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 [1, 2] 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 fully-managed high-level service with fine-grained billing. 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 fast-growing serverless market[1] and serverless also remains a hot research topic with hundreds of published papers since 2016, as summarized in multiple literature reviews [3, 4, 5, 6].

Application performance is an important decision criterion for practitioners who choose a suitable cloud provider for their application. It matters to the end-user experience [7, 8] and many performance challenges have been reported for serverless platforms [6, 9, 3, 5]. Further, operational costs are directly connected to performance given the pay-per-use pricing model, which is typically based on execution duration and resource consumption.

[1]https://www.mordorintelligence.com/industry-reports/serverless-computing-market

Prior work focused on simplistic micro-benchmarks and incomplete response time metrics [3, 5] but does not capture the end-to-end latency characteristics of realistic serverless applications, which often integrate other cloud services [10, 11, 12]. The typical response time metric of synchronously invoked serverless functions does not represent event-based serverless architectures, which often use asynchronous triggers to coordinate complex workflows [13, 10, 12, 14].

Although fairness is a key characteristic of a benchmark [15, 16, 17], serverless benchmarking lacks a transparent approach to address fairness. Following guidance on how to build a benchmark, "fairness ensures that systems can compete on their merits without artificial constraints" [15]. A 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) [17]. In heterogeneous serverless environments, fairness is particularly challenging to address because the underlying infrastructure is abstracted away with vendor-specific implementations.

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.We focus on fairness by thoroughly designing and discussing the application architecture, a provider-independent tracing model, and workload generation for cross-provider comparison of a serverless application. We motivate the choice of application based on real-world characteristics [10, 12, 18, 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. We release the *CrossFit* benchmarking software, data, and results as a replication package [19] to foster future research. The results of our case study (Section IV) show that storage triggering dominates the end-to-end latency and explains the main performance difference between two leading cloud providers, AWS and Azure. Finally, we discuss our contributions (Section V), relate them to prior research (Section VI), and outline challenges for future work (Section VII).

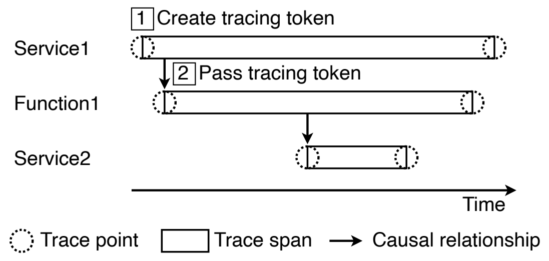

Figure 1. Simplified causal-time trace diagram of a synchronous invocation.

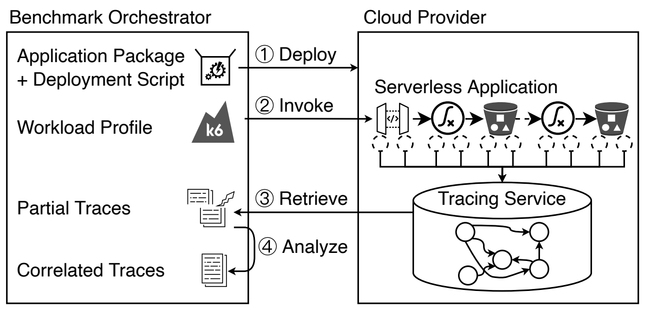

Figure 2. High-level overview of benchmarking approach.

II. Background

A. Serverless Computing

In serverless computing [1, 2], applications are composed of fully-managed services and often connected through FaaS. Serverless functions execute in ephemeral environments, where cloud providers manage and scale the underlying infrastructure. A one-off initialization called *coldstart* 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 but cloud providers recycle idle function instances after some time. 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 [13, 20].

B. Distributed Tracing

Distributed tracing [21, 22, 23] 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 from *Service1* over *Function1* into *Service2*. *Service1* receives an incoming request and generates a unique tracing token $\boxed{1}$ for each request. This tracing token is then used to label each timestamp captured at trace points of interest and needs to be passed $\boxed{2}$ 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.

