*

R4 – Extensibility for New Platforms: Today, all major cloud service providers offer FaaS platforms and there is a growing range of open-source FaaS systems, for example, systems that specifically target the edge [30], [31]. As interfaces are constantly evolving and new platforms are introduced, a cross-platform benchmarking framework *needs to be extensible to support future FaaS platforms.*

R5 – Support for Drill-down Analysis: An application-centric FaaS benchmark can help to evaluate the suitability of different sets and configurations of FaaS platforms for a specific application. What it can usually not provide are explanations for its finding, e.g., the different cold start management behavior of AWS Lambda and Apache OpenWhisk mentioned above [22]. To facilitate root cause analysis and help evaluators explain the patterns they see in the benchmark results, we argue that *an application-centric FaaS benchmarking framework should support drill-down analysis by logging fine-grained measurement results including typical metrics of microbenchmarks.*

R6 – Minimum Required Configuration Overhead: An application-centric FaaS benchmarking framework should be easy to use and provide reproducible results. This includes configuration, deployment, execution, as well as collection and
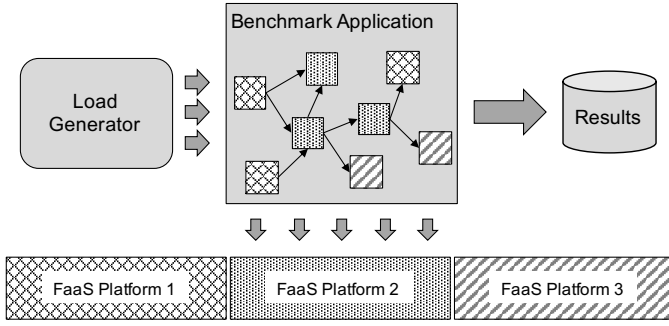
Fig. 1. High-level overview of the BeFaaS architecture.

analysis of results, e.g., based on infrastructure automation. Hence, *a FaaS benchmarking framework should be designed to require as little manual effort as possible*.

IV. DESIGN

In this section, we give an overview of the BeFaaS design, starting with an overview of the BeFaaS architecture and components (Section IV-A) before describing the key features of BeFaaS (Sections IV-B to IV-E).

A. Architecture and Components

In BeFaaS, the execution of functions of a benchmark application is the *workload* that actually benchmarks the FaaS platform, i.e., executing a function creates *stress* on the SUT. Since functions do not "self-start" executing, we need an additional load generator that invokes the FaaS functions of our benchmark application; see also Figure 1 for a high-level architecture overview.

For a benchmark run, BeFaaS requires three inputs: (i) the source code of the FaaS functions forming the benchmark application, (ii) a load profile for the load generator, and (iii) a deployment configuration that describes the environment configuration for each function and FaaS platform (the SUTs).

For a benchmark run, application code and deployment configuration are initially converted into deployment artifacts by the *Deployment Compiler*. The Deployment Compiler instruments and wraps each function's code with BeFaaS library calls and injects vendor-specific instructions defined in deployment adapters which enables request tracing and fine-grained metrics. The resulting deployment artifacts are passed to the *Benchmark Manager*.

The Benchmark Manager orchestrates the experiment: First, it sets up the *SUT* by deploying each function based on the information in the respective artifact. If there are external services, these can either be deployed by the Benchmark Manager as well or linked to the SUT using environment variables. In the second step, it initializes the *Load Generator* with the workload information described in a load profile. Then, the benchmark run is triggered and the Load Generator invokes the functions of the benchmark application which log every request in detail including timestamps, origin function, and called functions (if applicable). Finally, once the benchmark run is completed, the Benchmark Manager collects the log
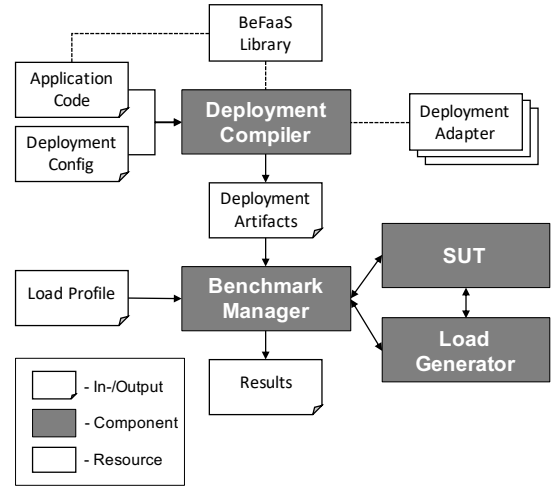

