CrossFit: Fine-grained Benchmarking of Serverless Application Performance across Cloud Providers

Joel Scheuner, Rui Deng, Jan-Philipp Steghöfer, Philipp Leitner
Department of Computer Science and Engineering
Chalmers | University of Gothenburg
Gothenburg, Sweden
scheuner@chalmers.se, ruidengsweden@gmail.com, jan-philipp.steghofer@cse.gu.se, philipp.leitner@chalmers.se

***Abstract*—Serverless computing emerged as a promising cloud computing paradigm for deploying cloud-native applications but raises new performance challenges. Existing performance evaluation studies focus on micro-benchmarking to measure an individual aspect of serverless functions, such as CPU speed, but lack an in-depth analysis of differences in application performance across cloud providers. This paper presents *CrossFit*, an approach for detailed and fair cross-provider performance benchmarking of serverless applications based on a provider-independent tracing model. Our case study demonstrates how detailed distributed tracing enables drill-down analysis to explain performance differences between two leading cloud providers, AWS and Azure. The results for an asynchronous application show that trigger time contributes most delay to the end-to-end latency and explains the main performance difference between cloud providers. Our results further reveal how increasing and bursty workloads affect performance stability, median latency, and tail latency.**

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

I. Introduction

Serverless computing emerged as a promising cloud computing paradigm and experiences strong interest in industry and academia. Serverless computing [1] aims to liberate users from operational concerns, such as managing or scaling server infrastructure, by offering a fully-managed high-level service with fine-grained billing [2]. One embodiment of serverless computing is Function-as-a-Service (FaaS), where FaaS platforms (e.g., AWS Lambda) execute *functions*, i.e., event-triggered code snippets. Serverless developers can leverage a growing ecosystem of serverless offerings such as object storage (e.g., Amazon S3) to build inherently scalable applications and achieve faster time-to-market. These innovations stimulate the serverless market, which will continue to grow at rates above $20\,\%$ between 2021 and 2027 according to several market research reports [3, 4]. Serverless is also a hot topic in academic research with over 275 papers published between 2016 and 2020 as summarized in multiple literature reviews [5, 6, 7, 8, 9].

One important decision criterion for architects and developers who choose a cloud provider to build serverless applications is the performance of the application on different candidate cloud platforms. Application performance matters to the end-user experience [10, 11] and many performance challenges have been reported for serverless platforms [5, 12, 7, 8]. Further, operational costs are directly connected to application performance given the pay-per-use pricing model, which is typically based on execution duration and resource consumption.

Understanding serverless application performance 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 FaaS function, and are prevalent in prior work [7, 8]. However, they cannot capture the end-to-end latency characteristics of realistic serverless applications, which often integrate other cloud services [13, 14, 15]. Prior work typically reports the response time of a synchronous HTTP request served by a serverless application but event-based serverless architectures often use asynchronous triggers to orchestrate complex workflows [16, 13, 15, 17]. Therefore, fine-grained asynchronous tracing is required to identify bottlenecks and capture the event-based execution of serverless applications.

Although fairness is a key characteristic of a benchmark [18, 19, 20], serverless benchmarking lacks a transparent approach to address fairness in the design of applications and tracing instrumentation. Following guidance on how to build a benchmark, "fairness ensures that systems can compete on their merits without artificial constraints" [18]. Fair benchmark design should be motivated by relevance (i.e., important functionality) and carefully balance over-simplified universality (i.e., lowest common denominator) and over-specific innovation (i.e., bleeding edge) [20]. In heterogeneous serverless environments, fairness is particularly challenging to address because the underlying infrastructure is abstracted away with vendor-specific implementations.

In this paper, we design an approach for detailed and fair cross-provider benchmarking called *CrossFit* and demonstrate drill-down analysis for an application in two leading cloud providers. We adopt distributed tracing (Section II) to generate fine-grained performance profiles for an asynchronous application instead of solely reporting the overall response time. We focus on fairness by thoroughly designing and discussing the application architecture, tracing design, and workload generation for fair cross-provider comparison. We motivate the choice of a benchmark application based on real-world application characteristics [13, 15, 21, 1] to overcome the limitations of micro-benchmarks. Our methodology (Section III) can serve as a reference for researchers and practitioners to conduct detailed and fair application-level benchmarking of serverless platforms. For practitioners (developers/architects) who decided to use serverless computing, the methodology and insights from this paper can guide them in selecting the

most suitable cloud provider and services for their serverless applications and improve their understanding of serverless application performance. The results of our case study (Section IV) for two leading cloud providers show that storage triggering dominates the end-to-end latency and explains the main performance difference between different providers. Finally, we discuss our contributions (Section V), relate them to prior research (Section VI), and outline challenges for future work (Section VII).

In summary, the main contributions of this paper are:

1) introducing a provider-independent tracing model for serverless applications;
2) providing guidelines for fair serverless benchmarking covering the design of applications and tracing;
3) demonstrating detailed distributed tracing and drill-down analysis for an application in two leading cloud providers (AWS and Azure); and
4) releasing the *CrossFit* benchmarking software, data, and results as a replication package[1].

II. Background

This section introduces serverless computing and distributed tracing.

A. Serverless Computing

In serverless computing, applications are composed of fully-managed components. The general computing component FaaS (i.e., the serverless function) often serves as a 'glue' to connect external services, such as an API gateway or database, while performing generic tasks, such as validation or format conversion. Serverless functions execute in ephemeral environments, where cloud providers manage and scale the underlying infrastructure, typically containers. A one-off initialization called *cold start* occurs whenever a new function instance needs to be provisioned to serve a function execution request. This provisioning step for a function instance can, e.g., include the start of a container and a language runtime. Subsequent warm invocations perform much faster without initialization. However, cloud providers recycle idle function instances after some time, which makes functions stateless. Therefore, external services are necessary to persist any output of function computation. External services can also define cloud events, which can trigger a serverless function (e.g., upon insertion of a database entry). Such asynchronous event-based function triggering is archetypical for serverless although other alternatives exist for coordinating a workflow of functions such as client orchestration, coordinator functions, and state machines [16, 22].

B. Distributed Tracing

Distributed tracing [23, 24, 25] aims to achieve end-to-end observability of a request across distributed components. Figure 1 visualizes an end-to-end backend trace for a synchronous application with a causal invocation chain starting

[1]Anonymous version for double-blind review: https://github.com/serverless-crossfit/replication-package

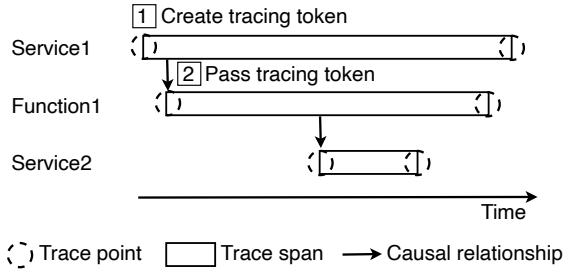

Figure 1. Simplified causal-time trace diagram of a synchronous invocation chain.

