Let's trace it: Fine-grained serverless benchmarking for synchronous and asynchronous applications

Downloaded from: https://research.chalmers.se, 2026-02-08 23:07 UTC

Citation for the original published paper (version of record):

Scheuner, J., Eismann, S., Talluri, S. et al (2026). Let's trace it: Fine-grained serverless benchmarking for synchronous and asynchronous
applications. Future Generation Computer Systems, 179.
http://dx.doi.org/10.1016/j.future.2025.108336

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Future Generation Computer Systems 179 (2026) 108336

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Let's trace it: Fine-grained serverless benchmarking for synchronous and asynchronous applications

Joel Scheuner [a], Simon Eismann [b], Sacheendra Talluri [c,*], Erwin van Eyk [c], Cristina Abad [d], Philipp Leitner [a], Alexandru Iosup [c]

[a] Chalmers | University of Gothenburg, Gothenburg, Sweden
[b] University of Würzburg, Würzburg, Germany
[c] Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
[d] Escuela Superior Politécnica del Litoral, Guayaquil, Ecuador

ARTICLE INFO

Keywords:
Serverless
Application benchmark
Distributed tracing
FaaS

ABSTRACT

Making serverless computing widely applicable requires detailed understanding of performance. Although benchmarking approaches exist, their insights are coarse-grained and typically insufficient for (root cause) analysis of realistic serverless applications, which often consist of asynchronously coordinated functions and services. Addressing this gap, we design and implement ServiTrace, an approach for fine-grained distributed trace analysis and an application-level benchmarking suite for diverse serverless-application architectures. ServiTrace (i) analyzes distributed serverless traces using a novel algorithm and heuristics for extracting a detailed *latency breakdown*, (ii) leverages a suite of serverless applications representative of production usage, including *synchronous and asynchronous serverless applications* with external service integrations, and (iii) automates comprehensive, *end-to-end experiments* to capture application-level performance. Using our ServiTrace reference implementation, we conduct a large-scale empirical performance study in the market-leading AWS environment, collecting over 7.5 million execution traces. We make four main observations enabled by our *latency breakdown analysis* of median latency, cold starts, and tail latency for different application types and invocation patterns. For example, the median end-to-end latency of serverless applications is often dominated not by function computation but by external service calls, orchestration, and trigger-based coordination; all of which could be hidden without ServiTrace-like benchmarking. We release empirical data under FAIR principles and ServiTrace as a tested, extensible, open-source tool at https://github.com/ServiTrace/ReplicationPackage.

1. Introduction

Emerging in the late 2010s from the integration of multiple technological breakthroughs [1], *serverless computing* [2–4] aims to abstract away operational concerns (e.g., autoscaling) from the developer by providing fully managed cloud platforms through self-serving application programming interfaces (APIs). Developers can leverage a rich *ecosystem of external services* (e.g., message queues, databases, image recognition), which are glued together by a Function-as-a-Service (FaaS) platform, such as AWS Lambda, through synchronous and especially *asynchronous invocations*. For the current generation of serverless platforms, ease of management comes with important performance trade-offs and issues, including high tail-latency and performance variability [5,6], and delays introduced by asynchronous use of external services [7,8]. Although understanding these performance trade-offs is essential, existing performance benchmarks, measurement frameworks, and distributed tracing approaches, do not explicitly consider asynchronous invocations and external services. Addressing this gap, we propose *ServiTrace*, a novel approach for fine-grained distributed trace analysis and an application-level benchmarking suite designed based on real-world characteristics, including asynchronous invocations and external services.

The need to understand and compare the performance of serverless platforms has received much attention, leading to many useful results. Extensive prior work proposes empirical performance evaluation of FaaS platforms [9–16]. However, there currently is no serverless benchmark that provides *white-box*, detailed analysis of realistic applications, which combine multiple functions and external services that are event-driven and asynchronously coordinated. Existing benchmarks are able to measure synchronous response times, but are unable to *explain* the end-to-end performance of complex applications. Particularly, these

* Corresponding author.
E-mail addresses: joel.scheuner@localstack.cloud (J. Scheuner), eismannsimon@gmail.com (S. Eismann), sacheendra.t@gmail.com (S. Talluri), erwinvaneyk@gmail.com (E. van Eyk), cristina.abad@gmail.com (C. Abad), philipp.leitner@chalmers.se (P. Leitner), alexandru.iosup@gmail.com (A. Iosup).

https://doi.org/10.1016/j.future.2025.108336
Received 24 January 2025; Received in revised form 28 May 2025; Accepted 21 December 2025
Available online 28 December 2025
0167-739X/© 2025 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

approaches do not explain *how* the components of a full serverless solution impact end-to-end performance. This may render root-cause analysis impossible, and limit opportunities for performance improvement and comparison.

Although distributed tracing [17,18] is essential for thoroughly understanding serverless performance, serverless architectures with asynchronous invocations are insufficiently supported, especially for analysis. Inspired by Google's Dapper [19] and popularized through open-source projects such as Zipkin,[1] Jaeger,[2] and OpenTelemetry,[3] distributed tracing requires a level of cooperation with the platform that remains unavailable to serverless applications. Various approaches exist for this broad class of problems that assume full observability and accurate timing, and they are useful for *synchronous* microservice architectures [20]. In contrast, many serverless applications use multiple functions and external services, and invocations often occur *asynchronously*.

We focus on fine-grained serverless benchmarking for synchronous and *asynchronous* applications, addressing the gap in supporting asynchronous invocations. Asynchronous triggers are the only way to execute serverless functions in response to a change in a data store like S3. S3 is used in 68% of systems architectures provided by AWS, and 25% of architectures contain serverless functions that read from S3 [21]. Asynchronous triggers are also the only way to efficiently and dynamically trigger other functions [22]. HTTP requests require the caller to keep running, increasing the cost. Services like Step Functions require users to specify all possible requests in advance. We propose a novel approach for serverless application trace analysis, which can extract detailed latency information even when asynchronous invocations occur, and does not assume tight cooperation between the serverless platform and the serverless application. Our approach builds on distributed tracing infrastructure made available by state-of-the-art cloud providers (e.g., AWS X-Ray,[4]) but additionally provides an end-to-end view of application performance that is not directly available from the raw traces. Furthermore, we design, implement, and experimentally use ServiTrace, a benchmarking suite that leverages our novel tracing approach to offer application-level serverless benchmarking.

Our detailed trace analysis helps users better understand the performance results of their experiments. For example, ServiTrace exposes where cold-start times come from, revealing that cold-start times are dominated by overhead in language runtimes compared to runtime initialization; platform engineers can use such information to guide their optimizations and debloat language runtimes. Application developers can evaluate competing architectures by quantifying the tradeoff between connection overhead and tail latency for different persistency services, an overhead that becomes apparent not from end-to-end performance numbers, but by our detailed trace analysis.