III. Benchmark Design

Figure 2 shows the high-level architecture of our application benchmarking approach. First ①, a serverless application (Section III-A) is deployed into a cloud provider using an automated deployment script. For fair cross-provider comparison (Section III-B), an application needs to be configured appropriately. The application is instrumented with detailed trace points (Section III-C) and forwards trace spans to a provider-specific tracing service. Second ②, a workload profile (Section III-D) is applied to invoke the application deployed within a cloud provider. Third ③, the benchmark orchestrator retrieves partial traces from the tracing service and correlates ④ them to build a dataset for performance analysis (Section IV).

A. Application Design

We select an application called *thumbnail generator* that was previously used for exploring cross-provider migration [24], performance prediction [25], and performance evaluation [26]. This application has representative, real-world characteristics [10, 12, 18]. Instead of isolated function computation, our application integrates a common external service (i.e., cloud storage) in an idiomatic way for serverless through asynchronous triggering rather than traditional synchronous request/response-interaction. Figure 3 visualizes the architecture of the thumbnail generator consisting of two chained functions connected through an asynchronous storage bucket trigger. Table II explains the lifecycle of the application in detail.

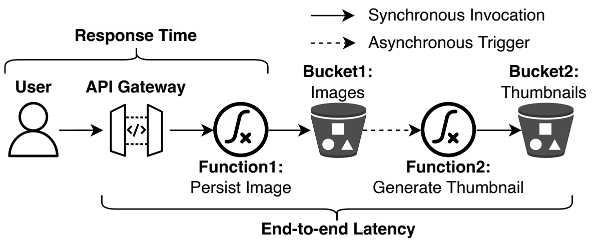

Figure 3. Architecture of the thumbnail generator application.

B. 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,

Table I
CROSS-PROVIDER SERVICE MAPPING FOR THUMBNAIL GENERATOR

	AWS	**Azure**
API Gateway	AWS API Gateway	Azure API Management
Function	AWS Lambda	Azure Functions
Object Storage	AWS S3	Azure Blob Storage

similar to prior work on cloud migration [24]. 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:

a) Application Architecture: **It is essential to maintain an equivalent architecture across cloud providers.** Prior work indicated that architecture-retaining migration is possible even for applications that are not optimized for portability [24].

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.** Table I shows the mapping for the thumbnail generator based on prior work [24]. For other services, we refer to an extensive list of cross-provider mappings[2].

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. [27]).** 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 [28, 29, 30], 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[3], 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[4], or runtime prediction such as Sizeless [31].

d) Cost: **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[5]. 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). Performance differs by runtime, especially for coldstart overhead [32]. 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 using programming language constructs (e.g., 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. Quantitative code metrics are of limited use for achieving accurate trace model parity.

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 it performs very inconsistently [32].

h) Function Pre-warming: **Premium coldstart 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 coldstarts through tracing instead of preventing them through intrusive options such as provisioned concurrency[6] in AWS or the Azure Functions Premium plan[7].

[2]https://cloud.google.com/free/docs/aws-azure-gcp-service-comparison

[3]https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html

[4]https://github.com/alexcasalboni/aws-lambda-power-tuning

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

[6]https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html

[7]https://docs.microsoft.com/en-us/azure/azure-functions/functions-premium-plan

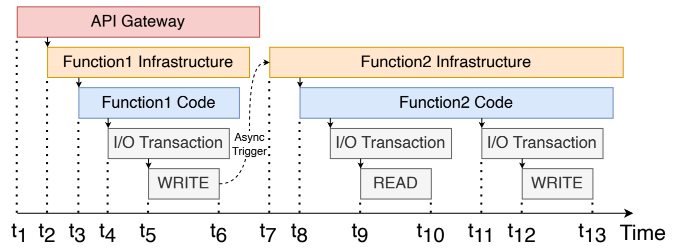

Figure 4. Trace design for thumbnail generator application.

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. To avoid up to 10-minute coldstart delays, Azure recommends[8] using the EventGrid[9] trigger.

j) Data Center Location: **We suggest choosing established data centers in geographically close regions.** We choose the AWS region *us-east-1* and the Azure region *eastus* as commonly used by other serverless studies [28, 33, 30, 34, 35].

k) Metrics: **We focus on fine-grained latency metrics because latency is typically more relevant than bandwidth [36], especially in massive scale-out cloud environments.** Average performance is only useful for summarizing cost [37] but currently over-used in serverless [3] and cloud [38] performance studies. Instead, we visualize the full performance distribution as violin plots and focus on typical performance (50th percentile), tail latency (e.g., 95th percentile), and stability (standard deviation). We derive the fine-grained latency metrics from the instrumentation design (Section III-C).