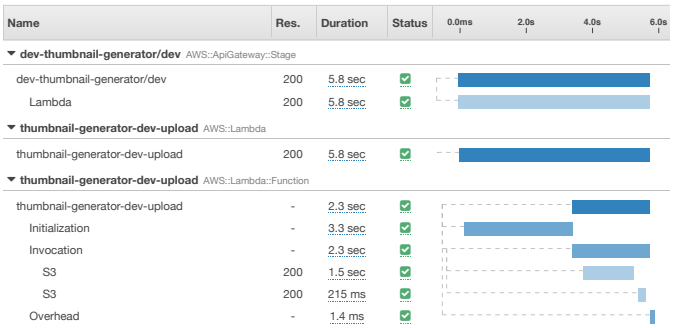

Figure 2. AWS X-Ray timeline view of a trace from a Lambda function experiencing a cold start. The trace follows the structure depicted in Figure 1.

from *Service1* over *Function1* into *Service2*. The service receiving an incoming request (e.g., *Service1*) generates a unique tracing token 1 for each request. This tracing token is then used to label each timestamp captured at trace points of interest and needs to be passed 2 into every downstream service along the invocation chain. Two consecutive timestamps are grouped into a trace *span* if they encompass a specific operation (e.g., computation) from the same component (e.g., *Service2*). A centralized tracing service correlates spans of the same request from all components using the tracing token to build a trace graph with causal relationships.

Serverless computing raises several challenges for distributed tracing. Provider-managed infrastructure limits access for fine-grained instrumentation and developers need to rely on distributed tracing services offered by cloud providers. This leads to observability gaps and typically requires implicit tracing of downstream services due to missing tracing support. Further, the event-based nature of serverless requires adaptations to traditional critical path analysis for synchronous invocation patterns [26]. Overall, serverless tracing is still immature and an industrial survey raised microservice tracing and analysis as a new big problem for software engineering [27].

We briefly summarize the distributed tracing services of the cloud providers AWS and Azure:

1) AWS X-Ray: AWS X-Ray[2] is a fully managed tracing service offering a web-based user interface for visualizing tracing data and an SDK in different programming languages

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

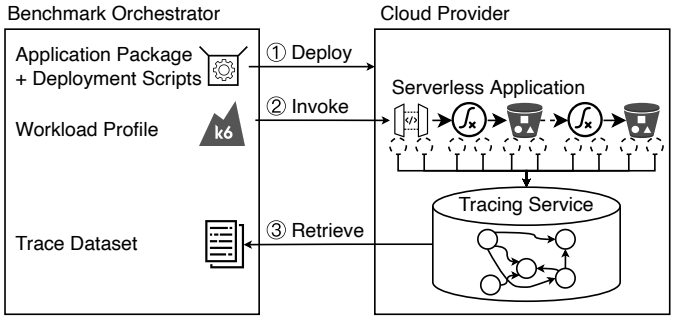

Figure 3. High-level overview of benchmarking approach.

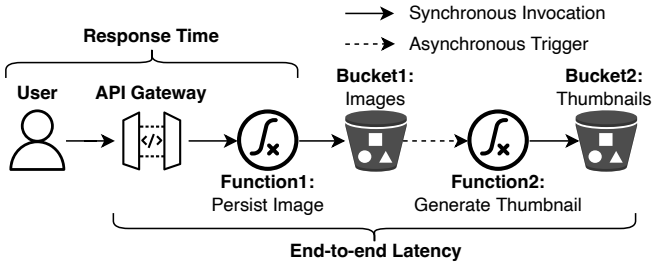

Figure 4. Architecture of thumbnail generator application.

for instrumentation and interacting with the X-Ray HTTP API. For minimal instrumentation overhead, X-Ray uses an asynchronous daemon to batch-upload trace data captured at each trace point to the X-Ray service. Many AWS services provide various levels of integration with X-Ray, including (semi-)automated instrumentation, dependency tracking, and request sampling. The X-Ray SDK provides a mechanism to capture additional timestamps and annotate custom metadata. X-Ray automatically correlates trace spans (called segments) of a request and offers an API to retrieve the end-to-end trace as a JSON document. Figure 2 shows an X-Ray timeline view of a single Lambda function invoking the Amazon S3 storage service.

2) Azure Application Insights: Azure Application Insights[3] provides similar functionalities as AWS X-Ray but differs in several aspects. Unlike the proprietary X-Ray specification, Azure adopts the open-source observability framework OpenTelemetry[4]. The maturity and support of features such as automated dependency tracking vary depending on the programming language. Trace spans of different functions are collected separately and need to be correlated manually.

III. Benchmark Design

We first introduce the high-level approach of our detailed cross-provider benchmarking model and then motivate the design of the application, fairness, instrumentation, and workload.

A. Overview

Figure 3 shows the high-level architecture of our application benchmarking approach. The figure visualizes the main interactions between the two main components *benchmark orchestrator* and *cloud provider*. The benchmark orchestrator is a client-side workload generator to automate (Section III-G) the workflow of an experiment. First ①, a serverless application (Section III-B) is deployed into a cloud provider using an automated deployment script. For fair cross-provider comparison (Section III-C), an application needs to be configured appropriately. The application is instrumented with detailed trace points (Section III-D) and forwards trace spans to a provider-specific tracing service (Section III-E). Second ②, a workload profile (Section III-F) is applied to invoke the serverless application deployed within the cloud provider. Third ③, the benchmark orchestrator retrieves correlated traces from the tracing service to build a trace dataset for performance analysis (Section IV).

B. Application Design

We select an application called *thumbnail generator* previously used for exploring cross-provider migration [28], performance prediction [29], and performance evaluation [30]. This application has representative, real-world characteristics [13, 15, 21]. Instead of prevalent single function applications [7, 8], we cover a small but non-trivial multi-function application based on the finding that serverless applications typically contain 5 or fewer functions [13]. Instead of isolated function computation, our application integrates a common external service (in this case cloud storage) in an idiomatic way for serverless through asynchronous triggering rather than traditional synchronous request/response-interaction. In contrast to prior studies, we address fair cross-provider comparison and derive actionable insights through detailed distributed tracing.

A thumbnail generator is commonly used in web applications to resize images for responsive web design across multiple devices. Figure 4 visualizes the architecture of the thumbnail generator consisting of two chained functions connected through an asynchronous storage bucket trigger. The life cycle of the application execution is as follow:

1) A user uploads the original image via an HTTP request to the API Gateway endpoint.
2) The API Gateway triggers *Function1* to persist the image.
3) *Function1* receives an image from the API Gateway and writes it to object storage into *Bucket1*.
4) Once the image is written to storage, *Function2* is triggered to process the image.
5) *Function2* resizes the image and writes a thumbnail to *Bucket2*.

The *response time* of the application includes the synchronous interactive part of persisting an image but misses asynchronous function triggering. Hence, the actual image processing is only covered by the backend *end-to-end latency*.