Fig. 2. The Deployment Compiler transforms application code into individual deployment artifacts based on a deployment configuration. These are then deployed and benchmarked by the Load Generator. Finally, the Benchmark Manager aggregates and reports fine-grained results.

files from all FaaS platforms used, aggregates them into a joint results file, and destroys all provisioned resources; see Figure 2 for an overview of the components in the BeFaaS framework and their interactions.

B. Realistic Benchmarks

To provide a relevant and realistic application-centric benchmark (**R1**), BeFaaS already comes with one built-in benchmark which mimics an e-commerce scenario and represents a typical use case for FaaS applications (this application is explained in further detail in Section V). Our benchmark adheres to the empirical findings of Shahrad et al. [23], is composed of several functions that interact with each other to form function chains, and uses external services such as a database system for persistence. The benchmark application comes with a default load profile that covers all relevant aspects as well as several further load profiles to emphasize selected stress situations, e.g., to provoke more cold starts. In combination, the benchmark represents a complete FaaS application: load balancing at the provider endpoint(s), interconnected calls of several functions, calls to external services such as database systems, and multiple load profiles which, e.g., provoke scaling of resources.

The modular design of BeFaaS, however, also allows us to easily add further benchmark applications and load profiles or to adapt existing ones to the concrete needs of the developer (**R2**). For adding a new benchmark, the respective application only needs to use the BeFaaS library (described in Section V) for function calls and to have unique function names.

C. Benchmark Portability and Federated FaaS Deployments

To support portability of benchmarks and federated deployments, BeFaaS relies on unique function names, individual deployment artifacts for every function, and a single endpoint for every deployed function (**R3**): With globally unique

function names, the endpoints of the deployed functions are already known during the compilation phase. The Deployment Compiler maps these endpoints to the canonical function names (defined in the application) and compiles them into the source code. Moreover, the compiler also injects endpoints to external services such as database systems using environment variables which were set in the respective setup script or defined manually. This decouples the ability of a function to call another function or a platform service from its deployment location and enables BeFaaS to support arbitrarily complex deployments: it is indeed possible to run every function on a different FaaS platform.

Each FaaS platform offers a different interface for life-cycle and configuration management of functions. As the smallest common interface, BeFaaS requires that each platform provides API-based access to (i) deploying functions, (ii) retrieving log entries from the standard logging interface, and (iii) removing functions. The Deployment Compiler wraps this functionality using an adapter mechanism and selects the appropriate instructions for the target platform specified in the deployment configuration. Additional FaaS platforms that fulfill this minimal interface can easily be added by implementing a corresponding adapter (**R4**).

D. Detailed Request Tracing

To enable a detailed drill-down analysis of experiment results (**R5**), the Deployment Compiler injects and wraps code that collects detailed measurements during the benchmark run: The compiler adds timestamping to determine start, end, and latency of calls to functions and external services.

Besides these timestamps, the compiler also injects code that generates context IDs and pair IDs to assign individual calls to their respective context later on. Here, a context ID is generated once for each function chain (the first function call) which is propagated to every subsequent call to other functions. To link the individual calls of a function chain, the compiler injects source code to create pair IDs of randomly generated keys that link calling and called function. Thus, it is possible to trace every single request through the benchmark application and to generate call trees for every context and function chain.

Finally, to independently and reliably detect cold starts, the Deployment Compiler also injects code that evaluates a local environment variable on the executor at the provider side. If this variable is not present, the function runs on a new executor (cold start), the variable is created, filled with a randomly generated key, and the cold start is logged.