Overall, our contribution in this paper is four-fold:

1. **Latency breakdown analysis** (Section 4): We design a novel algorithm and heuristics for distributed trace analysis for serverless architectures. The key capability over prior work is that our approach is applicable in a serverless context, across asynchronous call boundaries and external services. Our *detailed* latency breakdown handles implicit transitions (e.g., caused by asynchronous triggers or observability gaps) and accounts for every millisecond along the critical path by identifying and classifying each time segment. Our trace analyzer implementation for AWS X-Ray is directly applicable to production serverless traces.
2. **Principled design of an application-level benchmarking suite** (Sections 3 and 4): Starting from five design principles, including three that are serverless-specific, we design the comprehensive *ServiTrace benchmarking suite*, containing 10 realistic open-source applications that cover different forms of orchestration, synchronous and asynchronous triggers, and real-world characteristics such as programming language, application size, and external services. ServiTrace orchestrates reproducible deployments, automates trace-based load generation, collects distributed traces, and provides detailed white-box analysis.
3. **Empirical performance study** (Section 6): As a demonstration of usefulness, and to foster the research community's overall understanding of serverless application performance, we conduct a comprehensive white-box analysis of serverless application performance in the market-leading AWS environment. Our results cover, e.g., cold starts, tail latency, and the impact of application type and invocation patterns on performance. This study is enabled by and only made possible through the latency breakdown approach presented in this paper.
4. **FAIR release** of the ServiTrace software, data, and results at https://github.com/ServiTrace/ReplicationPackage: We release ServiTrace on Github, and the configurations and full data (~ 70GB) on Zenodo. The replication package follows the FAIR principles ("findable, accessible, interoperable, and reusable" [23]), which we consider imperative in science and engineering.

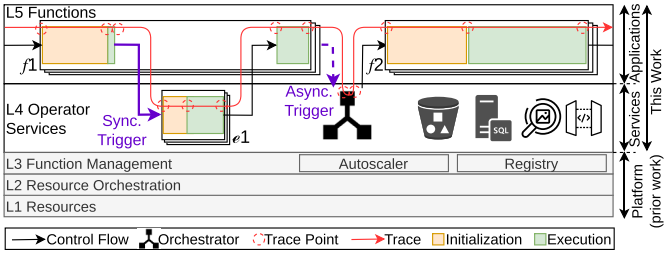


Fig. 1. System model of a serverless application composed of multiple functions: ≤1 and ≤2 are user-defined functions, and ≡1 is an external service. This work focuses on application-level benchmarking.

The remainder of the paper is organized as follows: Section 2 summarizes a system model for serverless applications to clarify the context and scope of our work. Section 3 gives a high-level overview of our approach, which consists of the trace analysis in Section 4 and ServiTrace application-level benchmarking suite in Section 5. In Section 6, we use ServiTrace to demonstrate our trace-based benchmarking approach with a detailed latency breakdown analysis of median latency, cold starts, and tail latency for different application types and invocation patterns in AWS. Finally, we discuss related work in Section 7 and present our conclusions in Section 8.

2. System model for serverless applications

Our work assumes a system model, introduced by the SPEC Research Group, to represent the operation of tens of existing serverless platforms [24]. In this model, the serverless stack provides resources (Label L1 in Fig. 1) whose use is orchestrated (L2) enabling complex function management (L3), such as auto-scaling and image-registry. These elements have been covered extensively by the community [25,26] and are not in the scope of this study.

The system model further includes two layers that directly service the application and are the focus of this work. Each application is composed of a single or, more commonly, multiple user-defined functions (L5) that can asynchronously trigger external (operator-provided) services (L4). Fig. 1 depicts an example of a serverless application. There are two user-defined functions (≤1 and ≤2). During its execution, ≤1 synchronously triggers an operator-provided service, ≡1.

[1] https://zipkin.io/
[2] https://www.jaegertracing.io/
[3] https://opentelemetry.io/
[4] https://aws.amazon.com/xray/

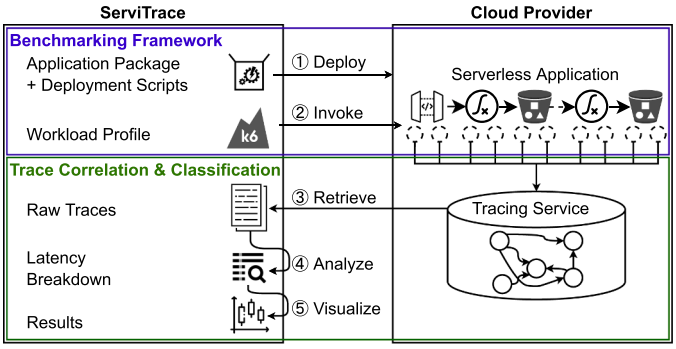


Fig. 2. ServiTrace high-level design.

At the end of its execution, ≤1 asynchronously triggers an operator-provided orchestrator service.

Production-level serverless applications fit this model well. The functions can communicate with external services (≡1)—and with each other—directly via synchronous or asynchronous triggers. They can also communicate via an orchestrator, which decides on the control flow based on user-provided instructions. Using a combination of these methods, we can construct complex execution paths using any of the operator-provided services, e.g., object stores, databases, and ML services.

We assume there is a *monitoring service* that provides detailed performance information and, in particular, traces the upper layers in the serverless stack (i.e., L4 and L5). We do *not* assume that the other layers are observable by the application; e.g., there is no server-side tracing. Although we assume the presence of *distributed tracing*, we do *not* assume its results are consistent and ordered across multiple components in the system or that the application triggers can only occur synchronously. These assumptions match the operation of common serverless platforms, such as AWS, Azure, and Google.

3. Principled design for fine-grained serverless benchmarking

We introduce here the design principles and the high-level design of ServiTrace. Sections 4 and 5 detail the latency breakdown analysis and the construction of the benchmarking suite, respectively.

3.1. Design principles

We formulate five design principles by adapting to serverless best practices on benchmarking [27,28] and by extending ideas from the microservice benchmark DeathStarBench [6]:

1. **End-to-end operation:** An application-level serverless benchmark should implement end-to-end functionality starting from an incoming client request, following through individual functions, across external services, and into different composition methods. It is unlikely to capture with a single number the performance of all these components, so instead, benchmarks should collect fine-grained measurements across individual components. To obtain such measurements, benchmarks should implement realistic serverless applications and instrument them using distributed tracing, and further compute end-to-end details.
2. **Asynchronous applications:** A representative and relevant benchmark suite closely matches the characteristics of real-world applications, e.g., applications ending with a synchronous response, or after a chain of asynchronous event-based function triggers. For this work, we select applications from industrial workshops and academic studies based on the most common serverless application types [2,29], programming languages [29,30], application sizes [25,29,30], and external services [25,29,30].
3. **Heterogeneity:** Benchmarks should strive for *generality* by covering a wide span of applications in the serverless design space—a *heterogeneous* benchmark suite, which includes diverse applications across multiple design dimensions. The suite should be able to execute workloads based on real execution traces (i.e., trace-based) so that the applications can be realistically evaluated.
4. **Reproducibility:** A reproducible benchmark suite mitigates threats to internal validity that could affect the ability to obtain the same results with the same method under *changed conditions* of measurements [31]. We provide automated containerized benchmark orchestration for all applications including their configurations and pinned dependencies.
5. **Extensibility:** The variety of existing serverless applications introduces the need for *extensibility*—benchmarks should allow adding existing serverless applications written in common programming languages, using common frameworks, or cloud service dependencies with no or only minor code changes. We give evidence of the extensibility of our plugin-based benchmark by integrating diverse existing applications, with diverse structures.