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

[4]https://opentelemetry.io/

Table I
SERVICE MAPPING BETWEEN AWS AND AZURE FOR THE THUMBNAIL GENERATOR APPLICATION

	AWS	Azure
API Gateway	AWS API Gateway[5]	Azure API Management[6]
Function	AWS Lambda[7]	Azure Functions[8]
Object Storage	AWS S3[9]	Azure Blob Storage[10]

C. Fairness Design

We defined guidelines on 12 important aspects (a-l) to architect an application for fair performance comparison across cloud providers. We aim to simulate a scenario where a general-purpose serverless application is migrated to another provider, similar to prior work on cloud migration [28]. In this process, we strive to mitigate misconfigurations that could lead to a competitive advantage of one provider over another by design, for example by ignoring official recommendations or comparing different pieces of work. To balance between universality and innovation, we suggest the following guidelines for general-purpose applications and to avoid optional features that are highly provider-specific.

a) Application Architecture: **It is essential to maintain an equivalent architecture across cloud providers.** Prior work indicated that architecture-retaining migration is typically possible even for applications that are not optimized for portability [28].

b) Application Component Mapping: **High-level application components such as function computation or object storage need to be mapped based on their functionality because implementations differ for every cloud provider.** We adopt an established mapping from prior work on serverless migration [28] for the thumbnail generator application (Table I) and refer to an extensive list of service mappings for other components [31].

c) Function Resource Allocation: **For single-core applications, we suggest avoiding CPU throttling by aiming for a full vCPU core while minimizing over-provisioning (in line with Ustiugov et al. [32]).** The resources available to a serverless function in terms of CPU power, memory, and network connectivity vary depending on provider-specific configurations. Many performance benchmarking studies tried to reverse engineer provider-specific resource allocation policies [33, 34, 35], especially regarding CPU allocation. For Azure, neither the memory size (max. 1.5 GB) nor the CPU allocation (fixed) is configurable in the default consumption plan. For AWS, the memory size is configurable (128 MB to 10 248 MB) and determines the CPU allocation (proportional to memory size). Following the AWS documentation [36], we configure all Lambda functions with 1769 MB memory which is equivalent to one vCPU. CPU-intensive multi-core applications might require trade-off analysis with multiple configurations, dynamic optimization such as AWS Lambda Power Tuning [37], or runtime prediction such as Sizeless [38].

d) Costs: **Cost-relevant configuration options should at least strive to match pricing categories (e.g., standard vs. premium) because cost parity is often infeasible.** Costs are determined by cloud providers but usually depend on certain configuration options. Function computation is typically billed based on per-time resource consumption and the number of executions. For example, AWS and Azure charge \$0.20 per million executions and \$0.000 016 666 7 (AWS) and \$0.000 016 (Azure) for every GB-second execution time[11]. Given that Azure automatically determines the memory size, it is impossible to align costs dependent on memory size across providers. For object storage, we recommend the default on-demand general-purpose options for AWS and Azure rather than specialized low-price long-term options or high-price premium options. Ideally, the actual cost should be reported to compare the cost-performance ratio across providers.

e) Function Runtime: **The function runtime defines the execution environment and needs to match across providers.** Each cloud provider supports a list of managed function runtimes in specific versions (e.g., Node.js 14.x). The runtime determines the compatible programming languages (e.g., JavaScript), software libraries, operating system, and processor architecture (e.g., x86 or arm). Prior work has shown that performance differs by runtime, especially for cold start overhead [39]. Therefore, it is essential to use the same runtime and programming language preferably in the same version across providers. Fair comparison must only include mature production-ready versions rather than experimental preview versions. In this study, we choose .NET Core 3.1 and C# because essential instrumentation features were only available with this runtime in Azure.

f) Function Code Reuse: **We recommend programming language constructs such as classes, methods, or libraries to reuse as much code as possible across provider-specific implementations.** For the thumbnail generator, we reuse the business logic code for image resizing. Provider-specific code is inevitable but the high-level flow should match across providers. For example, we ensure that the storage client library is initialized at the same time in the control flow.

g) Function Operating System: **Fair comparison should adopt the recommended operating system for each provider.** We argue that comparable maturity is more relevant for fair comparison than using an identical operating system because the underlying operating system could be considered an implementation detail as long as the high-level functionality matches. Therefore, we choose *Amazon Linux 2* for AWS and *Windows* for Azure. Azure would offer a Linux operating system but prior work has shown that it performs very inconsistently [39].

[5]https://aws.amazon.com/api-gateway/
[6]https://azure.microsoft.com/en-us/services/api-management
[7]https://aws.amazon.com/lambda/
[8]https://azure.microsoft.com/en-us/services/functions
[9]https://aws.amazon.com/s3/
[10]https://azure.microsoft.com/en-us/services/storage/blobs/

[11]Pricing at experimentation time January 2022 for the data center regions *us-east-1* (AWS) and *East US* (Azure)

h) Function Pre-warming: **Premium cold start mitigation features such as function pre-warming should be disabled in most cases because they are highly provider-specific and violate the serverless premise of fine-grained pay-per-use billing.** We actively detect cold starts through tracing instead of preventing them through intrusive options such as provisioned concurrency [40] in AWS or the Azure Functions Premium plan [41].

i) Function Triggering Mechanism: **The triggering mechanism should be aligned across providers and configurable poll intervals should be reported.** Function triggering can either happen event-driven (i.e., push-based) or poll-based and varies between services. Reactive event-driven triggers are typically much faster than potentially unreliable poll-based triggers that periodically check for new cloud events (e.g., database insertion). For example, AWS uses event-driven S3 notifications to trigger functions but the default Azure Blob Storage trigger uses periodic pulling to scan the storage container logs for events (e.g., created file). To avoid up to 10-minute cold start delays, Azure recommends [42] using the EventGrid[12] trigger.

j) Data Center Location: **We suggest choosing established data centers in geographically close regions.** We deploy the application in the AWS region *us-east-1* and the Azure region *eastus* as commonly used by other serverless studies [33, 43, 35, 44, 45].

k) Metrics: **We focus on fine-grained latency metrics because latency is typically more relevant than bandwidth [46], especially in massive scale-out cloud environments.** Average performance is only useful for summarizing cost [47] but currently over-used in serverless [7] and cloud [48] performance studies. Instead, we visualize the full performance distribution as violin plots and focus on typical performance (50^{th} percentile), tail latency (95^{th} percentile), and stability (standard deviation). We derive the fine-grained latency metrics from the instrumentation design (Section III-D).

l) Workload: **The same workload is required for fair cross-provider comparison.** Our workload generation approach allows for sharing workload models (Section III-F) across applications and cloud providers.

D. Instrumentation Design

This section introduces a detailed provider-agnostic trace model showcased with the thumbnail generator application.

Figure 5 visualizes 13 interesting timestamps (explained in Table II) along the critical path for the thumbnail generator application (depicted in Figure 4). Distributed tracing provides spans with a start and end timestamp for each service used in an application (e.g., API Gateway). Serverless functions have multiple spans: an outer span includes infrastructure-related operations such as initialization and an inner span for the execution of user code. Similarly, storage I/O transactions have an outer span that includes authentication operations and an inner span for the actual storage data transfer.