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

C. Instrumentation Design

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

Figure 4 visualizes 13 interesting timestamps (explained in Table II) along the critical path for the thumbnail generator application (depicted in Figure 3). 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.

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.

D. 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 [12] 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 [12]. Additionally, FaaS workloads exhibit highly dynamic bursty workload patterns as confirmed by production workloads from Azure [12] and Alibaba [39] as well as by a characterization study of serverless applications from diverse cloud providers [10]. To represent these higher invocation rates and bursty workload characteristics, we designed three bursty workloads (`B1`-`B3`). 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.

E. Implementation

We automate *CrossFit* to mitigate reproducibility challenges in serverless experimentation [3] using our benchmarking orchestrator ServiBench [40] (see Figure 2). We extend ServiBench for Azure including trace correlation. We port the thumbnail generator application based on ServiBench [40] to .NET and Azure and automate its deployment using Terraform[10]. Using the tracing SDKs, we instrument the application with AWS X-Ray[11] and Azure Application Insights[12]. The workloads from Section III-D are provided as K6[13]

[8]https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob-trigger

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

[10]https://www.terraform.io/

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

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

[13]https://k6.io/

Table II
SUMMARY OF TIMESTAMPS AND OPERATIONS BETWEEN TIMESTAMPS FOLLOWING THE TRACE DESIGN IN FIGURE 4

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}		

configurations reusable across applications and cloud providers. All code, configuration, and data is documented and available in our replication package [19].

IV. CASE STUDY

We first summarize the experiment setup before presenting the results for the latency breakdown of the thumbnail generator under the different workloads from Section III-D.

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 coldstarts because they exhibit fundamentally different performance characteristics and need to be studied separately.

B. Latency Breakdown

The violin plot in Figure 5 visualizes the latency breakdown of the thumbnail generator application comparing the providers AWS and Azure. We group the operations into three 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 for both providers introducing unpredictable delays of $(485.0 \pm 117.0)\,\mathrm{ms}$ (median±standard deviation) for Azure and $(1176.0 \pm 355.0)\,\mathrm{ms}$ 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 higher variability as the standard deviation shows: $(17.0 \pm 3.8)\,\mathrm{ms}$ for AWS vs. $(16.0 \pm 6.2)\,\mathrm{ms}$ for Azure. *InitF1* of the lightweight *Function1* shows that AWS exhibits consistently less initialization overhead than Azure, which suffers from extreme outliers going beyond 500 ms. In contrast, the heavyweight *Function2* with larger code libraries initializes almost instantly in Azure compared to much slower initialization in AWS $(67.0 \pm 31.0)\,\mathrm{ms}$. This caching effect in Azure is caused by their in-process function scheduling[14], 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-B). Larger memory configurations with multi-core virtual CPUs are available for AWS at a 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.

Key Findings:

- Asynchronous storage triggering dominates the end-to-end latency of the thumbnail generator application.
- The Azure EventGrid-based storage trigger has lower and more predictable latency than the AWS S3 trigger.
- The AWS HTTP trigger latency is more predictable compared to Azure.
- Azure storage operations (read/write) are faster and more predictable than in AWS under low load.