3.2. High-level design

Fig. 2 depicts the high-level design of the trace-based serverless application benchmarking with ServiTrace. Our tracing approach (green box, Section 4) enables fine-grained analysis of serverless applications, which are provided and orchestrated by the serverless application benchmarking framework (blue box, Section 5). Step ①, a serverless application of our benchmark suite (Section 5.1) is deployed into a cloud provider using automated deployment scripts. Step ②, a workload profile (Section 5.2) invokes application-specific scenarios with different load levels. Step ③, ServiTrace retrieves raw traces from the provider-specific tracing service, which collects traces from the instrumented applications. Step ④, ServiTrace analyzes the raw traces by extracting a detailed latency breakdown along the critical path of potentially asynchronous invocations. Step ⑤, our replication package visualizes the results, as evidenced in Section 6.

4. Distributed trace analysis for serverless architectures

A key contribution of this work is our novel approach for distributed trace analysis of serverless applications. In this section, we motivate the need for a new approach and describe how to extract the critical path and a *detailed* latency breakdown from distributed traces.

4.1. Challenges and background

Distributed tracing, which records how a distributed application operates over time, has been adopted for various use cases [17,18] such as distributed profiling (i.e., latency analysis), anomaly detection (i.e., identifying and debugging rare problems), and workload modeling (e.g., identifying representative workflows). Tracing systems such as Google's Dapper [19] or Facebook's Canopy [32] help improve performance, correctness, understanding, and testing.

The ephemeral and distributed nature of serverless architectures makes their tracing challenging. Because the commonly used event-based coordination is inherently *asynchronous*, hard-to-track background workflows need to be included in tracing data and cannot be ignored as for synchronous microservice architectures [20]. Limited control in serverless environments makes users dependent on provider tracing implementations or forces them to resort to less detailed third-party or custom implementations. Further, tracing issues are common at large scale, and trace analysis must detect and handle clock inaccuracy and incomplete traces.

We introduce here the terms needed to address distributed tracing for serverless applications. To simplify understanding these abstract, time-related concepts, we illustrate them in Fig. 3. We represent each server-

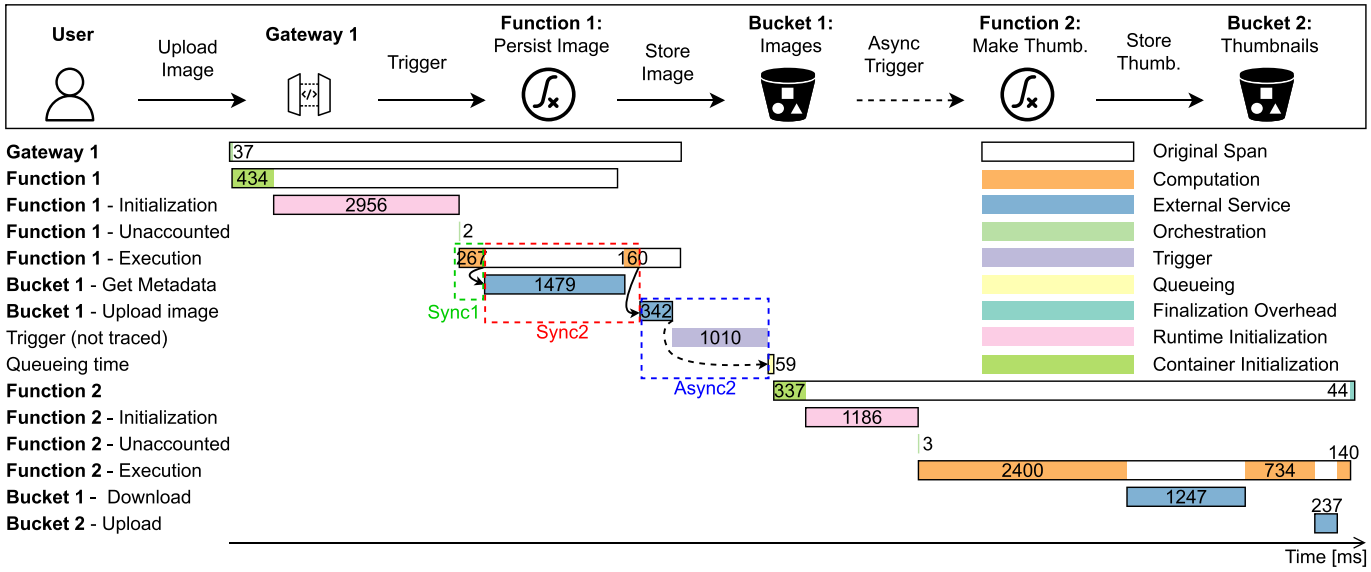


Fig. 3. Simplified depiction of an *execution trace* (Definition 1) with annotated *latency breakdown* (Definition 3) from App-B with two cold starts. Values represent time in milliseconds. Labels Sync/Async refer to Fig. 4.

less application, from when it is triggered by an incoming request to its completion, as an *execution trace*, where the *critical path* determines the end-to-end latency, and the *latency breakdown* lists and classifies each time span along the critical path, as in Fig. 3.

Definition 1. An *execution trace* of a serverless application is a causal-time diagram of the distributed execution of a request, where a node is a *trace span* that corresponds to an individual unit of work (e.g., computation) and an edge represents a causal relationship through a synchronous or asynchronous invocation. Each trace span contains a start and end timestamp and is correlated by a trace identifier. Fig. 3 illustrates a simplified execution trace of a serverless application (App-B) with synchronous and asynchronous coordination.

Definition 2. A *critical path* in an execution trace is the longest path weighted by duration, which starts with a client request and ends with the trace span that has the latest end time. This definition of end-to-end latency includes asynchronous background workflows that do not return to their parent spans to capture the event-based nature of serverless systems, as for *Async2* in Fig. 3. Hence, our definition differs from a critical path of a synchronous client response in microservices [20].

Definition 3. A *latency breakdown* of an execution trace is the most detailed list of time segments along the critical path without any temporal gaps—see the highlighted segments, across multiple units of work, in Fig. 3. This explicitly includes transitions between trace spans, which are often implicit in an execution trace and thus not or incorrectly recorded by current tools. We propose that each time segment can be classified[5] into the following categories common to serverless applications:

1. *Computation* represents the actual processing time of serverless functions.
2. *External service* represents the time spent waiting for the completion of a services request (e.g., database query, file upload to a storage service).
3. *Orchestration* represents the time spent coordinating serverless function executions by workflow engines (e.g., AWS Step Functions) or API gateways dispatching requests to functions.
4. *Trigger* represents the implicit transition time between an event and a function bound to this event (e.g., time between enqueueing a message until the event is dispatched to a function).
5. *Queueing* represents the time spent in function worker-queues before starting execution.
6. *Container initialization* represents the time it takes to provision the function execution environment.
7. *Runtime initialization* represents the time it takes to initialize the function language runtime during a cold start.
8. *Finalization overhead* represents cleanup tasks after function execution and before freezing the sandbox.
9. *Other* represents a catch-all timing class.

4.2. Latency breakdown extraction

We propose a novel algorithmic approach for latency breakdown extraction. The approach first extracts the *critical path* of an *execution trace* and then refines it into a detailed *latency breakdown*.

