TriggerBench: A Performance Benchmark for Serverless Function Triggers (Short Paper)

Joel Scheuner, Marcus Bertilsson, Oskar Gronqvist, Henrik Tao,
Henrik Lagergren, Jan-Philipp Steghöfer, Philipp Leitner
Department of Computer Science and Engineering
Chalmers | University of Gothenburg
Gothenburg, Sweden
scheuner@chalmers.se, marcbert@student.chalmers.se, oskgro@student.chalmers.se, htao@student.chalmers.se,
henlag@student.chalmers.se, jan-philipp.steghofer@cse.gu.se, philipp.leitner@chalmers.se

***Abstract*—Serverless computing offers a scalable event-based paradigm for deploying managed cloud-native applications. Function triggers are essential building blocks in serverless, as they initiate any function execution. However, function triggering is insufficiently studied and inherently hard to measure given the distributed, ephemeral, and asynchronous nature of event-based function coordination. To address this gap, we present *TriggerBench*, a cross-provider benchmark for evaluating serverless function triggers based on distributed tracing. We evaluate the latency of eight types of triggers in Microsoft Azure and three in AWS. Our results show that all triggers suffer from long tail latency, storage triggers introduce variable multi-second delays, and HTTP triggers are most suitable for interactive applications. Our insights can guide developers in choosing optimal event or messaging triggers for latency-sensitive applications. Researchers can extend *TriggerBench* to study the latency, scalability, and reliability of further trigger types and cloud providers.**

***Index Terms*—serverless, FaaS, triggers, distributed tracing, observability, performance, benchmarking**

I. Introduction

Serverless computing emerged as a promising cloud computing paradigm and experiences strong interest in industry and academia. It aims to liberate users from operational concerns such as managing or scaling server infrastructure, by offering a high-level service with fine-grained billing [1, 2].

One important decision criterion for developers choosing a specific cloud provider is performance. Prior work has reported many performance challenges for serverless platforms [3, 4, 5, 6] such as coldstarts [7, 8, 9, 10], slow tail latency [9], and branch mispredictions in short-lived functions [11]. However, little work has studied the performance of event-based function triggers [5], although they are a core building block of practical serverless applications [12]. A study comparing serverless control flow methods [13] raised performance challenges for event-based triggers. Similarly, Pelle et al. [14] reported varying invocation delays in AWS depending on message payload size. However, these results are limited to a single provider and difficult to reproduce.

Understanding function triggers as the communication primitives for building serverless applications across cloud providers requires fine-grained tracing to address gaps in current research. Micro-benchmarks measure individual aspects of serverless platforms, such as the CPU speed or platform overhead of a simple Function-as-a-Service (FaaS) function, and are prevalent in prior work [5, 6]. However, they do not capture the end-to-end latency characteristics of multiple functions coordinated through event-based serverless triggering mechanisms. Prior work typically reports the response time of a synchronous HTTP request served by a serverless function, but event-based serverless architectures often use asynchronous triggers to orchestrate complex workflows [13, 12, 15, 16]. Therefore, fine-grained asynchronous tracing is required to identify bottlenecks and capture the event-based execution of serverless function coordination.

In this paper, we propose an initial cross-provider benchmark for evaluating serverless function triggers called *TriggerBench* and demonstrate its utility through experimentation in two leading cloud providers (Microsoft Azure and AWS). We adopt distributed tracing to collect and correlate performance traces for synchronous and asynchronous function triggers. Our methodology (Section II) can serve as a reference for researchers and practitioners to evaluate function triggers as foundational communication primitives of modern serverless applications. Our experimental results (Section III) for two leading cloud providers show that the Azure HTTP trigger performs best and most stable. However, queue and storage triggers in Azure are five times slower than their AWS counterparts. We identify triggers that suffer from extreme long tail latency and derive insights that can guide developers in choosing between four similar trigger types, namely streams, messages, events, and queues. Finally, we discuss our contributions (Section IV), relate them to prior research (Section V), and outline challenges for future work (Section VI). We release the *TriggerBench* benchmarking software, data, and results as a replication package[1] to foster future research.