All data that enable fine-grained results (timestamps, context IDs, pair IDs, and executor keys) are recorded on the console using the standard logging interface of the respective FaaS vendor. Initial experiments with Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure (Azure) have shown that the cost of logging is at most in the microsecond range.

E. Automated Experiment Orchestration

The BeFaaS framework requires only the application code, a deployment configuration, and a load profile to automatically perform the benchmark experiment (**R6**). First, all business logic, dependencies, and BeFaaS instrumentation logic are bundled into a single deployment artifact by the Deployment Compiler. Next, the Benchmark Manager orchestrates the experiment and provides a simple interface for starting the benchmark run, monitoring its process, and collecting fine-grained results for further analysis.

V. IMPLEMENTATION

Our open-source prototype implementation of BeFaaS[2] includes (i) the BeFaaS library, (ii) six deployment adapters, (iii) the Deployment Compiler, (iv) the Benchmark Manager, (v) one realistic benchmark application, and (vi) several load profiles for the benchmark application (see Figure 2).

The BeFaaS library is written in JavaScript and handles calls to other functions depending on their canonical name, generates tracing IDs, and takes timestamps. BeFaaS deployment adapters are implemented using Terraform[3] commands. Currently, BeFaaS thus supports three major cloud offerings (AWS Lambda, Google Cloud Functions, and Azure Functions) as well as the three open-source systems tinyFaaS [31], OpenFaaS, and OpenWhisk [32] which support the deployment of functions on private infrastructure, including edge or fog nodes. The Deployment Compiler is a shell script that uses several tools to build the deployment adapters for the respective platforms, parses and injects information from the Deployment Configuration, and generates the deployment artifacts from the application code. The Benchmark Manager uses Terraform to create the infrastructure based on these artifacts, collect the logs, and later remove provisioned resources. The implemented benchmark application is written in JavaScript and includes calls to external services such as a Redis[4] instance. The Load Generator uses Artillery[5] to call the benchmark application. New load profiles can easily be added by specifying new Artillery load descriptions (YAML[6] configuration files).

E-Commerce Application (Webshop)

Our e-commerce benchmark implements a webshop as a FaaS application based on Google's microservice demo application[7]. Our corresponding benchmark implementation follows a typical request-response invocation style, comprises 17 functions, and uses a Redis instance as an external service to persist state (see Figure 3). Besides functions that provide recommendations and advertising, customers can log in, set their preferred currency, view products, fill a virtual shopping cart, check out orders, and finally observe the shipping. Each

---

[2]https://github.com/Be-FaaS
[3]https://www.terraform.io/
[4]https://redis.io/
[5]https://artillery.io/
[6]https://yaml.org/
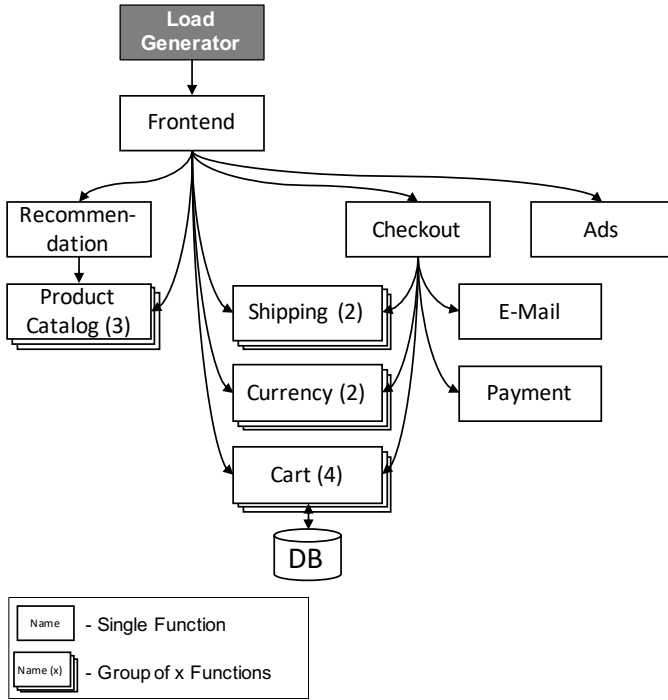[7]https://github.com/GoogleCloudPlatform/microservices-demo