To extract the critical path, we use Algorithm 1, which is a modified version of the weighted longest-path algorithm proposed for microservices [20]. The algorithm iteratively builds the critical path from the start of a trace (i.e., span with earliest *startTime*) to its end (i.e., span with latest *endTime*) and uses recursion at Lines 11 and 13 to follow into child spans. Our heuristic $current.\textsc{happensBefore}(next)$ detects sequential relationships by checking for span overlapping ($current.endTime < next.startTime$) and $current.\textsc{isAsync}(next)$ detects asynchronous invocations by comparing end times ($next.endTime > current.endTime$), with a configurable error margin ϵ (default 1 ms[6]) to gracefully handle minor clock inaccuracies.

In comparison to previous work, our modifications

1. fix an ordering bug,
2. support asynchronous invocations, and
3. mitigate timing issues [33].

[5] We use high-level categories for readability and cross-application comparison because full trace-breakdowns are too detailed for high-level analysis. We posit that individual external services (such as cloud storage) can be classified separately.

[6] Suitable error margins for different datacenters can be parametrized using tools to quantify the clock error bound, for example ClockBound (https://github.com/aws/clock-bound).

We fix a bug that led to a wrong order in the critical path by moving the recursive call at Line 13 for the *lastChild* from *before* to *after* the for loop. We support asynchronous invocations through conditional recursion by only following child spans that are connected to the end of the trace. We mitigate common timing issues in fine-grained serverless traces due to different timestamp resolutions and clock shifts in large-scale distributed systems. Our sorting heuristic at Line 7 uses a secondary sort key to determine the order of a trace span with a duration of 0 milliseconds. For more implementation detail including line-by-line explanations and test cases, we refer to our replication package.

Algorithm 1 Critical path extraction supporting asynchronous invocations.

Require: Serverless execution trace T with *span* attributes $childSpans$, $startTime$, $endTime$ and
stack S with all parent spans from the end of the trace (i.e., span with the latest $endTime$).

1: **procedure** T.CRITICALPATH(S, $currentSpan$)
2: $path \leftarrow$ [currentSpan]
3: **if** S.top() == $currentSpan$ **then** ▷ update auxiliary stack
4: S.pop()
5: **if** $currentSpan.childSpans == None$ **then** ▷ base case of recursion
6: Return $path$
7: $sortedChildSpans \leftarrow$ sortAscending($currentSpan.childSpans$, by=[$endTime$, $startTime$])
8: $lastChild \leftarrow sortedChildSpans.last$ ▷ last returning child span
9: **for** each $child$ in $sortedChildSpans$ **do**
10: **if** ($currentSpan$.ISASYNC($lastChild$) **and** S.top() == $lastChild$) **or** ▷ async *child* case
(not $currentSpan$.ISASYNC($lastChild$) **and** $child$.HAPPENSBEFORE($lastChild$) **and not** $currentSpan$.ISASYNC($path$.last)) **then** ▷ sync *child* case
11: $path$.extend(CRITICALPATH($S, child$)) ▷ recursion into *child* span
12: **if** ($currentSpan$.ISASYNC($lastChild$) **and** S.top() == $lastChild$) **or** ▷ async *lastChild* case
(**not** $currentSpan$.ISASYNC($lastChild$) **and not** $currSpan$.ISASYNC($path$.last)) **then** ▷ sync *lastChild* case
13: $path$.extend(CRITICALPATH($S, lastChild$)) ▷ recursion into *lastChild* span
14: Return $path$

We extract the detailed latency breakdown along the critical path by identifying and categorizing every time segment while accounting for all gaps between spans. Fig. 4 visualizes the common cases for synchronous and asynchronous invocations that can occur while iterating pairwise (*current*, *next*) over the critical path. For synchronous invocations, we distinguish two different cases: *Sync1* handles a traditional synchronous invocation from a *current* parent span into a *next* child span. *Sync2* handles a potentially recursive transition from the *current* span on a synchronous invocation stack (*parent*→*middle*→*current*) across a common *parent* into its *next* child span. For asynchronous invocations, we distinguish two cases: *Async1*, if the *next* child span overlaps with the *current* parent span, and *Async2*, if there is a gap between the *current* parent span and the *next* child span, which occurs frequently in serverless systems when triggering a function using a slow trigger. There is a third case, which is structurally equivalent to *Sync1* except that the call to *next* is asynchronous; although we cannot currently detect this case, as discussed for Open Telemetry [34], trace specifications could define labels for synchronous and asynchronous parent-child relationships. Finally, the analyzer automatically assigns an activity label (e.g., computation) to each breakdown segment depending on the span type (e.g., function) as annotated in Fig. 3 using the provider-specific service mappings and metadata.

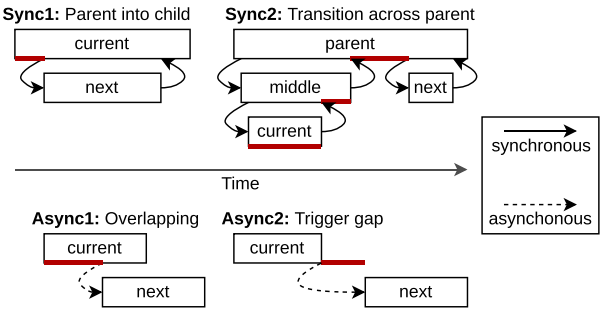


Fig. 4. Extraction cases of latency breakdown (red segments) for pairs of current and next nodes on the critical path. **Sync#** invocations are synchronous. **Async#** invocations are asynchronous.

5. ServiTrace benchmarking suite: Design, implementation, extensions

The ServiTrace application-level benchmarking suite provides 10 representative open-source applications to demonstrate the capabilities of our detailed trace analysis for serverless architectures, presented in the previous section. The suite includes workloads for all applications and supports various invocation scenarios, such as load generation based on production workloads. We implement ServiTrace as a Python library with Docker support, test all its applications on AWS, and outline its extensibility for other cloud providers.

5.1. Serverless applications

Table 1 characterizes the 10 serverless applications in the ServiTrace serverless application benchmarking suite, covering a wide span of realistic choices across multiple design dimensions; our replication package[7] describes and motivates each application. The suite covers each of the most common types identified by survey studies [29,35] except for the type of *operations and monitoring* because such applications are difficult to test in isolation. In particular, in the table, *API* applications use synchronously invoked web endpoints (e.g., REST, GraphQL), *async* processing applications are triggered through events (e.g., an upload to a storage bucket triggers a function), and *batch* are larger computation tasks often processed as concurrent functions.

We further cover three other dimensions in the serverless design space:

1. *programming language*, we select multiple applications for each of the dominant serverless programming languages JavaScript (JS) and Python [29,30], and one application for each of the popular enterprise languages Java, C#, and Go;
2. representative *size*, as datasets [25] and surveys [29,30] show that most applications are composed of 10 or fewer functions;
3. most popular *external services* used in serverless applications [25,29,30] with a focus on API gateways and persistency services (e.g., S3 cloud storage, DynamoDB cloud database), including both external services that are used for synchronous cloud orchestration (e.g., AWS Step Functions) and asynchronous function coordination through cloud pub/sub (e.g., SNS), cloud queue (e.g., SQS), and cloud streaming (e.g., Kinesis).

Apps B, D, and J are examples of applications whose results would apply to emerging workloads like LLM applications. In these apps, the serverless functions are used as a glue to combine external services to process multimedia data. Similarly, serverless functions can glue different components in an LLM application together.