II. TriggerBench

Figure 1 shows the high-level architecture of our benchmarking approach for trigger latency. The figure visualizes the main interactions between the two main components *benchmark orchestrator* and *cloud provider*. First ①, an *invoker*- and

[1]Zenodo archive planned upon peer-reviewed publication based on: https://github.com/joe4dev/trigger-bench/

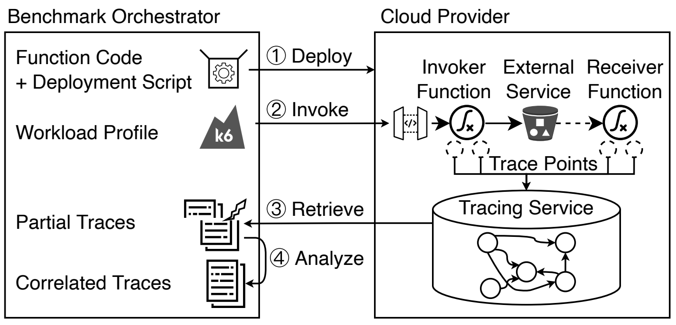


Figure 1. High-level overview of benchmarking approach.

receiver-function as well as an *external service* connecting these two functions (Section II-A) are deployed into a cloud provider using an automated deployment script. The invoker function calls the external service, which is configured to trigger the receiver function through different trigger types (Section II-B). All cloud resources are instrumented with detailed trace points (Section II-A) and forward trace spans to a provider-specific tracing service such as AWS X-Ray[2] or Azure Application Insights[3]. Second ②, a workload profile (Section II-C) is applied to invoke the invoker function though an HTTP gateway. Third ③, the benchmark orchestrator retrieves partial traces from the tracing service and ④ analyzes them by correlating disconnected traces and extracting relevant timestamps (Section II-D). This correlated trace dataset is then available for further performance analysis (e.g., Section III).

A. Measurement Methodology

This section describes the measurement methodology for *synchronous* and *asynchronous* function triggers. In both cases, we define *trigger latency* as the time difference between t_1 and t_4, where t_1 denotes the last line of user code before starting the service call in the invoker function, and t_4 denotes the first line of user code in the receiver function.

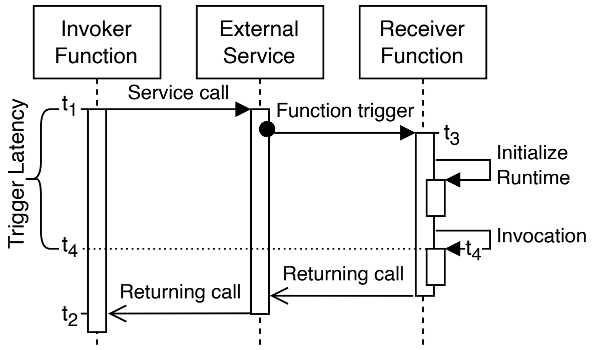
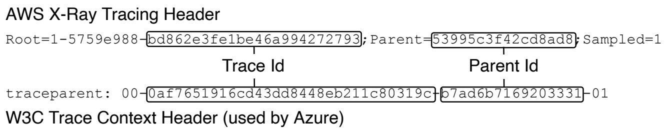


Figure 2. Trigger latency $\Delta(t_1, t_4)$ for *synchronous* function trigger.

The sequence diagram in Figure 2 visualizes *synchronous* function triggering where the invoker function and the external service remain active during the entire invocation of the receiver function. After t_1, the invoker function initiates a service call to an external service (e.g., API gateway), which triggers the receiver function within the cloud-provider internal infrastructure. The invoker function needs to attach a tracing token to the outgoing service call, typically in the form of an HTTP tracing header. Figure 3 shows examples of tracing headers for AWS X-Ray[4] and the W3C standard used by Azure called Trace Context[5]. Most importantly, these headers contain a *trace id* to correlate trace points of the *same* request from the invoker- and receiver-function (Figure 1). The *parent id* helps to generically identify the source of the service call by linking to the trace point initiating the service call.



```
Root=1-5759e988-bd862e3fe1be46a994272793;Parent=53995c3f42cd8ad8;Sampled=1
traceparent: 00-0af7651916cd43dd8448eb211c80319c-b7ad6b7169203331-01
```

Figure 3. Examples of HTTP tracing headers for trace context propagation.

At t_3, the function infrastructure (e.g., Firecracker [17] for AWS) receives the function execution request and starts to initialize the function runtime (e.g., Node.js). This initialization overhead can be substantial for coldstarts [7, 8, 9, 10], but is reduced for repeated invocations if the function is kept in memory. Subsequently, the function invocation starts by executing the user code in the receiver function at t_4. Once the function completes, the control flow returns via the external service back to the invoker function ending the service call in t_2.