Fig. 3. The e-commerce application implements a webshop in 17 functions. The frontend serves as a single entry point and an external database is used to store state.

task is implemented in a separate function (in the figure, we grouped some functions to increase legibility) and all requests arrive at a single function, the frontend, which takes the customer calls and routes them to the respective backend functions. There are blocking synchronous calls to other functions as well as asynchronous call blocks that idle until all functions returned.

The Load Generator for the benchmark application uses Artillery running in a Docker container that can be deployed on an arbitrary instance. It either executes a realistic default load profile that stresses all relevant aspects of the application or specific additional load profiles that emphasize stress situations, e.g., to provoke more cold starts. The default load profile simulates four different customer workflows and constant traffic for 15 minutes. The benchmark also includes alternative load profiles for a growth workload which linearly ramps up the load to 20 workflows per second over 15 minutes and a spike workload which suddenly increases the load from 3.5 to 20 workflows per second after five minutes, retains the high load for ten minutes, and finally continues with the lower load (3.5 workflows per second) for five minutes.

Our e-commerce benchmark is particularly well suited for comparing different cloud providers but can also be used to explore federated cloud deployments, e.g., for scenarios in which the application is running on multiple cloud platforms.

VI. EVALUATION

We evaluate BeFaaS in two different ways. We start by presenting the results of several experiments in which we
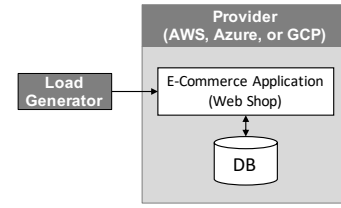

Fig. 4. As part of the FaaS application, the database instance is deployed in the same region and on the same provider as the rest of the webshop.

use BeFaaS to stress different FaaS platforms (Section VI-A). Afterwards, in Section VI-B, we discuss to which degree BeFaaS fulfills our requirements from Section III.

A. Experiments

To show how BeFaaS can be used to evaluate cloud FaaS platforms, we deploy BeFaaS in single cloud provider setups in which all functions of the respective benchmark application are deployed on a single provider. Here, we deploy the e-commerce benchmark on three major cloud providers (namely AWS, Azure, and GCP) and use the default load profile to compare them.

1) Setup: Figure 4 shows the basic setup of our cloud experiments: We deploy the Load Generator on a (vastly over-provisioned) virtual machine (2 vCPUs and 4 GB RAM) and let it execute the default load profile against the e-commerce application deployed in either *eu-west-1* for AWS, *westeurope* for Azure, or *europe-west1* for GCP. Moreover, the Redis database system used by the webshop also runs on an over-provisioned virtual machine (2 vCPUs and 4 GB RAM; *ta3.medium* at AWS, *Standard_B2S* in Azure, and *e2-medium* at GCP) at the respective provider site. This ensures that the database instance and Load Generator will not be a bottleneck during the experiment [5]. During each experiment, the Load Generator executes 18, 000 workflows, which each consist of 1 to 9 requests, over a time span of 15 minutes. Since the focus of this paper is on BeFaaS and its features and not on providing an in-depth performance analysis of different cloud providers, we decided not to repeat the experiment several times.

2) Results: Figure 5 shows the execution duration of four selected functions which are called from the frontend function (as boxplots, boxes represent quartiles, whiskers show the minimum and maximum values without outliers beyond 1.5 times the Inter Quartile Range). For the four functions examined in more detail, the overall picture is similar for all three providers: As expected, simpler functions that only read or write a single value have a lower execution duration than more complex ones such as the *getCart()* or *checkout()* function. In our experiment, Azure provided the fastest environment for this single run while AWS was notably slower with higher variance.