[7] https://github.com/ServiTrace/ReplicationPackage

Table 1
Characteristics of each end-to-end serverless application. Abbreviations: Programming language (Lang), number of functions (# Fs), API gateway (◊), cloud storage (Γ), cloud DB (Θ), cloud orchestration (▲), cloud pub/sub (Δ), cloud queue (■), cloud streaming (Ω), cloud ML (O). Details[7].

	Application	Type	Lang	# Fs	External Services						
A	Min Baseline [36]	API	JS	1	◊						
B	Thumb Gen [37]	Async	Java	2	◊	Γ					
C	Event Proc [37]	Async	JS	8	◊		Θ		Δ	■	
D	Facial Recog [38]	Async	JS	6	◊	Γ		▲	Δ		O
E	Model Train [39]	Async	Python	1	◊	Γ					
F	Rw Backend [40]	API	JS	21	◊		Θ				
G	Hello Retail! [41]	API	JS	10	◊	Γ	Θ	▲		Ω	
H	To-do API [37]	API	Go	5	◊		Θ				
I	Matrix Mult [37]	Batch	C#	6	◊	Γ		▲			
J	Video Proc [39]	Batch	Python	1	◊	Γ					

5.2. Serverless workloads

In our benchmark design, workload profiles define how applications are invoked through *application scenarios*, and how their load changes over time through *invocation scenarios*.

5.2.1. Application scenarios

Application scenarios represent user stories and are provided by each application as a fully programmable workload script. We implement this integration through the open-source, network load-testing tool k6[8], and provide scenarios written in JavaScript for all applications.

Application scenarios demonstrate common testing techniques that can be used for extending ServiTrace, for example, we implement client-side tracing support to validate load generation. For multimedia applications (e.g., App-B, App-D, App-J), we support dynamic media collections, for example through randomized image generation. Some applications have business logic constraints (e.g., App-G, App-H), which we model through probabilistic state machines; for example, in App-H a to-do item must be created before it can be marked complete.

5.2.2. Invocation scenarios

Invocation scenarios model the intensity and shape of the generated load when invoking an application. Thorough performance evaluation requires different workloads [42] because no single workload can cover potentially conflicting design choices [27]. Therefore, ServiTrace supports three types of invocation scenarios:

1. Sequentially invoke the application a fixed number of times to support development and microbenchmarking,
2. programmable scenarios to control the number of user sessions, change of request arrival rate, and more through the k6 API, and
3. replay real traces such as from the Azure Functions [25] dataset, the most comprehensive invocation logs from serverless production systems publicly available ([44,45]).[9]

5.3. ServiTrace reference implementation

We implement the ServiTrace suite and trace analysis as a Python library that offers a CLI and SDK to orchestrate serverless application benchmarking. New applications can be integrated by providing a Python file with three lifecycle methods to prepare, invoke, and cleanup themselves. The benchmarking lifecycle (Fig. 2) is programmable through a CLI, as demonstrated by the following example that deploys an application, sequentially invokes it 1000 times, waits for 60 seconds, retrieves and analyzes the traces, and finally removes all its cloud resources: `sb prepare invoke 1000 wait 60 get_traces analyze_traces; sb cleanup`. Our Python SDK offers an equivalent API and is suitable for more advanced scenarios (used in Section 6) with dynamic re-configuration, re-deployment, error handling, and workload generation (Section 5.2). The suite supports Docker to package application-specific build and deployment dependencies, and automatically manages directory mounts and provider credentials. For deployment automation, most applications in the suite leverage the Serverless Framework but we have also tested other cross-provider infrastructure-as-code solutions such as Terraform and Pulumi.[10] For workload generation, our integration makes powerful k6 invocation scenarios[8] reusable across applications and cloud providers.

AWS Lambda uses NTP to synchronize clocks [46]. This does not guarantee time synchronization across domains. Therefore, we performed tests to quantify the clock drift. We find that in 99.99% of the cases, the drift is minimal. However, when clock drift occurs, it is 5-8x the baseline [47].

6. Experimental results with the ServiTrace reference implementation

In this section, we use ServiTrace to break down and analyze the latency of a popular serverless platform. To demonstrate our trace-based benchmarking approach, we deploy ServiTrace on the real-world, market-leading, serverless platform AWS Lambda, which various reports [29,48] indicate is popular for serverless applications used in production.

6.1. Experiment design

We design experiments that showcase how ServiTrace supports diverse real-world performance scenarios:

1. latency breakdown analysis to understand the performance of warm invocations and of cold starts, and on
2. the impact of invocation patterns on (median) end-to-end latency. As this study is enabled by and only made possible through ServiTrace, we can not provide a comparable baseline. In the following, we discuss our experiment design, our findings, and their implications for serverless practitioners and researchers.

We conduct a performance benchmarking experiment [28] with an open-loop load generator in the Northern Virginia (us-east-1) data center region, as commonly used by other serverless studies [9,49–52]. We collected over 7.5 million traces, through 12 months of experimentation in 2021 and 2022.

Application configuration. All functions are configured with the same memory size of 1024 MB as this provides a balanced cost-performance ratio [53] between the minimal memory size of 128 MB (heavy CPU throttling) and the maximum memory size for a single CPU core of 1769 MB [54] (inefficient for non-CPU-intensive load). For application-specific memory size tuning, we refer to *aws-lambda-power-tuning* [55], systematic literature reviews [15,16], and many empirical studies [9,11,53,56–59]. All supported cloud services (i.e., API Gateway, Lambda, Step Functions) are traced with ServiTrace. For applications with chained functions, we filter out "partial cold starts" and only consider "full" warm or cold invocations, where every function in the critical path shares the same cold start status. For applications with multiple

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

[9] FaaSNet [43] published a 24-hour trace from Alibaba Cloud Function Compute but it only includes cold starts, which makes it incomplete for load generation.

[10] Deployment automation: https://www.serverless.com/, https://www.terraform.io/, and https://www.pulumi.com/.

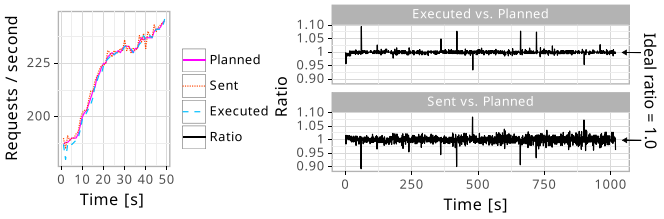


Fig. 5. Comparison of per-second invocation rates planned vs. sent vs. executed (left) and validation ratio for pairwise comparison (right).

endpoints, we present one representative endpoint in the paper and refer to the replication package for detailed results.

Load generator. For accurate load generation, we deploy an over-provisioned EC2 instance of the type *t3a.large* in the same region as the serverless applications. We validate per-second invocation rates for accurate load generation (planned vs. sent) by correlating the load configuration with the client logs and actual load serving (sent generated vs. executed) by correlating the client logs with the backend traces. We combine visual comparison (see Fig. 5) with FastDTW [60], an approximate Dynamic Time Warping (DTW) algorithm. We monitor application error rates client-side by checking response status codes and server-side by checking for any exceptions in each trace. Finally, we investigate any invalid traces due to incomplete or invalid trace data following both logical and time-based validation.