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

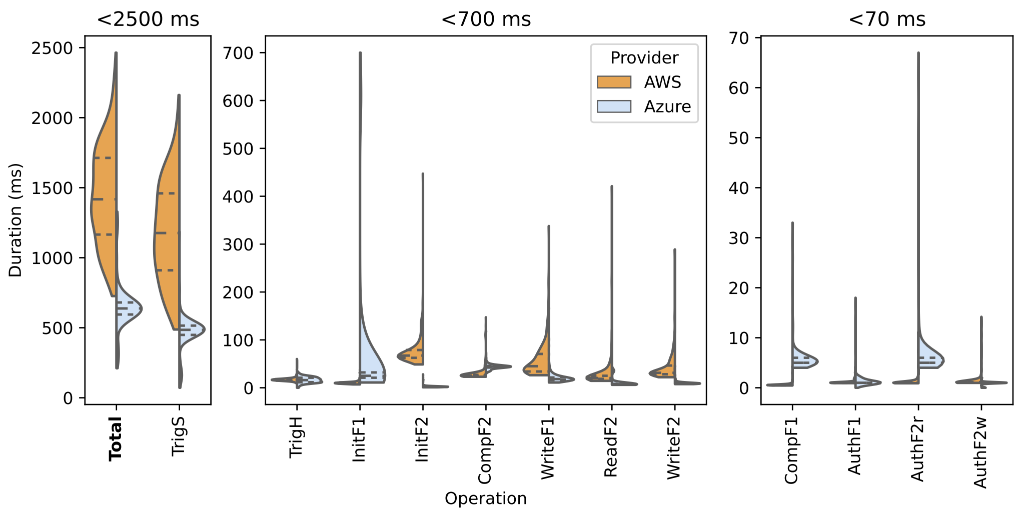

Figure 5. Detailed latency breakdown of the thumbnail generator application for warm invocations using the constant baseline workload. The horizontal middle dash denotes the median and the outer dashes denote quartiles.

C. Workload Types

Figure 6 compares the baseline constant workload (`C`) to bursty workloads with different burst sizes (`B1`-`B3`). The *total* duration 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 long tail latency 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 more tail latency observed for Azure as well.

Storage I/O operations (e.g., *WriteF1*) suffer from very long tail latency in the constant baseline, especially in AWS. Bursty workloads might deteriorate the situation but additional burst rates should be tested. While the other write operation (*WriteF2*) follows a similar pattern, the read operation (*ReadF2*) is more predictable in AWS than in Azure.

The initialization of *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 coldstarts but introduces massive queueing delays. In contrast, AWS delivers predictable warm initialization performance at the cost of more frequent coldstarts because their scheduling policy provisions new 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 workloads.
- 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 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 5 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 coordination 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 [3, 5, 4] in the context of serverless applications. 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. In particular, some asynchronous trigger implementations might be unsuitable for interactive applications.

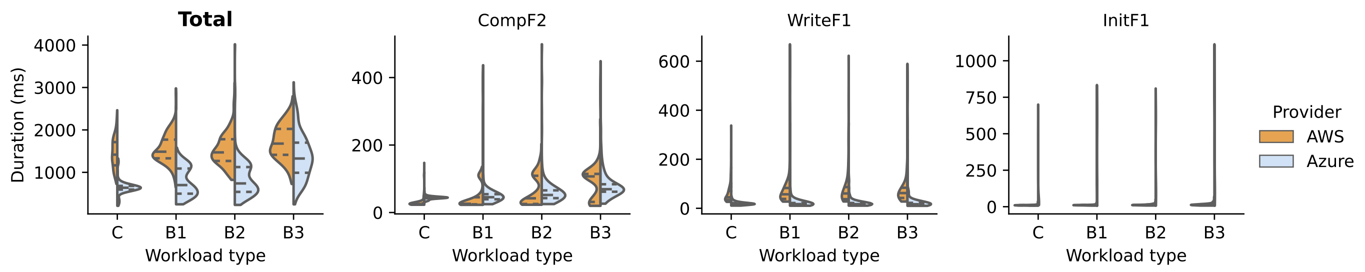

Figure 6. Comparison of workload types for *constant* (C) and *bursty* (B1-B3) workloads. Warm invocations only.

B. Scalability Implications of Serverless

The serverless solutions from AWS and Azure follow different paradigms with implications for scalability. AWS purposefully designed massive multi-tenant services such as Firecracker [11] for serverless functions or Amazon S3 [41] and DynamoDB [42] for serverless storage. Azure also offers dedicated storage solutions under the umbrella of Azure Storage [43] (e.g., blob for files) but appears to operate services with stronger per-tenant isolation of the underlying hardware. For example, Azure Blob storage performs more predictable in a presumably more isolated environment compared to Amazon S3. Amazon S3 experiences worse tail latency, which is common for multi-tenant systems subject to noisy neighbors [44]. 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 [12] compared to purposefully built microVMs based on Firecracker in AWS [11]. This limitation was also observed by other studies [45, 27] 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 [46].