In a further fine-grained analysis, we investigate the distribution of computing, network transmission, and database query latency for a synchronous and blocking section which involves
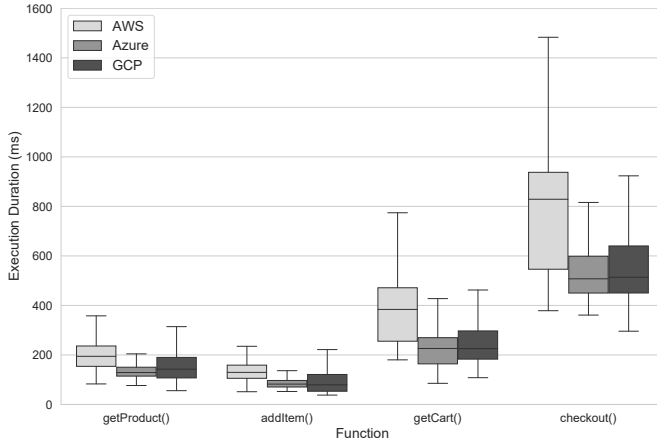
Fig. 5. A detailed analysis of four functions called from the frontend shows that Azure provides the best performance and that the execution duration has the highest variance on AWS.
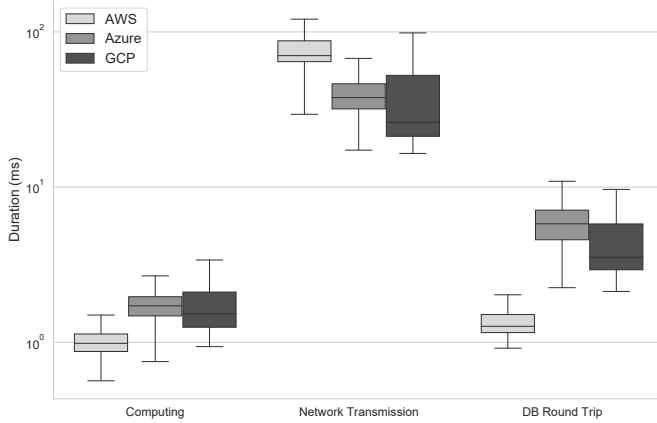

Fig. 6. A drill-down analysis of a function sequence reveals that the network transmission time is the most relevant driver of execution time on all providers.

two functions and database operations when an item is put into the shopping cart. For this evaluation, we consider the (i) computation part as function runtime without the duration of outgoing network calls, (ii) network latency as the duration of outgoing calls to other function without the runtime of the called function itself, and (iii) query latency as the duration of calls to the external database. The detailed timestamp mechanisms of BeFaaS allow us to easily separate these times, which are shown in Figure 6. Even though the results are based on only one benchmark run, it is noticeable that for all providers time is mostly spent on network transmission followed by the database round-trip time while the actual computing time is relatively low. In addition, AWS showed the fastest compute and database times for the experiment run compared to the other providers Azure and GCP, while also incurring significantly more time for network transmission.

B. Discussion of Requirements

In Section III, we had identified six requirements for application-centric FaaS benchmarking frameworks. We now discuss to which degree BeFaaS fulfills these requirements.

BeFaaS already comes with one standard benchmark that covers a representative FaaS application scenario, namely standard web applications, and can be easily extended by implementing more FaaS application scenarios using the Be-FaaS library. We, hence, believe that BeFaaS fulfills the requirements **R1** (*Realistic Benchmark Application*) and **R2** (*Extensibility for New Workloads*).

In BeFaaS, benchmark users can define arbitrarily complex deployment mappings of functions to target FaaS platforms including federated multi-cloud setups or mixed cloud/edge/fog deployments. In fact, each function could run on a different platform. To achieve this, BeFaaS transforms the benchmark application into deployment artifacts fitted to the target platform. Adding another target platform is also straightforward and only requires the benchmark user to implement an adapter component for the respective FaaS platform or to copy and adapt an existing adapter component. Based on this, we argue that BeFaaS fulfills the requirements **R3** (*Support for Modern Deployments*) and **R4** (*Extensibility for New Platforms*).