6.2. Latency breakdown

We first drill down into the end-to-end latency of serverless applications to identify critical components using ServiTrace (Section 5) and trace breakdown extraction (Section 4). This application-level perspective complements existing work, which primarily focused on micro-benchmarking individual components or reporting client-side response times for synchronously orchestrated applications [15,16]. As a baseline, we focus on warm invocations and subsequently compare the latency penalty of cold invocations and tail latency.

Method. For each of the 10 applications, we send 4 bursts of 20 concurrent requests with an inter-arrival time of 60 seconds between each burst. The first burst triggers up to 20 cold invocations used in Section 6.2.2 and after the function completes within 60 seconds, the following 3 bursts trigger more warm invocations used for tail-latency analysis in Section 6.2.3 and as baseline in Section 6.2.1. To collect sufficient samples under the same conditions, we conduct 10 trials and 14 repetitions resulting in up to 8400 warm invocations ($3 \times 20 \times 10 \times 14$)[11] For each of the 10 trials, we invoke each application using round-robin scheduling with inter-trial times of 50 minutes to trigger cold invocations in the first burst for Section 6.2.2. Before each of the 14 repetitions of trials, we re-deploy each application to ensure a clean state. We perform cold start filtering based on ServiTrace annotations, as described in Section 5.

6.2.1. Warm invocations

Serverless platforms can have optimizations to improve performance when functions are invoked more than once in a short period of time. It is key to consider this class of invocations since they are very common; [25] found that 99.6 % of all function invocations come from functions that are executed on average at least once a minute and should therefore contain a large share of warm starts.

[11] A related study [5] uses 3000 samples for individual functions; we target more per-application samples as requests can be distributed across endpoints.

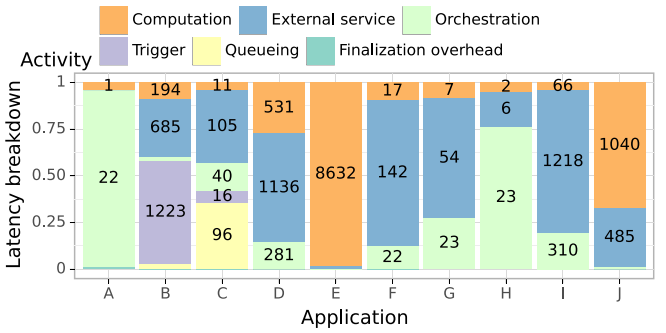


Fig. 6. Latency breakdown of warm invocations as median fraction of end-to-end latency. Values inside the bar-stacks represent absolute time per activity, in milliseconds.

> **Observation 1:** The median end-to-end latency of serverless applications is often dominated by external service calls and synchronous orchestration or asynchronous trigger-based coordination. The actual computation time in serverless functions is relatively low, except for inherently compute-heavy workloads.

Results. The relative latency breakdown in Fig. 6 shows the median latency for each activity introduced in Definition 3. App-A exemplifies the orchestration overhead (22 ms) of a common serverless pattern where an API gateway is connected to a function. Lightweight applications such as App-H are similarly dominated by *orchestration* time (23 ms) because they do minimal computation work and use fast external services (e.g., 6 ms database insert). App-E and App-J are examples of computation-heavy workloads. External services such as blob storage or computer vision APIs are often the dominating factor, especially for many I/O operations (App-I) or larger files (App-D). Asynchronous applications are typically dominated by transition delays due to trigger and queueing time as demonstrated by the applications App-B and App-C.

6.2.2. Cold starts

We now study which time categories contribute to higher cold start latency using the results from Section 6.2-1 as a baseline. Tracing cold starts requires access to timestamps captured within the provider-internal infrastructure. ServiTrace can extract these internal timestamps from their traces and distinguish between container and runtime initialization time, which would otherwise only be possible for self-hosted [13] or provider-internal [61,62] systems. Insights on cold starts are relevant for applications that are invoked irregularly (e.g., inter-arrival times >10 minutes) or exhibit bursty invocation patterns and, hence, need to provision new function instances.

Results. Fig. 7 shows the latency difference between the medians for cold invocations compared to warm invocations. App-A depicts a common initialization overhead for a function behind an API gateway of 265 ms (98 + 167) in line with prior cold start studies for Node.js by [9, Figure 6] and [63, Figure 7]. Our results are more detailed and reveal differences for realistic applications. Our trace details show that runtime initialization typically accounts for the majority of cold start overhead compared to container initialization. For App-A, the container initialization time of 98 ms is ~20 ms faster than the boot times for the underlying Micro VMs as reported for pre-configured Firecracker [61, Figure 6]. In comparison, other realistic applications have much higher absolute initialization times due to large packaged dependencies (e.g., App-E, App-J) and chains of multiple functions in the critical path (e.g., App-B, App-D).

Beyond runtime and container initialization, other categories can add cold start overhead that is often overlooked. Computation can

Table 2
Partial cold starts in applications.

App	Partial cold start fraction	Partial cold start latency [ms]	Warm start latency [ms]	Cold start latency [ms]
B	0.9%	9232	1934	15,789
C	0.04%	856	32	1550
D	0.3%	2770	1517	4779
I	0.8%	4279	1119	8134

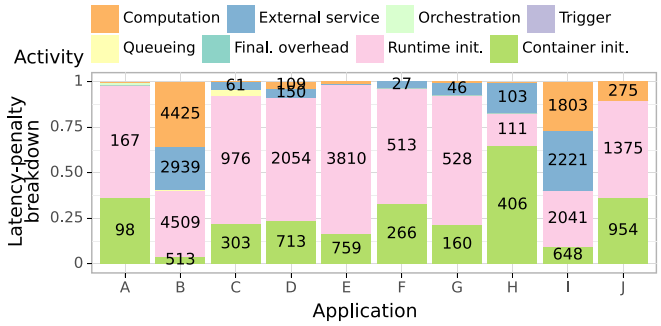


Fig. 7. Latency-penalty breakdown for cold invocations compared to the baseline of warm invocations (Fig. 6) as a fraction of the difference between the medians. Values inside the bar-stacks are absolute, in milliseconds (ms), e.g., for App-B, Computation takes 4425 ms longer compared to a warm invocation.

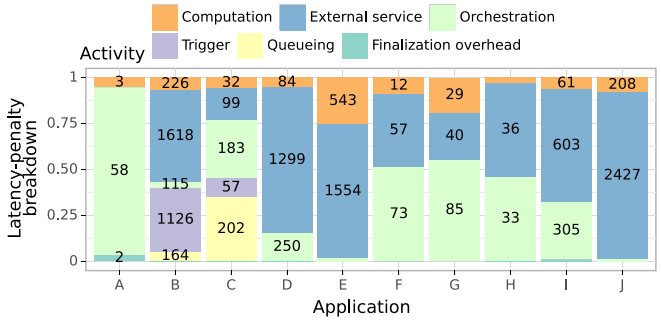


Fig. 8. Latency-penalty breakdown for slow invocations compared to baseline of warm invocations (Fig. 6) as a fraction of the difference between the median and 99th percentile. Values inside the bar-stacks are absolute, in milliseconds (ms), e.g., for App-J, the Cloud storage service adds 2427 ms of delay for slow invocations; this accounts for ~90 % of the tail-latency slowdown.