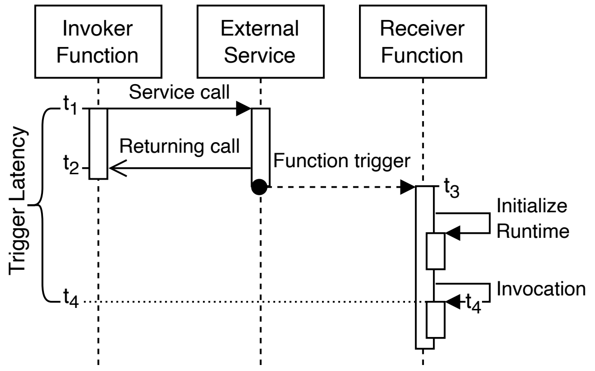


Figure 4. Trigger latency $\Delta(t_1, t_4)$ for *asynchronous* function trigger.

Figure 4 shows *asynchronous* function triggering where the invoker function potentially terminates before the external service triggers the receiver function. The service call immediately returns to the invoker function after completion (e.g., uploaded file to object storage). In the meantime, the external service asynchronously triggers the receiver function at t_3 and additional queueing time might occur before proceeding with

[2]https://aws.amazon.com/xray/

[3]https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview

[4]https://docs.aws.amazon.com/xray/latest/devguide/xray-concepts.html#xray-concepts-tracingheader

[5]https://www.w3.org/TR/trace-context/#trace-context-http-headers-format

initialization. Asynchronous triggers typically do not propagate a tracing header. Hence, the receiver function creates a *new trace id* that differs from the invoker function, essentially breaking the trace into two partial/disconnected traces. To mitigate this issue, we explicitly attach a tracing token to the metadata or payload of the service call. At t_4, we extract this tracing token and send it to the tracing service together with the new trace id of the receiver function. This custom trace propagation enables our trace analysis to correlate such partial traces (Section II-D). Finally, the function terminates without returning to any caller, which makes observability hard without distributed tracing.

B. Trigger Types

Table I summarizes eight common serverless trigger types used in this study. The most popular trigger type in serverless is the *HTTP* trigger according to an application characterization study [12] and a comprehensive analysis of the Azure production workload [15]. This trigger reacts to an inovcation of an HTTP endpoint via a corresponding HTTP request and is inherently synchronous (all other trigger types are asynchronous). In Azure Functions, the *queue* trigger causes the second-most invocations (33.5%) and is used by 15.2% of the functions [15]. Queues are first-in, first-out and trigger a function whenever a new message is received, thus enabling independent, asynchronous processing of items involving considerable workload. The cloud object *storage* trigger is important given that cloud storage is the most popular external service used by serverless applications [12]. The storage can trigger subscribed functions, e.g., when a new item is created or an existing item is modified.

Table I
TRIGGER TYPES AND SERVICE MAPPINGS FOR AWS AND AZURE

Trigger	AWS Service	Azure Service
HTTP	API Gateway	API Management
Storage	S3	Blob Storage
Queue	SQS	Queue Storage
Database	DynamoDB	CosmosDB
Event	SNS	Event Grid
Stream	Kinesis	Event Hubs
Message	EventBridge	Service Bus Topic
Timer	CloudWatch Events	Timer

We implement these three important triggers for the two leading cloud providers AWS and Azure [12] and an additional five triggers for Azure. *Database triggers* react to events in a database such as insertion, deletion, or update. *Event triggers* are provided by Azure Event Grid. Subscribers reliably receive events when the status of the service or application has changed and can react to these changes directly. *Stream triggers* are enabled via Azure EventHub and are useful when events arrive in series, the ingestion rate of events is very high, and low latency is desirable. If latency is not a major concern, *message triggers* via Azure Service Bus Topic can be used. The service provides a more reliable asynchronous delivery of critical payloads. Finally, *timer triggers* are used for scheduled execution of functions, either in a certain interval or at certain times of the day.

C. Workload Profile

We choose a baseline workload with a low request rate because a comprehensive characterization of the production workload from Azure Functions [15] has shown that 81 % of the applications were invoked at most once per minute on average. Hence, our baseline workload sends 1 request per second for 60 minutes to collect up to 3600 invocation samples. This low request rate also prevents excessive coldstarts or overloading any triggering infrastructure.