[12]https://azure.microsoft.com/en-us/services/event-grid/

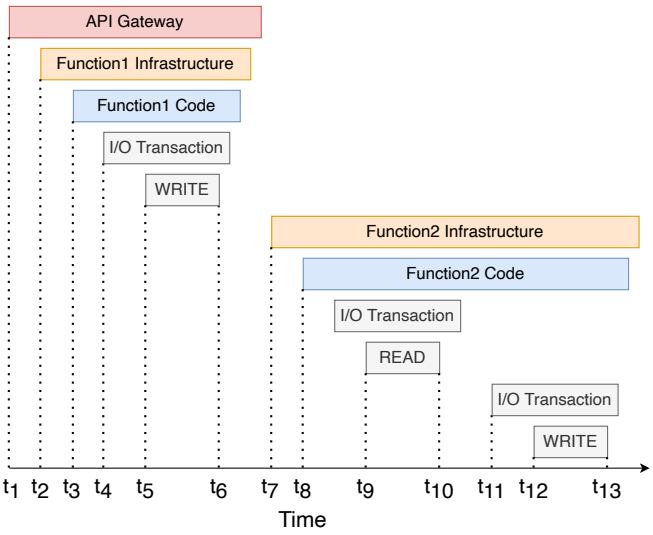

Figure 5. Trace design for thumbnail generator application.

Table II describes each operation between pairwise timestamps along the critical path from an incoming HTTP request (t_1) until the processed thumbnail image is written to object storage (t_{13}). Function computation (e.g., *CompF1*) is preceded by initialization (e.g., *InitF1*) and trigger (e.g., *TrigH*) operations. Storage transactions include authentication overhead (e.g., *AuthF1*) before the actual read/write operation (e.g., *WriteF1*). Our model includes important timestamps available in both AWS and Azure and therefore needs to omit some finer-grained timestamps that are unmappable across providers. For example, AWS traces could further split function initialization (e.g., *InitF1*) into container and runtime initialization.

E. Instrumentation Implementation

We now summarize how the provider-agnostic trace model can be implemented for two leading cloud providers and highlight some of the challenges we encountered since the field of serverless tracing is still relatively immature.

1) Instrumentation for AWS: We enable AWS X-Ray tracing and augment out-of-the-box tracing with custom instrumentation using the AWS X-Ray .Net SDK[13] to obtain the timestamps t_3, t_4, t_8, and t_{11}.

2) Instrumentation for Azure: Analogous to AWS, we add custom instrumentation using the Azure Application Insights C# SDK[14] to obtain the same timestamps unavailable with out-of-the-box tracing. Unlike AWS X-Ray, Azure Application Insights lacks automated correlation of spans originating from different functions. We manually pass the trace id using object storage metadata between the two functions and add it as a custom property to the trace of *Function2*. We subsequently correlate the two distinct traces during analysis based on this tracebility metadata. We acknowledge that adding metadata to object storage and traces adds a small overhead in

[13]https://docs.aws.amazon.com/xray/latest/devguide/xray-sdk-dotnet.html
[14]https://github.com/microsoft/ApplicationInsights-dotnet

Table II
SUMMARY OF TIMESTAMPS AND OPERATIONS BETWEEN TIMESTAMPS FOLLOWING THE TRACE DESIGN IN FIGURE 5

Operation		Timestamp			Operation	
Description	Name		Description		Name	Description
HTTP triggering: Time to trigger F1 from an HTTP request	*TrigH*	t_1	API gateway receives HTTP request	t_1		
		t_2	F1 (upload) gets triggered	t_2	*InitF1*	Startup overhead of F1: container + runtime initialization
Computation of F1: HTTP event parsing	*CompF1*	t_3	F1 executes first line of user code	t_3		
		t_4	F1 initiates WRITE to storage operation	t_4	*AuthF1*	Storage overhead: authenticate with storage for WRITE
Data transfer: write data from F1 to storage	*WriteF1*	t_5	F1 starts sending data to storage	t_5		
		t_6	F1 completes image transfer to storage	t_6	*TrigS*	Storage triggering: Time to trigger F2 from a storage event
Startup overhead of F2: container + runtime initialization	*InitF2*	t_7	F2 (create thumbnail) gets triggered	t_7		
		t_8	F2 executes first line of user code (READ)	t_8	*AuthF2r*	Storage overhead: authenticate with storage for READ
Data transfer: read data from storage to F2	*ReadF2*	t_9	F2 starts receiving data from storage	t_9		
		t_{10}	F2 completes image transfer from storage	t_{10}	*CompF2*	Computation of F2: resize image
Storage overhead: authenticate with storage for WRITE	*AuthF2w*	t_{11}	F2 initiates WRITE to storage operation	t_{11}		
		t_{12}	F2 starts sending data to storage	t_{12}	*WriteF2*	Data transfer: write data from F2 to storage
		t_{13}	F2 completes image transfer to storage	t_{13}		

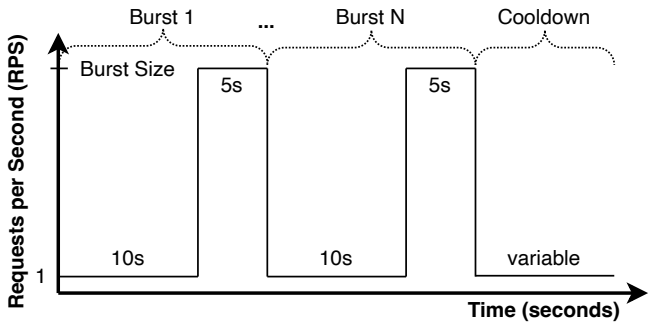

Figure 6. Bursty workload model.

Azure. However, trace correlation is necessary for comparable instrumentation and AWS implements a similar approach under the hood. To detect cold starts in Azure, additional scale controller logs [49] need to be enabled and manually correlated with the function traces.

F. Workload Design

We compare a constant baseline workload against three bursty workloads with different levels of burstiness. The constant workload `C` generates 1 request per second and runs for 5 minutes yielding 300 invocations. We choose 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. At the same time, the most popular 18.6 % applications are responsible for 99.6% of all invocations and exhibit higher invocation rates of at least 1 request per minute on average [15]. Additionally, FaaS workloads exhibit highly dynamic bursty workload patterns as confirmed by production workloads from Azure [15] and Alibaba [50] as well as by a characterization study of serverless applications from diverse cloud providers [13]. To represent these higher invocation rates and bursty workload characteristics, we designed three bursty workloads (`B1`-`B3`) following the model in Figure 6. The bursty workload model distributes a fixed number of requests (e.g., 300 to match the constant baseline) over a variable number of bursts of a given size. After `N` bursts, a cooldown phase follows to meet the target number of requests. We instantiate the bursty workload model with increasing burst sizes:

`B1` 4 bursts of size 12 with 20 seconds cooldown.
`B2` 2 bursts of size 25 with 30 seconds cooldown.
`B3` 1 burst of size 50 with 40 seconds cooldown.

G. Experiment Automation

We automate our benchmarking approach to mitigate reproducibility challenges, which are common in serverless [7] and cloud experimentation [48]. We use a Python-based benchmarking orchestration framework that leverages Docker containerization for packaging dependencies and integrates powerful load testing using K6[15]. We automate application deployment using the open-source infrastructure as code [51] tool Terraform[16]. We retrieve telemetry traces and logs through the API of the provider-specific tracing service. Our trace processing script correlates disconnected traces and exports all timestamps and operations from Table II including additional metadata (e.g., cold start status) into a structured CSV file for further analysis. The workloads from Section III-F are provided as K6 configurations reusable across applications and cloud providers. All code, configuration, and data is documented and available in our replication package[17].