C. Fairly Comparing Cloud Providers

Portability is a key challenge for fair performance comparison across heterogeneous cloud providers in serverless computing. Vendor lock-in makes it impossible to implement a single provider-agnostic benchmark because there exists no common interface. Therefore, cross-provider support requires careful application migration [24].

Our fairness design (see Section III-B) 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 offers similar 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). Configuration selection requires high technical expertise for the experiment design and should ideally involve an independent committee [15].

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-C) 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

1) Construct Validity: We alleviate threats to construct validity by thoroughly discussing the design of the instrumentation (Section III-C) and fairness (Section III-B). Unlike other benchmarks that solely report overall response times, our 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 is a primary concern in cloud benchmarking [47] and is 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 literature reviews [5, 3, 4] to identify known factors and motivate fair configuration in Section III-B. 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: This study focuses on two leading cloud providers (i.e., AWS and Azure) representing over 80 % of the serverless applications [10, Fig. 17]. Serverless performance results are not generalizable beyond the studied cloud providers [5, 3, 32, 27, 35]. The results within a cloud provider are generalizable to a certain extent for similar operations under comparable workloads (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 within the context of serverless benchmarking and related application benchmarks.

A. Serverless Benchmarking

Performance benchmarking is the most active field of research in serverless [4] with hundreds of studies published since its inception in 2016 [3, 5]. Prior studies primarily focus on serverless functions using micro-benchmarks to evaluate the CPU performance of a single function [3]. Many studies try to reverse-engineer serverless function platforms to characterize factors [48] such as tenant isolation [28], instance lifetime [29], network and disk I/O [49], coldstart overhead [50], language performance [32, 33], processor architecture [51], or elasticity [45, 27]. In contrast to these micro-benchmarks, we use a realistic application to evaluate end-to-end latency and provide actionable insights through 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.

B. Serverless Application Benchmarking

Serverless applications that represent real-world characteristics such as multi-function workflows or external service integration are rare in prior work. Some studies started to explore function chaining [52] and the effects of different function coordination mechanisms [13] including asynchronous function chains [53]. There is more work on evaluating serverless workflow services [54, 55, 56, 57, 55] 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 [52] compares function execution times of a chat application between AWS and Google. ServerlessBench [58] presents a course-grained latency breakdown of four applications on the open-source platform OpenWhisk. Most similar to our work, BeFaaS [59] 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, 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

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 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). 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[15] to study aspects such as coldstarts [27], function instance lifetime [12], and elasticity [45] 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 coldstarts for different language runtimes. Finally, longitudinal performance studies can keep the results updated as serverless systems evolve.

Acknowledgment

We are grateful to the SPEC-RG Cloud research group for insightful discussions and contributions to the benchmark orchestration tool. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

[15]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 *Proc. of the 2nd International Workshop on Serverless Computing (WOSC)*. ACM, 2017, pp. 1–4.

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

[4] 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)*. ACM, 2019, pp. 229–240.

[5] 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.

[6] 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.

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

[8] R. Kohavi and R. Longbotham, “Online experiments: Lessons learned,” *Computer*, vol. 40, pp. 103–05, 2007.

[9] 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 ESEC/FSE*, 2021.

[10] 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.

[11] 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)*, 2020, pp. 419–434.

[12] 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 *USENIX Annu. Tech. Conf. (ATC)*, 2020.

[13] S. Quinn, R. Cordingly, and W. Lloyd, “Implications of alternative serverless application control flow methods,” in *Proc. of 7th Int. Workshop on Serverless Comp.*, 2021.

[14] P. G. López, A. Arjona, J. Sampé, A. Slominski, and L. Villard, “Triggerflow: trigger-based orchestration of serverless workflows,” in *the 14th ACM Int. Conf. on Distributed and Event-based Systems (DEBS)*, 2020.