D. Trace Analysis

The trace analyzer correlates disconnected partial traces (see Figure 1) and yields a summary of fully correlated traces. For the synchronous HTTP trigger, the tracing services AWS X-Ray and Azure Application Insights support auto-correlation, where the receiver function detects a tracing header from the invoker function and associates it to the fully connected trace. For asynchronous triggers (Figure 4), trace token propagation is not supported for all our implemented trigger types except for the AWS storage trigger, which implements a custom trace re-parenting strategy. Therefore, each invocation of an asynchronous trigger creates two disconnected traces with distinct trace ids. Our trace analyzer uses the tracing token explicitly associated with t_4 (Section II-A) to match the *trace id* of the invoker function with the *new trace id* of the receiver function. This correlation process joins the two disconnected traces by id and enables our analyzer to extract results from the same request, including relevant timestamps identified in Section II-A and the coldstart status (i.e., whether a function experienced a coldstart).

E. Implementation

We implement *TriggerBench* as a Python library (CLI and SDK) to automate the entire benchmarking lifecycle (Figure 1). We use the provider-specific Node.js libraries to instrument the receiver and invoker functions using *Application Insights for Node.js*[6] and the *AWS X-Ray SDK*[7]. All triggers can be automatically deployed through Infrastructure-as-Code [18] using Pulumi[8] for modular and reproducible deployments. The open source load testing tool k6[9] provides reliable and customizable load generation. Our Python tool is easy to use because it leverages Docker virtualization to abstract dependencies and automatically inject provider credentials when needed.

III. EXPERIMENTAL RESULTS

This section describes the experiment setup and results of our benchmarking study and summarizes key findings.

[6]https://www.npmjs.com/package/applicationinsights
[7]https://www.npmjs.com/package/aws-xray-sdk
[8]https://www.pulumi.com/
[9]https://k6.io/

A. Setup

We deploy the serverless components (see Section II-A) in the AWS region *us-east-1* and the Azure region *eastus* as commonly used by other serverless studies [8, 19, 20, 21, 22]. For load generation, we deploy the *benchmark orchestrator* (see Figure 1) in an over-provisioned virtual machine (*t3.xlarge* for AWS, *B4ms* for Azure) within the same datacenter region as the serverless resources.

B. Results

The empirical cumulative distribution function (ECDF) plot in Figure 5 visualizes the trigger latency distribution for three AWS and eight Azure triggers investigated in our study on a logarithmic scale. It shows that the synchronous *HTTP* trigger in Azure delivers the best and most stable performance of all studied trigger types with a median latency of 32 ms (annotated and indicated by the dotted line) and a tail latency of 65 ms (99th percentile/p99). In comparison, the AWS HTTP trigger has higher tail latency (151 ms), but is still clearly the most responsive trigger type studied in AWS. The Azure *timer* trigger performs similar to the HTTP trigger, but is a few milliseconds slower. The asynchronous *database* trigger in Azure is surprisingly fast with 43 ms median latency but deteriorates towards the 95th percentile (86 ms) and exceeds 1.6 s delay in its extremely long tail (p99).

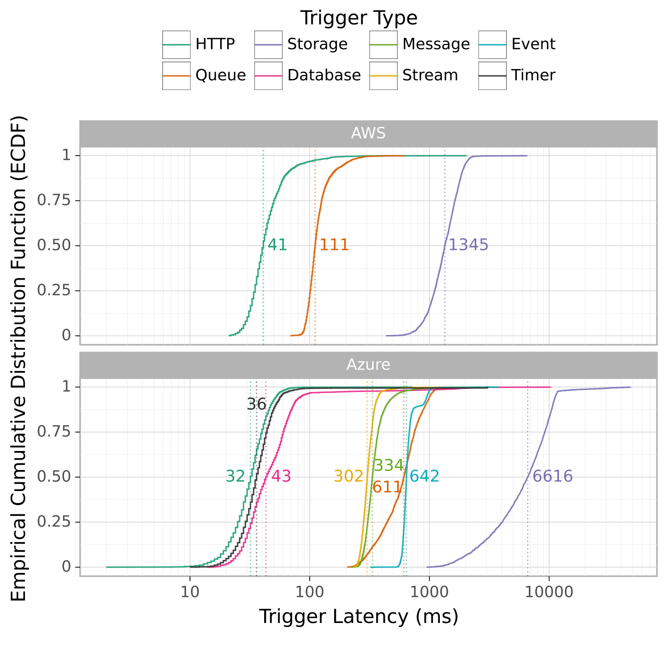


Figure 5. Trigger latency for three AWS and eight Azure triggers based on 3400–3600 samples per configuration excluding coldstarts. The y-axis is in log-scale and the dotted line denotes the median.