Observation 2: Runtime initialization and container initialization add the most overhead for cold invocations but external service connection initialization and one-off computation tasks can also contribute.

Observation 3: Tail latency is primarily caused by external services, particularly by object storage.

contain conditional code executed only upon cold starts or trigger one-off optimizations such as just-in-time compilation for interpreted languages [64] exemplified by the applications App-B in Java and App-I in C#. External service time can add connection overhead due to extra authentication upon cold starts (e.g., App-B caches S3 authentication) or database connection setup (e.g., App-H connects to DynamoDB). Finally, the following categories related to application coordination remain unaffected by cold starts: orchestration, trigger, queueing, and instrumentation overhead.

We observe partial cold starts in four of the ten applications we run. These occur due to some functions, but not all, in the function chain experience a cold start. We summarize the fraction of requests that experienced a partial cold start and compare their latency to warm and cold invocations in Table 2. The application with the highest fraction (0.9%) of partial cold starts is the thumbnail generator (B), despite only being composed of two functions. The second highest (0.8%) is matrix multiplication (I) which is composed of five functions and many parallel invocations. We hypothesize that the large fraction of partial cold starts for B is due to the long duration of the functions. Their long duration means that function instances are not freed frequently, increasing the likelihood of subsequent invocations being subjected to a cold start. The end-to-end latency of requests that experience a partial cold lies between the latencies experienced by warm and cold invocations.

6.2.3. Tail latency

Tail latency is increasingly important at scale for cloud providers [65] and hence particularly challenging for massive multi-tenant serverless systems. Prior studies [5,7,9,10] conducted micro-benchmarks to measure tail latency of individual serverless components. By leveraging our trace analysis (Section 4), we can directly identify which time categories contribute to tail latency (99th percentile) for entire applications.

Results. Fig. 8 shows that external services cause major variability. In particular, storing a large file (+2429 ms for App-J) causes massively more tail-latency delay than storing many chunks of small files (+602 ms for App-J). Database services contribute less to tail latency than object storage as demonstrated by the applications App-C, App-F, App-G, and App-H with latency penalties between 35 ms and 99 ms.

Another factor of tail latency is the serverless overhead for orchestrating synchronous applications (i.e., *orchestration* time) and asynchronous applications (i.e., *trigger* and *queueing* time). These categories double or triple their latency in comparison to the baseline in Fig. 6. Computation is inherently variable in a multi-tenant system but contributes at most 25 % to the latency penalty for compute-heavy App-E.

6.3. Invocation patterns

Real-world applications exhibit diverse invocation patterns [25], but prior work rarely investigated dynamic workloads over time [10] or different invocation patterns [14] and if so, using artificial applications, patterns, and one-time bursts [5,9,51]. It remains unclear how different invocation patterns derived from the Azure Function Traces [25] affect the end-to-end latency of serverless applications. To address this gap, we investigate the performance effect of varying invocation rates over time under an equivalent average invocation rate.

Scalability prestudy. We conducted a prestudy to adjust the average invocation rates to our 10 heterogeneous applications. Using the same invocation rate or concurrency level for all applications is inappropriate because it overloads some applications while others remain close

Table 3
Invocation rates (in reqs./s) per application (50 % of the achieved load in the scalability pre-study).

A	B	C	D	E	F	G	H	I	J
200	37	50	25	22	167	154	200	10	25

to idle[12] Therefore, we test increasing load levels with constant arrival rates for 90 seconds until an application exceeds a rate of 5 % for trace errors or invalid traces twice in succession. Trace-based root cause analysis identified rate limits and function tracing issues.[13] To avoid overloading an application, we select 50 % of the achieved load level as the target average invocation rate for parameterizing the invocation patterns (Table 3).

Method. We treat each application from Table 1 with two artificial and four realistic workloads derived from real-world traces as described below. The artificial workloads serve as a baseline for fully *constant* load and maximal burstiness simulated by *on_off* alternations with load for one second and idle time of three seconds. We scale the average invocation rate per-application following Table 3. We discard warmup measurements of the first 60 seconds as the actual invocation rate can deviate from the target rate in the first second and initial cold starts dominate the start of every experiment.

To derive invocation patterns from the Azure Function Traces [25], we selected 528 functions (out of 74 347 functions in the traces) with relevant properties for benchmarking. We removed 45 564 temporary functions not available over the entire two-week period and skip 15 319 timer triggers because these follow predictable periodic patterns [25] and are typically not latency-critical. Knowing that the 18.6 % most popular applications with invocation rates $\geq 1/min$ represent 99.6 % of all function invocations [25], we selected the 2.6 % most popular functions with average invocation rates $\geq 1/s$ as they are relevant for high-volume benchmarking.

We visually identified four typical invocation patterns by manually classifying two time ranges for 100 of the selected 528 functions. We first created 200 individual line plots with invocation counts over 20 minutes[14] and grouped similar traffic shapes into several clusters. After merging similar patterns, we identified four common patterns (see Fig. 9):

1. *steady* (32.5 %) represents stable load with low burstiness,
2. *fluctuating* (37.5 %) combines a steady base load with continuous load fluctuations especially characterized by short bursts,
3. *spikes* (22.5 %) represents occasional extreme load bursts with or without a steady base load, and
4. *jump* (7.5 %) represents sudden load changes maintained for several minutes before potentially returning to a steady base load.

Results. Fig. 10 shows the partial CDF of the end-to-end latency for applications that accurately followed the target invocation pattern (<10 % deviation from target invocation rate and error metrics). The median latency is unaffected by invocation patterns as shown by the overlapping CDF curves. Percentiles up to p99 clipped in the ECDF also show

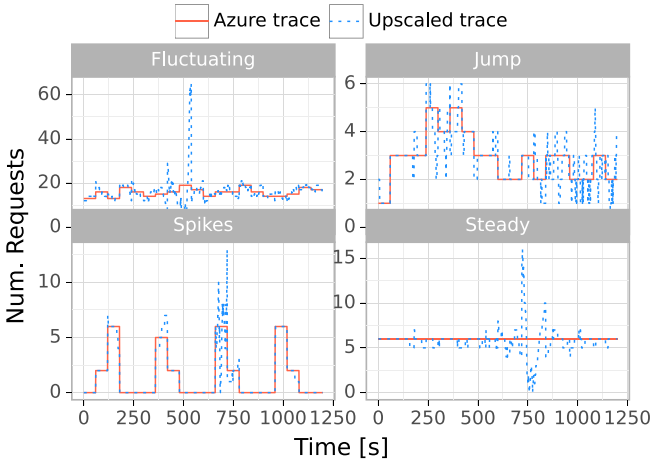


Fig. 9. Typical serverless invocation patterns over 20 min.