[15] J. von Kistowski, J. A. Arnold, K. Huppler, K. Lange, J. L. Henning, and P. Cao, “How to build a benchmark,” in *Proc. of the 6th ACM/SPEC Int. Conf. on Performance Engineering (ICPC)*. ACM, 2015, pp. 333–336.

[16] W. Hasselbring, “Benchmarking as empirical standard in software engineering research,” in *Evaluation and Assessment in Software Eng. (EASE)*, 2021, pp. 365–372.

[17] K. Huppler, “The art of building a good benchmark,” in *Perf. Eval. and Benchmarking*, ser. Lecture Notes in Computer Science, vol. 5895. Springer, 2009, pp. 18–30.

[18] 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, 2019.

[19] J. Scheuner, R. Deng, J.-P. Steghöfer, and P. Leitner, “Serverless CrossFit: Replication package for application benchmark,” doi:10.5281/zenodo.7267118, 2022.

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

[21] 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,” 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

[22] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, “X-trace: A pervasive network tracing framework,” in *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[23] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, “Dapper, a large-scale distributed systems tracing infrastructure,” Google, Tech. Rep., 2010. [Online]. Available: https://research.google/pubs/pub36356/

[24] 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 *Proc. of the 12th IEEE/ACM International Conf. on Utility and Cloud Comp. (UCC)*, 2019.

[25] L. Zhu, G. Giotis, V. Tountopoulos, and G. Casale, “RDOF: deployment optimization for function as a service,” in *14th IEEE Int. Conf. on Cloud Comp. (CLOUD)*, 2021.

[26] A. Mahéo, P. Sutra, and T. Tarrant, “The serverless shell,” in *Proc. of 22nd Int. Middleware Conf.: Industry*, 2021.

[27] D. Ustiugov, T. Amariucai, and B. Grot, “Analyzing tail latency in serverless clouds with stellar,” in *IEEE Int. Symp. on Workload Characterization (IISWC)*, 2021.

[28] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, “Peeking behind the curtains of serverless platforms,” in *Proc. of the USENIX Annu. Tech. Conf. (ATC)*, 2018.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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 Int. Conf. on Cloud and Big Data (CBDCom)*. IEEE, 2020, pp. 704–711.

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

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

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

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

[38] 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 Trans. on Software Engineering (TSE)*, 2019.

[39] 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.

[40] J. Scheuner, S. Eismann, S. Talluri, E. van Eyk, C. Abad, P. Leitner, and A. Iosup, "Let's trace it: Fine-grained serverless benchmarking using synchronous and asynchronous orchestrated applications," doi:10.48550/ARXIV.2205.07696, 2022.

[41] 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.

[42] S. Sivasubramanian, "Amazon DynamoDB: a seamlessly scalable non-relational database service," in *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 2012.

[43] B. Calder *et al.*, "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.

[44] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding long tails in the cloud," in *10th USENIX Symp. on Networked Systems Design and Impl. (NSDI)*, 2013.

[45] 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 *Proc. of the 35th ACM/SIGAPP Symp. on Applied Computing (SAC)*, 2020, pp. 1576–1585.

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

[47] C. Laaber, J. Scheuner, and P. Leitner, "Software microbenchmarking in the cloud. How bad is it really?" *Empirical Software Engineering (EMSE)*, vol. 24, 2019.

[48] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, Apr. 2018, pp. 159–169.

[49] H. Lee, K. Satyam, and G. C. Fox, "Evaluation of production serverless computing environments," in *3rd Int. Workshop on Serverless Computing (WoSC)*, 2018.

[50] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *4th Int. Workshop on Serverless Computing (WoSC)*, 2018.

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

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

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

[54] 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.

[55] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Middleware*, 2020, pp. 356–370.

[56] P. G. López, M. Sánchez-Artigas, G. París, D. B. Pons, Á. R. Ollobarren, and D. A. Pinto, "Comparison of FaaS orchestration systems," in *4th Int. Workshop on Serverless Computing (WoSC)*, Dec. 2018, pp. 148–53.

[57] Q. Jiang, Y. C. Lee, and A. Y. Zomaya, "Serverless execution of scientific workflows," in *Conf. on Service-Oriented Computing (ICSOC)*, 2017, pp. 706–721.

[58] 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.

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