[15]https://k6.io/
[16]https://www.terraform.io/
[17]Anonymous version for double-blind review: https://github.com/serverless-crossfit/replication-package

IV. Case Study

This section demonstrates detailed distributed tracing for the thumbnail generator application in two leading cloud providers, AWS and Azure. We first summarize the experiment setup before presenting and discussing results for the detailed latency breakdown and the different workloads from Section III-F.

A. Experiment Setup

We instantiate the benchmark design from Section III by conducting a performance experiment in AWS and Azure. All workloads are executed from a local computer to simulate user interaction with the thumbnail generator application. The location of the load generator has a limited impact on the results because we use backend tracing rather than relying on client-side response time measurements as common in prior work. We repeat all workloads multiple times and report representative executions from January 2022. We focus on the typical scenario of warm invocations and filter out cold starts because they exhibit fundamentally different performance characteristics and need to be studied separately.

B. Latency Breakdown

The violin plot in Figure 7 visualizes the detailed latency breakdown of the thumbnail generator application comparing the providers AWS and Azure. We group the operations into three duration categories: <2500 ms (left), <700 ms (middle), and <70 ms (right).

The left group shows that storage triggering dominates the total duration and explains the performance difference between providers. Asynchronous storage triggering is by far the slowest operation within the thumbnail generator application for both providers introducing delays of (485.0 ± 117.0) ms (median±standard deviation) for Azure and very unpredictable (1176.0 ± 355.0) ms for AWS.

The middle group reveals interesting differences across providers. The HTTP trigger *TrigH* has practically the same median for both providers but Azure exhibits much higher variability as the standard deviation shows: (17.0 ± 3.8) ms for AWS vs. (16.0 ± 6.2) ms for Azure. For the lightweight *Function1*, *InitF1* shows that AWS exhibits consistently less initialization overhead than Azure, which suffers from extreme outliers going beyond 500ms. In contrast, the heavyweight *Function2* with larger code libraries initializes almost instantly in Azure compared to much slower initialization in AWS (67.0 ± 31.0) ms. This caching effect in Azure is caused by their in-process function scheduling[18], where both functions share the same process and therefore *InitF1* includes the initialization overhead for both functions. For the computation in Function2 (*CompF2*), AWS performs ~40 % faster than Azure in single-core performance with the selected memory configuration (1769 MB for AWS, see Section III-C). Furthermore, larger memory configurations with multi-core virtual CPUs are available for AWS at higher cost, which can be beneficial for parallelizable workloads. Storage I/O operations for both read (*ReadF2*) and write (*WriteF1*, *WriteF2*) perform faster and more predictable on Azure compared to AWS, which suffers from slow tail performance, especially for write operations.

The right group contributes almost negligible time towards the total duration because *Function1* performs no computation and authentication is cached for warm invocations and hence primarily affects cold starts.

Key Findings:

- Asynchronous storage triggering dominates the end-to-end latency.
- The storage trigger in Azure performs significantly faster and more predictable than in AWS.
- Azure delivers faster and more predictable end-to-end latency for the thumbnail generator application than AWS because of its faster storage triggering.
- The AWS HTTP trigger delivers more predictable performance compared to Azure.
- Azure storage operations (read/write) are faster and more predictable than in AWS under low load.

C. Workload Types

Figure 8 compares the baseline constant workload `C` to bursty workloads with different burst sizes (`B1`-`B3`). The *total* duration (top left) shows that bursty workloads change the shape of the distribution and introduce more variability with increasing burst size. For AWS, bursty workloads exhibit distributions that are more skewed towards having more tail latency outliers compared to the flat distribution for the constant baseline workload. For Azure, bursty workloads introduce a bimodal distribution and significant variability compared to the relatively predictable constant baseline.

The computation for the second function *CompF2* exemplifies the effects of bursty workloads compared to the constant baseline workload. For both providers, the median performance increases and major performance variability is introduced. AWS develops a clear bimodal distribution in addition to slower tail latency observed for Azure as well.

Storage I/O operations such as *WriteF1* already suffer from very long tail latency performance in the constant baseline workload, especially in AWS. Bursty workloads might deteriorate the situation but additional burst rates should be tested to consolidate the trend because `B3` shows lower variability than expected. While the other write operation (*WriteF2*) follows a similar pattern, the read operation (*ReadF2*) is less predictable in Azure than in AWS.

The initialization of the first function *InitF1* explains the large variability caused by bursty workloads in Azure. In addition to the Azure in-process caching effects discussed in the previous section, Azure tends to re-use existing function instances rather than provision new ones. This scheduling policy reduces cold starts (omitted in these results) but introduces massive queueing delays captured in *InitF1*. In contrast, AWS delivers predictable warm initialization performance at the cost of more frequent cold starts because their scheduling policy provisions new

[18]Default behavior in the latest stable .NET version (3.1) during benchmark development.

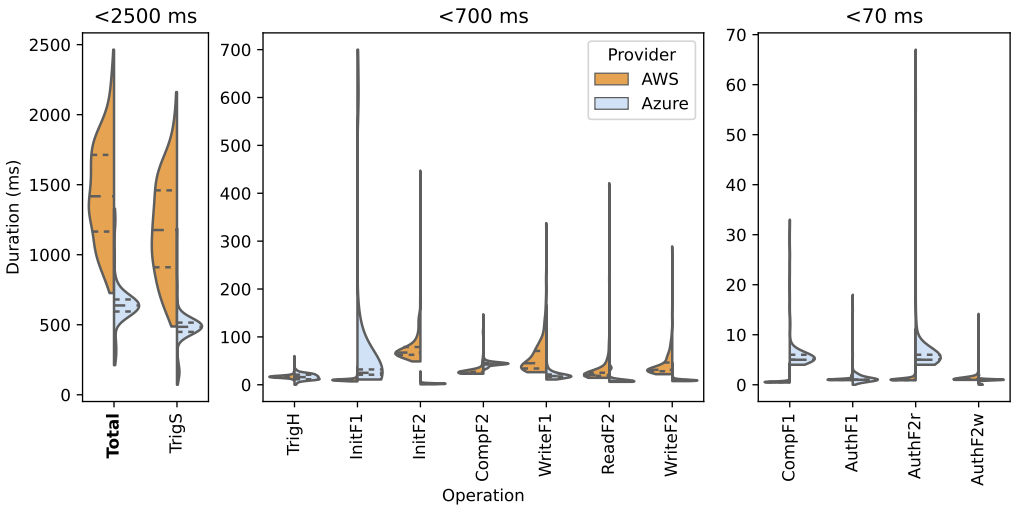

Figure 7. Detailed latency breakdown of the thumbnail generator application for warm invocations using the constant baseline workload. The horizontal middle dash denotes the median and the outer dashes denote quartiles.