At runtime, BeFaaS collects fine-grained measurements and traces individual requests similar to what Dapper [33] does for microservice applications. This offers the necessary information basis for drill-down analysis. Beyond this, BeFaaS also offers visualization capabilities for select standard measurements to further support analysis needs. Overall, we hence conclude that BeFaaS addresses requirement **R5** (*Support for Drill-down Analysis*).

Finally, we believe that BeFaaS is easy to use due to its experiment automation features and requires only very few configuration files (requirement **R6** – *Minimum Required Configuration Overhead*). Nevertheless, this is a highly subjective matter that depends on the respective individual. Therefore, we invite all researchers to use our proof-of-concept prototype[8] and to try it out themselves.

VII. DISCUSSION

BeFaaS is a powerful, modern application-centric FaaS benchmarking framework. There are, however, also some points to consider and limits when using BeFaaS.

Tracing token generates constant network overhead. BeFaaS supports a detailed tracing of requests by injecting a small token in each call. On the one hand, this supports the clear mapping of different calls to function chains, yet on the other hand, it also causes an additional network overhead. This token, however, is mostly constant in size (depending on the length of the respective function name), so the overhead can be easily determined and considered in results analysis. Furthermore, this will only matter if the goal of the benchmark

---

[8]https://github.com/Be-FaaS

is to find the optimal deployment for an existing application which is then instrumented to be used as a BeFaaS benchmark.

No detailed measurements for external services. Currently, BeFaaS handles external services and components as a black-box and only measures end-to-end latency of such service calls. In future work, however, we plan to implement a small BeFaaS sidecar proxy that can be deployed on external service instances to forward calls to the respective service and to inject the BeFaaS tracing token there as well.

External services can affect the comparability and fairness. The included benchmark uses an external database system to persist state but further benchmarks and use cases may also require external services such as pub/sub message brokers or web APIs. Although the modular design of BeFaaS supports this, there are also some pitfalls in terms of fairness and comparability: In our experiments, we placed the database instance in the same region and deployed it at the same provider to minimize latency between functions and database. A function calling the external service and awaiting the response will not idle for a long time and the execution environment at the provider side will soon be available again for the next request. On the other hand, a function calling an external service in another region with larger latency will block the environment and (may) cause a cold start for the next incoming request. Thus, when using external services, these should be located and deployed with similar latency for all alternatives. Moreover, as cloud environments are virtually infinitely scalable, it has to be assured that the external service does not become a performance bottleneck during the experiment. Otherwise, the benchmark would benchmark the compute resources of the external service instead of the FaaS environment.

Provider-specific features can affect portability. Competing FaaS vendors are constantly developing new and exclusive features that simplify development and deployment for customers. These features, however, can also affect the portability of the BeFaaS framework if a (future) benchmark uses exclusive features that are not available at all vendors. Thus, we strongly recommend not to use exclusive features of individual providers when developing new BeFaaS benchmarks. BeFaaS can, however, help to determine the impact of new features within a provider or across multiple providers by adjusting and configuring the respective deployment adapter.

Clock synchronization is required for some drill-down analysis tasks. The drill-down analysis features of BeFaaS require approximately synchronized clocks. Although this will usually be provided by the provider with sufficient accuracy, a user should assert this before running experiments as this will affect the reliability of tracing insights. Nevertheless, such detailed insights may often not be needed and the tracing of BeFaaS also offers something to counteract this: If the call follows a request-response pattern, BeFaaS measures the total round trip time at the calling function and knows the

computing duration at the called function. Thus, it is possible to approximate the network transmission latency under the assumptions that both directions took comparably long. For event-based calls that do not return a message to the sender, however, this is not possible. In our experience, though, this is not a problem in the cloud and for self-hosted FaaS platforms, where the user has direct control over clock synchronization.