Azure offers four similar services for delivering events or messages with different performance profiles: streaming, messaging, queueing, and eventing. Among them, the *streaming* trigger performs best (302 ms median) and most stable with manageable tail latency (519 ms p99). The *message* trigger exhibits a similar performance profile, just adding more delay (+10 %) and tail latency (+54 %). In contrast, the *queue* trigger is more variable ranging from 205 ms (minimum) beyond 1155 ms (p99), indicated by the flat slope of its curve. Although the *event* trigger has a similar median (642 ms) than the queue trigger (611 ms), it performs more stable exemplified by its lower tail latency (1026 ms).

Storage-based triggers are the slowest type of triggers in both providers. They also suffer from extreme tail latency (p99) exemplified by their 23 s delay in Azure and 2.2 s delay in AWS.

The comparison between AWS and Azure shows that Azure offers the best HTTP trigger but AWS massively outperforms Azure for the queue and storage trigger by a factor of five.

Key Findings:

- The synchronous HTTP trigger has the shortest and most stable latency of all studied trigger types for both AWS and Azure.
- Storage triggers perform worst, introducing variable multi-second delays.
- All triggers suffer from long tail latency.
- The Azure queue and storage triggers are five times slower than their AWS counterparts.

IV. Discussion

We now discuss our results in the context of interactive applications and latency-sensitive function coordination.

A. Trigger types for interactive applications

From the studied trigger types, only the synchronous *HTTP* trigger is suitable for interactive applications that require immediate response. Users perceive an interactive reaction within 100 ms as immediate according to longstanding research on user experience [23, Chapter 5] and the user-centric performance model RAIL from Google [24]. Hence, our results for the HTTP trigger show that even tail latency can satisfy this requirement for Azure (p99) and most AWS users (p96), which makes sense because latency is most important for synchronous triggers. However, our results demonstrate a best-case scenario and substantial delay could be added by other factors such as coldstarts [7, 8, 9, 10] and bursty workloads [9]. The Azure *database* trigger has almost the same median latency as the AWS HTTP trigger, but is hardly suitable for interactive scenarios because enabling user interaction following an asynchronous trigger adds further delay. An additional concern for some applications might be its extreme tail latency (1614 ms p99). We disregard the asynchronous *timer* trigger because it is typically not used in latency-sensitive applications. Instead, the primary use case for this type of trigger is in scheduling periodic background tasks, which are not sensitive to delays.

Many trigger types could be suitable for interactive applications where users are freely navigating and can cope with task delays between 100 ms and 1000 ms [23, 24], for example loading a new web page. As counter-examples, *storage* triggering is clearly unsuitable for interactive applications and the Azure trigger types *database*, *event*, and *queue* are problematic due to their tail latencies.

For delays beyond $10\,\mathrm{s}$, users are unwilling to wait and are likely to abandon a task [23, 24], such as waiting for an image to process. The AWS storage trigger is suitable for such a common task, but using an Azure storage trigger would likely lead to frustrated users due to excessive tail latency ($23\,\mathrm{s}$ p99).

B. Latency-sensitive function coordination

From the studied trigger types, the *stream* trigger for Azure and *queue* trigger for AWS are most suitable for connecting multiple functions asynchronously and efficiently. Asynchronous function coordination is essential in serverless because synchronous function chaining causes double-billing [25] and is subject to tight execution time limits (e.g., $30\,\mathrm{s}$ for AWS HTTP trigger). In Azure, the *stream* trigger offers the lowest latency for reliable and consistent function triggering. However, $302\,\mathrm{ms}$ median latency is rather slow for low-latency stream processing and even the AWS *queue* trigger performs 2.7 times better. It might be worthwhile to explore the *database* trigger for latency-critical applications where extreme tail latency is acceptable. In AWS, better triggering performance than the *queue* trigger could possibly be achieved with other trigger types beyond the scope of this study, for example through direct invocation using a CLI or SDK.