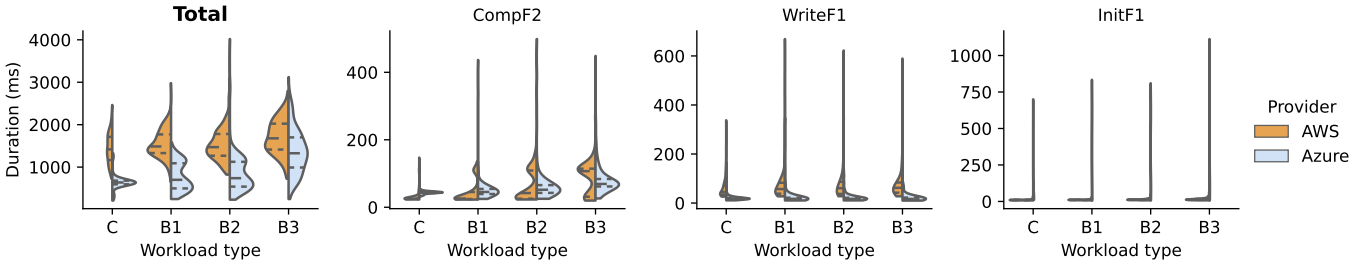

Figure 8. Comparison of workload types for *constant* (C) and *bursty* (B1-B3) workloads. Warm invocations only.

function instances whenever needed to avoid queueing at the function instance. Therefore, the results for *InitF2* follow the same behavior as discussed previously: Azure delivers much more predictable performance than AWS and higher burst rates increase performance variability.

Key Findings:

- The function scheduling policy of a provider is the main factor affecting performance under different workload types.
- Bursty workloads primarily cause more performance variability (i.e., flatter and bi-modal distributions)
- Bursty workloads can also degrade median latency, especially for compute-heavy operations.
- AWS copes better with bursty workloads than Azure.

V. Discussion

We now discuss results and challenges related to detailed tracing, serverless scalability, fair comparison, and threats to validity.

A. Importance of Detailed Tracing

Our results demonstrate that a detailed latency breakdown is instrumental for deriving actionable insights. We show that a single aspect (e.g., storage triggering) can dominate the end-to-end latency. It is important to identify such dominating aspects to guide performance optimization efforts. Further, aggregating heterogeneous aspects could mask performance issues. For example, the *total* duration in Figure 7 suggests a normal distribution but masks extreme outliers in the heavily skewed *InitF1* operation. Finally, detailed traces are essential for differentiating provider capabilities for different operation types such as computation, storage I/O, and orchestration overhead. For example, the *total* duration might suggest that Azure clearly outperforms AWS but the latency breakdown attributes this observation primarily to storage triggering.

Our study helps to interpret the many micro-benchmarking studies in the context of serverless applications. Traditional performance characteristics such as CPU speed are primarily studied in academic literature [7, 8, 6]. We demonstrate that other aspects such as trigger time and storage I/O can be more relevant than computation for a wide range of applications that are not compute-heavy. In particular, some asynchronous trigger implementations might be unsuitable for interactive applications.

B. Scalability Implications of Serverless

We now discuss scalability implications of serverless solutions from different cloud providers related to our results. The serverless solutions from AWS and Azure follow different paradigms. AWS purposefully designed massive multi-tenant services such as Firecracker [14] for serverless functions or Amazon S3 [52] and DynamoDB [53] for serverless storage. Azure also offers dedicated storage solutions under the umbrella of Azure Storage [54] (e.g., blob for files) but appears to operate services with stronger per-tenant isolation of the underlying hardware. For example, our results show that Azure Blob storage performs clearly more predictable in a presumably more isolated environment compared to Amazon S3. Amazon S3 experiences much worse tail latency, which is common for massive multi-tenant systems subject to noisy neighbors [55]. Another example is the in-process function scheduling in Azure, which is only possible with co-scheduling functions on the same host isolated from other tenants. We previously discussed performance implications of such a scheduling strategy in Sections IV-B and IV-C, including positive (e.g., more predictable performance), negative (queueing delay due to function reuse), and trade-offs (e.g., caching effect in multi-function workflows). Elasticity remains a major limitation of using more heavyweight virtual machines (VMs) for serverless functions in Azure [15] compared to purposefully built microVMs based on Firecracker in AWS [14]. This limitation was also observed by other studies [56, 32] but our results on function initialization overhead (*InitF1*) enable direct explanation through root cause analysis. To mitigate some of these challenges, Azure plans to phase out the in-process design to decrease host affinity according to their roadmap [57].

C. Fairly Comparing Cloud Providers

Portability is a key challenge for fair performance comparison across heterogeneous cloud providers. In traditional system-level benchmarks (e.g., SPEC CPU 2000), portability across operating systems was mainly dependent on the programming language of an application [58]. High-level languages such as Python or Java solve most of the traditional portability challenges [59] but vendor lock-in in serverless raises new challenges at a higher level of abstraction (e.g., provider-managed service) [60]. It is no longer possible to implement a single provider-agnostic benchmark because there exists no common interface. Therefore, cross-provider support requires careful application migration [28].

Our fairness design (see Section III-C) demonstrates that it is sometimes impossible or undesired to choose identical options within a heterogeneous environment. When mapping application components to provider-specific services, an identical option is often unavailable but there typically exists a different implementation of the same service type that can offer an equivalent high-level functionality (e.g., object storage, database storage). If multiple substitute services exist (e.g., message queues), parity in functionality and pricing can be used to select the most related service. When selecting comparable configurations, a presumably identical option could lead to an unfair competitive advantage if it contradicts provider recommendations. Hence, rather than insisting on literally identical options, the goal towards fair cross-provider compatibility should be to compare the recommended configuration for each provider (e.g., most mature). This configuration selection requires high technical expertise for the experiment design and should ideally involve an independent committee [18]. These challenges highlight the importance of documenting and motivating fairness design decisions.

Comparable instrumentation is crucial for measuring equivalent segments of work but comes with several portability challenges. A provider-agnostic trace model (e.g., Section III-D) is a compromise based on varying instrumentation capabilities (e.g., more fine-grained) but it can only incorporate mappable timestamps. Further, the exact timing of certain timestamps within the provider-internal infrastructure is sometimes not well-defined and requires clarification through provider support.

D. Threats to Validity

This section discusses the threats to validity following common categories in empirical research including construct validity, internal validity, and external validity [61, 62, 19].

1) Construct Validity: Construct validity refers to the extent to which the benchmark measures what it is supposed to measure. We alleviate this threat by thoroughly discussing the design of the instrumentation (Section III-D) and fairness (Section III-C). Unlike other benchmarks that solely report overall response times, our detailed latency breakdown based on carefully selected trace points enables explainable benchmarking of fine-grained components. Additionally, we capture client-side response times and correlate them with backend traces for end-to-end request validation.

2) Internal Validity: Internal validity refers to the extent to which the cause-and-effect relation is trustworthy and cannot be explained by other confounding variables. This threat is generally a primary concern in cloud benchmarking [63] and especially challenging for serverless because its underlying infrastructure is abstracted away, appearing almost like a black box for end users. We reviewed many existing academic and industrial serverless studies from systematic literature reviews [8, 7, 6] to identify known factors and motivate fair configuration in Section III-C. We acknowledge that some confounding factors are out of our control when studying inherently multi-tenant large-scale production services. To mitigate this threat, we quantify the cumulative effect of performance variability by visualizing performance distributions as violin plots.