VIII. Conclusion

FaaS platforms are a popular cloud compute paradigm and have also been proposed for edge environments. For comparing and choosing different FaaS platforms in terms of performance, developers usually rely on benchmarking. Existing FaaS benchmarks, however, tend to fall into the microbenchmark category – an application-centric FaaS benchmarking framework is still missing.

In this paper, we presented BeFaaS, an extensible framework for executing application-centric benchmarks against FaaS platforms which comes with a realistic e-commerce benchmark. BeFaaS is also the first benchmarking framework with out-of-the-box support for federated cloud setups which allows us to also evaluate complex configurations in which an application is distributed over multiple FaaS platforms running on a mixture of cloud, edge, and fog nodes. We plan to use this feature of BeFaaS in future work to evaluate the feasibility of such infrastructure. Beyond this, BeFaaS is focused on ease-of-use through automation and collects fine-grained measurements which can be used for a detailed post-experiment drill-down analysis, e.g., to identify cold starts or other request-level effects; it can easily be extended with additional benchmarks or adapters for further FaaS platforms.

With BeFaaS, we provide developers with the necessary tool to explore, compare, and analyze FaaS platforms for their suitability for application scenarios. We also offer researchers the ability to study the performance effects of different FaaS deployment options across cloud, edge, and fog through experiments.

References

[1] P. Leitner and J. Cito, "Patterns in the Chaos - A Study of Performance Variation and Predictability in Public IaaS Clouds," in *Transactions on Internet Technology*. ACM, 2016.

[2] D. Bermbach, "Quality of Cloud Services: Expect the Unexpected," in *IEEE Internet Computing (Invited Paper)*. IEEE, 2017.

[3] T. Back and V. Andrikopoulos, "Using a Microbenchmark to Compare Function as a Service Solutions," in *Proc. of the European Conference on Service-Oriented and Cloud Computing (ESOCC '18)*. Springer, 2018.

[4] M. Malawski, K. Figiela, A. Gajek, and A. Zima, "Benchmarking Heterogeneous Cloud Functions," in *Proc. of the European Conference on Parallel Processing (Euro-Par '17)*. Springer, 2017.

[5] D. Bermbach, E. Wittern, and S. Tai, *Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective*. Springer, 2017.

[6] T. Kurze, M. Klems, D. Bermbach, A. Lenk, S. Tai, and M. Kunze, "Cloud Federation," in *Proc. of the International Conference on Cloud Computing, GRIDs, and Virtualization (CLOUD COMPUTING '11).* IARIA, 2011.

[7] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in *Proc. of the USENIX Annual Technical Conference (USENIX ATC '18).* USENIX, 2018.

[8] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski, "Performance evaluation of heterogeneous cloud functions," in *Concurrency and Computation: Practice and Experience.* Wiley, 2018.

[9] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold Start Influencing Factors in Function as a Service," in *Proc. of the 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion '18).* IEEE, 2018.

[10] H. Lee, K. Satyam, and G. Fox, "Evaluation of Production Serverless Computing Environments," in *Proc. of the International Conference on Cloud Computing (CLOUD '18).* IEEE, 2018.

[11] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing Serverless Platforms with ServerlessBench," in *Proc. of the Symposium on Cloud Computing (SoCC '20).* ACM, 2020.

[12] H. Martins, F. Araujo, and P. R. da Cunha, "Benchmarking Serverless Computing Platforms," in *Journal of Grid Computing.* Springer, 2020.

[13] J. Kim and K. Lee, "FunctionBench : A Suite of Workloads for Serverless Cloud Function Service," in *Proc. of the International Conference on Cloud Computing (CLOUD '19).* IEEE, 2019.

[14] N. Somu, N. Daw, U. Bellur, and P. Kulkarni, "PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications," in *Proc. of the International Conference on COMmunication Systems & NETworkS (COMSNETS '20).* IEEE, 2020.

[15] E. van Eyk, J. Scheuner, S. Eismann, C. L. Abad, and A. Iosup, "Beyond Microbenchmarks: The SPEC-RG Vision for A Comprehensive Serverless Benchmark," in *Proc. of the ACM/SPEC International Conference on Performance Engineering Companion (ICPE '20 Companion).* ACM, 2020.