V. Related Work

This work complements a large body of work in the very active field of serverless performance benchmarking [5, 6, 26, 27] by addressing the research gap of cross-provider trigger benchmarking [5].

Closest to this work, Pelle et al. [14] study invocation delays of containers and serverless functions for different event payload sizes in AWS Lambda and evaluate a latency-sensitive drone application. In comparison, our work covers two leading providers rather than solely AWS, contributes a reusable, reproducible, and extensible benchmark rather than merely reporting results, and characterizes the full empirical distribution based on $34\times$ more samples published in a documented replication package rather than mean and standard deviation from 100 samples per configuration. We also demonstrate and implement more detailed tracing using trace token propagation and generalizable trace correlation to mitigate the limitations of existing tracing systems, especially for asynchronous triggers causing disconnected traces.

Quinn et al. [13] indirectly measure trigger performance in their study about alternative control flow methods in serverless. They report average latency of an application in different control flow architectures but do not isolate trigger latency, probably using three (partly unspecified) triggers. In an evaluation of serverless production platforms, Lee et al. [28] study the median throughput of three trigger types (HTTP, storage, database) across four providers (not all triggers are available for all providers). Our work focuses on latency rather than throughput, which requires more elaborate measurement methodology and is harder to achieve in current performance trends [29].

VI. Conclusion

In this paper, we designed and implemented *TriggerBench*, a cross-provider benchmark for evaluating serverless function triggers, which are essential communication primitives for building serverless applications. We demonstrate the utility of *TriggerBench* in a benchmarking study covering eight triggers on Azure and three triggers on AWS. Our results show that all triggers suffer from long tail latency, synchronous HTTP triggers are most suitable for interactive applications, and storage triggers introduce variable multi-second delays. In conclusion, our study highlights the importance of trigger benchmarking to guide developers in choosing between similar event-based triggers (e.g., stream, message, event, and queue) when building latency-sensitive applications. We discuss that many triggers are currently unsuitable for building interactive serverless applications and asynchronous triggers with lower and less variable trigger latency are needed for efficient function coordination.

Future work can use *TriggerBench* to study relevant aspects related to function triggering. Different runtimes (e.g., Node.js, Java, Python) affect the initialization overhead included in the trigger latency as defined in Section II-A. The detailed tracing of TriggerBench could help to quantify this overhead using t_3 and study coldstarts in more detail than prior work. Similar to Pelle et al. [14], the effect of event payloads could be studied, potentially isolating transfer times by leveraging t_2 (Figure 4). Ustiugov et al. [9] indicated that bursty workloads affect function performance and TriggerBench could be used to study bursty workloads for different trigger types. Finally, a longitudinal study would be interesting to monitor performance fluctuations over time (e.g., peaks at certain times of the day or week [30]).

We contribute this initial version of *TriggerBench* to the research community as an enabler for future studies covering more providers, trigger types, and experiment designs. Our tool already supports basic authentication for other providers and our existing implementations for trace downloading and analysis can guide provider extensions. Our plugin-based architecture makes it easy to add additional trigger types without code changes in the benchmark orchestrator (Figure 1). Similarly, our SDK can orchestrate complex experiment scenarios (e.g., periodic re-deployments) and workload profiles are supported through flexible and advanced k6 scenarios[10].

Acknowledgment

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

[10]https://k6.io/docs/using-k6/scenarios/

References

[1] E. V. Eyk, A. Iosup, S. Seif, and M. Thömmes, "The SPEC Cloud group's research vision on FaaS and serverless architectures," in *Proceedings of the 2nd International Workshop on Serverless Computing (WOSC)*. ACM, 2017, pp. 1–4.

[2] P. C. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, Nov. 2019.

[3] H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, "Survey on serverless computing," *Journal of Cloud Computing*, vol. 10, no. 1, p. 39, 2021.

[4] J. Wen, Z. Chen, Y. Liu, Y. Lou, Y. Ma, G. Huang, X. Jin, and X. Liu, "An empirical study on challenges of application development in serverless computing," in *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2021, pp. 416–428.

[5] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," *Journal of Systems and Software (JSS)*, vol. 170, 2020.

[6] A. Raza, I. Matta, N. Akhtar, V. Kalavri, and V. Isahagian, "Sok: Function-as-a-service: From an application developer's perspective," *Journal of Systems Research (JSys)*, vol. 1, no. 1, 2021.