3) External Validity: External validity refers to the extent to which the results from the study can be generalized, and the conclusion can hold throughout the study domain [64]. The two major threats to external validity in the context of this study are how the insights generated from comparing serverless application performance in AWS and Azure can be applied to other cloud platforms and serverless applications. This study focuses on two leading cloud providers (e.g., AWS and Azure) representing over $80\,\%$ of the serverless

applications according to multiple studies [13]. Serverless performance results are not generalizable beyond the studied cloud providers [8, 7, 39, 32, 45]. The results within a cloud provider are generalizable to a certain extent for similar duration categories under comparable workload (e.g., file size, computation task). Our approach presented in Section III is transferrable to other applications and providers that support distributed tracing.

VI. Related Work

We discuss our contribution in the context of the popular field of research on serverless benchmarking and highlight the gaps we address in serverless application benchmarking.

A. Serverless Benchmarking

Performance benchmarking is the most active field of research in serverless [6] with hundreds of studies published since its inception in 2016 [7, 8]. Prior studies primarily focus on serverless functions using micro-benchmarks to evaluate the CPU performance of a single function [7]. Many studies try to reverse-engineer serverless function platforms to characterize aspects such as tenant isolation [33], instance lifetime [34], network and disk I/O [65], cold start overhead [66], language performance [39, 43], processor architecture [67], or elasticity [56, 32]. In contrast to these micro-benchmarks, this paper uses a realistic serverless application to evaluate end-to-end latency and provide actionable insights through detailed latency breakdown analysis. This better reflects real-world serverless applications, which typically consist of multiple functions integrated with external services such as object storage, queues, or messaging services. Throughout the paper, we interpret the results from such micro-benchmarking studies in the context of real applications.

B. Serverless Application Benchmarking

Serverless applications that represent real-world characteristics such as multi-function workflows or external service integration are rare in the literature. Some studies started to explore function chaining [68] and the effects of different function coordination mechanisms [16] including asynchronous function chains [69]. There is more work on evaluating serverless workflow services [70, 71, 72, 73, 71, 74] such as AWS StepFunction or Azure Durable Functions. However, most studies use empty or artificial functions that are not representative of real-world applications.

Only a few studies deploy non-trivial applications that integrate external services such as object storage or databases. PanOpticon [68] compares function execution times of a chat application between AWS and Google. ServerlessBench [75] presents a course-grained latency breakdown of four applications on the open-source platform OpenWhisk. Most similar to our work, BeFaaS [76] compares the latency of an e-commerce application between the three providers AWS, Azure, and Google. BeFaaS categorizes latency into three categories: computation, network transmission, and database service time. BeFaaS mentions fairness as a benchmark requirement but lacks a comprehensive discussion on how it should be addressed. In contrast to prior work, our approach leverages more fine-grained tracing for detailed latency breakdown analysis across asynchronous call boundaries of different function triggers. We believe to provide the most extensive guidelines of fairness aspects for cross-provider serverless benchmarking yet.

VII. Conclusion

In this paper, we designed and implemented *CrossFit*, an application benchmark targeting AWS and Azure with a focus on fair cross-provider comparison and fine-grained performance analysis. We identified 12 fairness aspects that highlight the importance of configuration options and demonstrate that identical configuration can in some cases lead to an unfair competitive advantage. Our provider-agnostic trace model enables detailed cross-provider tracing of applications under different workloads including a constant baseline and three bursty workloads with different levels of burstiness. The results of our case study demonstrate that other aspects than function execution contribute to the majority of cross-provider performance differences. In particular, asynchronous storage triggering dominates the end-to-end latency. We further show that bursty workloads cause more performance variability but the effects vary for different services and providers.

In conclusion, our study emphasizes the importance of fine-grained performance analysis on an application level to enable deriving actionable insights. Practitioners should carefully choose appropriate function triggers for latency-sensitive applications. For researchers, we offer a reference to design fair cross-provider application benchmarks. Finally, we synthesize insights on serverless performance by relating the results of many micro-benchmarking studies to serverless applications.

Future work can extend the scope of this approach to other applications, cloud providers, and workloads. Additional applications can cover other external services (e.g., database), triggers (e.g., queues), and language runtimes (e.g., Python, Node.js, Java). Other major cloud providers, such as Google and Alibaba, offer promising function tracing capabilities and many other providers (e.g., Oracle, IBM) are working on improving their function tracing capabilities. Our approach supports new workloads through powerful and well-documented K6 scenarios[19] to study aspects such as cold starts [32], function instance lifetime [15], and elasticity [56] for applications.

As serverless tracing matures, we envision that future application benchmarking becomes easier, more portable, and more fine-grained. Automated dependency tracking could facilitate tracing by replacing our intrusive manual instrumentation. Recent advancements in tracing standardization through OpenTelemetry could make instrumentation more portable across providers in future studies. Providers could expose more fine-grained trace points (e.g., through AWS Lambda Extensions) to optimize cold starts for different language runtimes. Finally, longitudinal performance studies can keep the results updated as serverless systems evolve.

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

References

[1] 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.

[2] 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.

[3] "Serverless architecture market to exceed $30 bn by 2027," Global Market Insights Inc, 2021. [Online]. Available: https://www.globenewswire.com/news-release/2021/06/16/2247916/0/en/Serverless-Architecture-Market-to-exceed-30-Bn-by-2027-Global-Market-Insights-Inc.html

[4] "Serverless computing market - growth, trends, covid-19 impact, and forecasts (2022 - 2027)," Mordor Intelligence, 2021. [Online]. Available: https://www.mordorintelligence.com/industry-reports/serverless-computing-market

[5] 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.

[6] 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.

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

[8] 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.

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

[10] J. Brutlag, "Speed matters for google web search," Google, Tech. Rep., 2009. [Online]. Available: https://ai.googleblog.com/2009/06/speed-matters.html

[11] R. Kohavi and R. Longbotham, "Online experiments: Lessons learned," *Computer*, vol. 40, no. 9, pp. 103–05, Sep. 2007.

[12] 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.

[13] 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.

[14] 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.

[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] 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.

[17] 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.

[18] J. von Kistowski, J. A. Arnold, K. Huppler, K. Lange, J. L. Henning, and P. Cao, "How to build a benchmark," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPC)*, L. K. John, C. U. Smith, K. Sachs, and C. M. Lladó, Eds. ACM, Feb. 2015, pp. 333–336.

[19] W. Hasselbring, "Benchmarking as empirical standard in software engineering research," in *Evaluation and Assessment in Software Engineering (EASE)*, R. Chitchyan, J. Li, B. Weber, and T. Yue, Eds. ACM, 2021, pp. 365–372.

[20] K. Huppler, "The art of building a good benchmark," in *Performance Evaluation and Benchmarking, 1st TPC Technology Conference*, ser. Lecture Notes in Computer Science, vol. 5895. Springer, 2009, pp. 18–30.

[21] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A mixed-method empirical study of function-as-a-service software development in industrial practice," *Journal of Systems and Software (JSS)*, vol. 149, pp. 340–359, Mar. 2019.