[16] D. Bermbach, J. Kuhlenkamp, A. Dey, A. Ramachandran, A. Fekete, and S. Tai, "BenchFoundry: A Benchmarking Framework for Cloud Storage Services," in *Proc. of the International Conference on Service-Oriented Computing (ICSOC '17).* Springer, 2017.

[17] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases," in *Proc. of the International Conference on Very Large Data Bases (VLDB '13).* VLDB Endowment, 2013.

[18] A. H. Borhani, P. Leitner, B.-S. Lee, X. Li, and T. Hung, "WPress: An Application-Driven Performance Benchmark for Cloud-Based Virtual Machines," in *Proc. of the International Enterprise Distributed Object Computing Conference (EDOC '14).* IEEE, 2014.

[19] K. Huppler, "The Art of Building a Good Benchmark," in *Proc. of the First TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC '09).* Springer, 2009.

[20] D. Bermbach, J. Kuhlenkamp, A. Dey, S. Sakr, and R. Nambiar, "Towards an Extensible Middleware for Database Benchmarking," in *Proc. of the Technology Conference on Performance Evaluation and Benchmarking (TPCTC '14).* Springer, 2014.

[21] E. Folkerts, A. Alexandrov, K. Sachs, A. Iosup, V. Markl, and C. Tosun, "Benchmarking in the Cloud: What it Should, Can, and Cannot Be," in *Proc. of the TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC '12).* Springer, 2013.

[22] D. Bermbach, A.-S. Karakaya, and S. Buchholz, "Using Application Knowledge to Reduce Cold Starts in FaaS Services," in *Proc. of the ACM Symposium on Applied Computing (SAC '20).* ACM, 2020.

[23] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proc. of the USENIX Annual Technical Conference (USENIX ATC '20).* USENIX, 2020.

[24] D. Bermbach, F. Pallas, D. G. Pérez, P. Plebani, M. Anderson, R. Kat, and S. Tai, "A Research Perspective on Fog Computing," in *Proc. of the Workshop on IoT Systems Provisioning & Management for Context-Aware Smart Cities (ISYCC '17).* Springer, 2017.

[25] B. Zhang, N. Mor, J. Kolb, D. S. Chan, K. Lutz, E. Allman, J. Wawrzynek, E. Lee, and J. Kubiatowicz, "The Cloud is Not Enough: Saving IoT from the Cloud," in *Proc. of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '15).* USENIX, 2015.

[26] T. Pfandzelter, J. Hasenburg, and D. Bermbach, "Towards a Computing Platform for the LEO Edge," in *Proc. of the International Workshop on Edge Systems, Analytics and Networking (EdgeSys '21).* ACM, 2021.

[27] F. Pallas, P. Raschke, and D. Bermbach, "Fog Computing as Privacy Enabler," in *Internet Computing.* IEEE, 2020.

[28] M. Grambow, J. Hasenburg, and D. Bermbach, "Public Video Surveillance: Using the Fog to Increase Privacy," in *Proc. of the Workshop on Middleware and Applications for the Internet of Things (M4IoT '18).* ACM, 2018.

[29] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar, "Serverless Edge Computing: Vision and Challenges," in *Proc. of the Australasian Computer Science Week Multiconference (ACSW '21).* ACM, 2021.

[30] G. George, F. Bakir, R. Wolski, and C. Krintz, "NanoLambda: Implementing Functions as a Service at All Resource Scales for the Internet of Things." in *Proc. of the ACM/IEEE Symposium on Edge Computing (SEC '20).* ACM, 2020.

[31] T. Pfandzelter and D. Bermbach, "tinyFaaS: A Lightweight FaaS Platform for Edge Environments," in *Proc. of the International Conference on Fog Computing (ICFC '20).* IEEE, 2020.

[32] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The Serverless Trilemma: Function Composition for Serverless Computing," in *Proc. of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '17).* ACM, 2017.

[33] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google, Tech. Rep., 2010.