[7] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *Companion of the 11th IEEE/ACM UCC: 4th International Workshop on Serverless Computing (WoSC)*, Dec. 2018, pp. 181–88.

[8] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, Jul. 2018, pp. 133–146.

[9] D. Ustiugov, T. Amariucai, and B. Grot, "Analyzing tail latency in serverless clouds with stellar," in *IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2021.

[10] P. Maissen, P. Felber, P. G. Kropf, and V. Schiavoni, "Faasdom: a benchmark suite for serverless computing," in *The 14th ACM International Conference on Distributed and Event-based Systems (DEBS)*. ACM, 2020, pp. 73–84.

[11] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2019, pp. 1063–1075.

[12] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering (TSE)*, 2021.

[13] S. Quinn, R. Cordingly, and W. Lloyd, "Implications of alternative serverless application control flow methods," in *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC)*, ser. WoSC '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 17–22.

[14] I. Pelle, J. Czentye, J. Dóka, and B. Sonkoly, "Towards latency sensitive cloud native applications: A performance study on AWS," in *Proceedings of the 12th IEEE International Conference on Cloud Computing (CLOUD)*, E. Bertino, C. K. Chang, P. Chen, E. Damiani, M. Goul, and K. Oyama, Eds. IEEE, Jul. 2019, pp. 272–280.

[15] 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 *2020 USENIX Annual Technical Conference (ATC)*, A. Gavrilovska and E. Zadok, Eds. USENIX Association, 2020, pp. 205–218.

[16] P. G. López, A. Arjona, J. Sampé, A. Slominski, and L. Villard, "Triggerflow: trigger-based orchestration of serverless workflows," in *The 14th ACM International Conference on Distributed and Event-based Systems (DEBS)*, J. Gascon-Samson, K. Zhang, K. Daudjee, and B. Kemme, Eds. ACM, 2020, pp. 3–14.

[17] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2020, pp. 419–434.

[18] M. Hüttermann, *Infrastructure as Code*, ser. DevOps for Developers. Apress, 2012.

[19] R. Cordingly, H. Yu, V. Hoang, D. Perez, D. Foster, Z. Sadeghi, R. Hatchett, and W. J. Lloyd, "Implications of programming language selection for serverless data processing pipelines," in *IEEE International Conference on Cloud and Big Data (CBDCom)*. IEEE, 2020, pp. 704–711.

[20] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: a serverless benchmark suite for function-as-a-service computing," in *22nd International Middleware Conference*. ACM, 2021, pp. 64–78.

[21] D. Barcelona-Pons and P. García-López, "Benchmarking parallelism in faas platforms," *Future Generation Computer Systems*, vol. 124, pp. 268–284, 2021.

[22] J. Wen, Y. Liu, Z. Chen, J. Chen, and Y. Ma, "Characterizing commodity serverless computing platforms," *Journal of Software: Evolution and Process*, 2021.

[23] J. Nielsen, *Usability engineering*. Morgan Kaufmann, 1994.

[24] Google, "Measure performance with the rail model," 2015, accessed April 2022. [Online]. Available: https://www.smashingmagazine.com/2015/10/rail-user-centric-model-performance/

[25] 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 *Proceedings of the ACM SIGPLAN International Sym-*

posium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), ser. Onward! 2017. New York, NY, USA: ACM, 2017, pp. 89–103.

[26] V. Yussupov, U. Breitenbücher, F. Leymann, and M. Wurster, "A systematic mapping study on engineering function-as-a-service platforms and tools," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, ser. UCC'19, K. Johnson, J. Spillner, D. Klusácek, and A. Anjum, Eds. New York, NY, USA: ACM, 2019, pp. 229–240.

[27] J. Spillner and M. Al-Ameen, "Serverless literature dataset," Apr. 2019.

[28] H. Lee, K. Satyam, and G. C. Fox, "Evaluation of production serverless computing environments," in *Proceedings of the 11th IEEE CLOUD: 3rd International Workshop on Serverless Computing (WoSC)*, Jul. 2018, pp. 442–50.

[29] D. A. Patterson, "Latency lags bandwith," *Communication of the ACM*, vol. 47, no. 10, pp. 71–75, 2004.

[30] P. Leitner and J. Cito, "Patterns in the chaos – a study of performance variation and predictability in public IaaS clouds," *ACM Transactions on Internet Technology*, vol. 16, no. 3, pp. 1–23, Apr. 2016.