[22] E. V. Eyk, "Function composition in a serverless world," May 2018. [Online]. Available: https://fission.io/blog/function-composition-in-a-serverless-world-video/

[23] R. R. Sambasivan, R. Fonseca, I. Shafer, and G. R. Ganger, "So, you want to trace your distributed system? key design insights from years of practical experience," Parallel Data Laboratory, Carnegie Mellon University, Tech. Rep. CMU-PDL-14-102, April 2014. [Online]. Available: https://www.pdl.cmu.edu/PDL-FTP/SelfStar/CMU-PDL-14-102.pdf

[24] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing frame-

work," in *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, H. Balakrishnan and P. Druschel, Eds., 2007.

[25] 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, Inc., Tech. Rep., April 2010. [Online]. Available: https://research.google/pubs/pub36356/

[26] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Nov. 2020, pp. 805–825.

[27] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy your observability: an industrial survey of microservice tracing and analysis," *Empirical Software Engineering (EMSE)*, vol. 27, no. 1, p. 25, 2021.

[28] V. Yussupov, U. Breitenbücher, F. Leymann, and C. Müller, "Facing the unplanned migration of serverless applications: A study on portability problems, solutions, and dead ends," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, ser. UCC'19. ACM, 2019, pp. 273–283.

[29] L. Zhu, G. Giotis, V. Tountopoulos, and G. Casale, "RDOF: deployment optimization for function as a service," in *14th IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2021, pp. 508–514.

[30] A. Mahéo, P. Sutra, and T. Tarrant, "The serverless shell," in *Proceedings of the 22nd International Middleware Conference: Industrial Track*, 2021, pp. 9–15.

[31] I. Google, "Compare aws and azure services to google cloud," 2021. [Online]. Available: https://cloud.google.com/free/docs/aws-azure-gcp-service-comparison

[32] 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.

[33] 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.

[34] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski, "Performance evaluation of heterogeneous cloud functions," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 23, Aug. 2018.

[35] 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.

[36] AWS, "Configuring lambda function options," 2021. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html

[37] A. Casalboni, "Aws lambda power tuning," 2019. [Online]. Available: https://github.com/alexcasalboni/aws-lambda-power-tuning

[38] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, "Sizeless: Predicting the optimal size of serverless functions," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 248–259.

[39] 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.

[40] "Managing lambda provisioned concurrency," AWS, 2022. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html

[41] "Azure functions premium plan," Microsoft Azure. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-premium-plan

[42] "Azure blob storage trigger for azure functions," Microsoft Azure. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob-trigger

[43] 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.

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

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

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

[47] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[48] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. von Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tuma, and A. Iosup, "Methodological principles for reproducible performance evaluation in cloud computing," *IEEE Transactions on Software Engineering (TSE)*, pp. 93–94, 2019.

[49] "How to configure monitoring for azure functions," Microsoft Azure, 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/configure-monitoring?tabs=v2#configure-scale-controller-logs

[50] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *USENIX Annual Technical Conference (ATC)*, 2021, pp. 443–457.

[51] M. Hüttermann, *Infrastructure as Code*, ser. DevOps for

Developers. Apress, 2012.

[52] T. Killalea, "A second conversation with werner vogels: The amazon cto sits with tom killalea to discuss designing for evolution at scale." *Queue*, vol. 18, no. 5, pp. 67–92, oct 2020.

[53] S. Sivasubramanian, "Amazon dynamodb: a seamlessly scalable non-relational database service," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 729–730.

[54] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. ul Haq, M. I. ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2011, pp. 143–157.

[55] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding long tails in the cloud," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, vol. 13, 2013, pp. 329–341.

[56] J. Kuhlenkamp, S. Werner, M. C. Borges, D. Ernst, and D. Wenzel, "Benchmarking elasticity of FaaS platforms as a foundation for objective-driven design of serverless applications," in *Proceedings of the 35th ACM/SIGAPP Symposium on Applied Computing (SAC)*. ACM, Mar. 2020, pp. 1576–1585.

[57] A. Chu, ".NET on Azure Functions roadmap," Microsoft, Tech. Rep., 2021. [Online]. Available: https://techcommunity.microsoft.com/t5/apps-on-azure-blog/net-on-azure-functions-roadmap/ba-p/2197916

[58] J. L. Henning, "SPEC CPU2000: measuring CPU performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[59] J. Hugunin, "Python and java: The best of both worlds," in *Proceedings of the 6th international Python conference*, vol. 9, 1997, pp. 2–18.

[60] D. Petcu, "Portability and interoperability between clouds: Challenges and case study - (invited paper)," in *Towards a Service-Based Internet - 4th European Conference on a Service-Based Internet (ServiceWave)*, vol. 6994. Springer, 2011, pp. 62–74.

[61] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 8, pp. 721–734, 2002.

[62] S. Easterbrook, J. Singer, M. D. Storey, and D. E. Damian, "Selecting empirical methods for software engineering research," in *Guide to Advanced Empirical Software Engineering*. Springer, 2008, pp. 285–311.

[63] C. Laaber, J. Scheuner, and P. Leitner, "Software microbenchmarking in the cloud. How bad is it really?" *Empirical Software Engineering (EMSE)*, vol. 24, no. 4, pp. 2469–2508, Apr. 2019.

[64] H. K. Wright, M. Kim, and D. E. Perry, "Validity concerns in software engineering research," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER)*, 2010, pp. 411–414.

[65] 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.

[66] 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.

[67] D. Xie, Y. Hu, and L. Qin, "An evaluation of serverless computing on x86 and arm platforms: Performance and design implications," in *IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021, pp. 313–321.

[68] N. Somu, N. Daw, U. Bellur, and P. Kulkarni, "Panopticon: A comprehensive benchmarking tool for serverless applications," in *Proceedings of the International Conference on COMmunication Systems NETworkS (COMSNETS)*, Jan. 2020, pp. 144–51.

[69] D. Balla, M. Maliosz, and C. Simon, "Performance evaluation of asynchronous faas," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021, pp. 147–156.

[70] J. Wen and Y. Liu, "A measurement study on serverless workflow services," in *2021 IEEE International Conference on Web Services (ICWS)*, 2021, pp. 741–750.

[71] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 356–370.

[72] P. G. López, M. Sánchez-Artigas, G. París, D. B. Pons, Á. R. Ollobarren, and D. A. Pinto, "Comparison of FaaS orchestration systems," in *Companion of the 11th IEEE/ACM UCC: 4th International Workshop on Serverless Computing (WoSC)*, Dec. 2018, pp. 148–53.

[73] Q. Jiang, Y. C. Lee, and A. Y. Zomaya, "Serverless execution of scientific workflows," in *Proceedings of the 15th International Conference on Service-Oriented Computing (ICSOC)*, ser. Lecture Notes in Computer Science, vol. 10601. Springer, Oct. 2017, pp. 706–721.

[74] E. van Eyk, "The design, productization, and evaluation of a serverless workflow-management system," Master's thesis, TU Delft, Jun. 2019.

[75] 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 *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2020.

[76] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, and D. Bermbach, "Befaas: An application-centric benchmarking framework for faas platforms," in *IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2021, pp. 1–8.