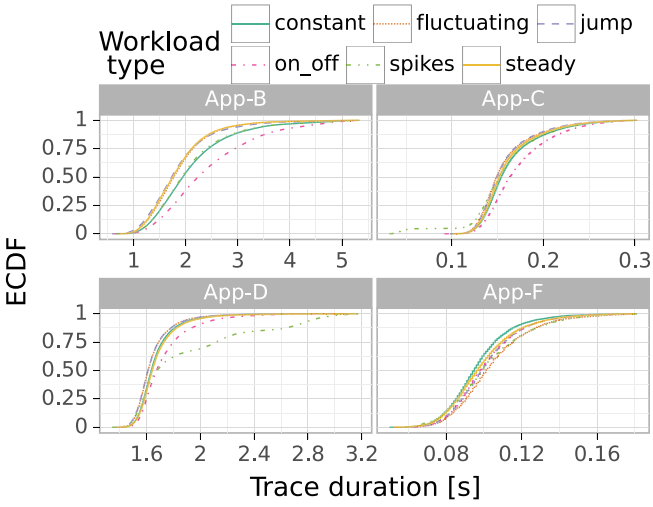


Fig. 10. End-to-end latency for different invocation patterns clipped at the 99th percentile.

Observation 4: Different invocation patterns with per-second load fluctuations do not meaningfully affect the median end-to-end latency except when the workload causes rate limiting of external services.

no relevant difference with the exception of App-D, where the peak invocation rates of the *spikes* workload reach the rate limit of the facial recognition service causing external service delays.

The number of initial cold starts differs by invocation pattern but remains very low after the 60 seconds warmup time. The *on_off* and *spikes* patterns have higher peak invocation rates and trigger more initial cold starts. However, after the warmup time, additional cold starts are rare (below 10).

6.4. Discussion

Our results emphasize the importance of serverless benchmarking that integrates fine-grained latency breakdown analysis and varied

[12] We collected over 700K traces for App-B to App-J using the invocation patterns described later in this subsection with an average rate of 20 reqs/sec. We tried different concurrency levels but noticed that long-running applications were overloaded and short-running applications were served by few function instances.

[13] We reported this and additional issues related to clock drifting and trace correctness to AWS for further investigation.

[14] We explored different time resolutions (2 weeks, 1 day, 4 hours, 1 hour, 30 min, 20 min, 10 min) and found that hourly patterns are similar enough to 20 minutes, which is feasible cost-wise for repeated experimentation with many different applications under varying configurations.

benchmark applications. Furthermore, our experiments lead to relevant implications for serverless practitioners and researchers.

Slowdowns are caused by control flow and coordination, not computation. Our trace-based latency break-down suggests that future research should go beyond computation-optimization approaches [53,56,66], given how little computation time contributes to the end-to-end latency of many applications. The high fraction of external service time shows that fast data exchange between stateless functions remains a key challenge for serverless applications. Many applications would benefit from low-latency storage solutions such as Pocket [67], Shredder [68], or Locus [69]. Finally, efficient function coordination through triggers [7,22] and workflow orchestration [70,71] deserves more research attention given the high transition delays.

Cold start times are best improved by debloating language runtimes. Language runtimes should be the primary focus for optimizing cold start latency given their major impact, adding >500 ms overhead for most applications. Existing runtimes were not designed for serverless architectures and recent optimizations for Java [72] and .NET [73] achieve large speedups of up to 10×, though sometimes at the cost of more memory usage or larger deployment sizes. Debloating system stacks [74] and application dependencies [75] is another promising optimization motivated by large initialization overheads for applications with large dependency trees (e.g., App-E and App-J). Alternatively, serverless developers can select languages with lower runtime initialization overhead, such as Golang [63].

The main cause of tail-latency problems for warm invocations are external services and poorly chosen triggers. Latency-critical applications should carefully choose external services and trigger types. Measurement studies covering different external services can guide the initial selection process [5,7,9,10]. Our trace-based latency breakdown confirms these findings and identifies cloud storage as a key contributor to performance variability [5]. Beyond that, ServiTrace can provide insights into alternative application implementations. For example, applications using database services (App-C, App-F, App-G, App-H) exhibit better tail latency than those using cloud storage (App-B, App-I, App-J). However, initializing a database connection can add additional cold start delay (cf., Fig. 7). For asynchronous orchestration, choosing appropriate trigger types is crucial as the cloud storage trigger introduces massive tail latency (e.g., App-B in Fig. 6). In contrast, the pub/sub trigger used in App-C adds minimal tail latency. However, queueing time may become an issue as non-HTTP-triggered functions have lower scheduling priority [76].

Serverless fulfills its core promise of stable performance under bursty workloads. Our results show that serverless is indeed well-suited for bursty workloads after initial cold starts and when staying below platform-specific rate limits. Hence, serverless fulfills its promise of built-in scalability under the given load levels for our 10 applications. This result was somewhat unexpected, especially contrasting with prior research [5,10,51]. However, of course, bursty workloads may still negatively impact performance on different platforms or with even more rapid bursts than what we evaluated (e.g., per-microsecond bursts rather than per-second bursts).

6.5. Limitations

Despite careful design, we cannot avoid a small number of limitations in our design and results.

First, the presented results are specific to the AWS serverless platform. The applications were implemented and evaluated on AWS. Similarly, ServiTrace itself relies on the comprehensive tracing information offered by the AWS X-Ray service. The reason for focusing on AWS in this work is one of scope, with the knowledge that this cloud provider is considered to be state-of-the-art and, at the moment, a popular solution in the serverless domain. However, conceptually ServiTrace enables benchmarking of a wider range of cloud providers and is designed for extensibility. Some cloud providers may provide less detailed tracing information than X-Ray—a mostly technical issue that we believe will lessen as the market matures, or can be overcome with third-party tooling.

Second, we do not tune the performance of the individual benchmark applications and functions. For instance, increased memory allocation in AWS Lambda has been studied extensively [9,11,53,56–59] and typically provides a function with more CPU shares, more priority in IO, and higher network throughput limits. We rely on the default settings provided by the platform, which is common practice in benchmark design [27] to increase the fairness of comparison. The community could further work to investigate the ideal settings for each application in the benchmarking suite.

Third, specifically for the invocation pattern results, there are limitations because of the Azure dataset, which only provides data on a per-minute granularity. For simulating extremely bursty workloads, a more fine-grained configuration would be necessary, for example, to configure a burst that happens within a few milliseconds using custom invocation scenarios (Section 5.2.2). This may explain why we observed only a limited impact of different invocation patterns on end-to-end latency in Section 6.3.

Fourth, tracing instrumentation could introduce overhead to the performance of the benchmark applications. Tracing libraries increase the deployment and memory footprint [77], which can lead to increased cold start times [78], but they are often required anyways in production deployments [79]. Our testing of sequential trace points has shown that asynchronous tracing APIs minimize the runtime overhead below measurement granularity (i.e., <1 ms). However, asynchronous trace data uploading comes with a limitation at high load levels, where traces become invalid due to discarded trace spans caused by an X-Ray buffer overflow in the AWS Lambda infrastructure. Future work can use request sampling to achieve higher load levels at the cost of partial observability.

We experimentally found that X-Ray drops traces after 1000 req/s. The maximum invocation rates we use (200 req/s) do not generate enough tracing requests to trigger drops. There should be no trace drops as long as the throughput remains under 1000 req/s, even with sub-second bursts. To reduce tracing overhead when collecting a large amount of data at high throughput, users could use a tail sampling approach [80].

This work depends on a unified tracing framework such as X-Ray or OpenTelemetry to extract fine-grained information from applications. Such frameworks currently do not support accelerators (e.g., GPUs), limiting this work's usefulness in those scenarios.

We identify but do not address the possible issue of long-term performance changes in cloud settings. Cloud providers iterate rapidly on their services. Similarly, the operational policies and practices of providers can change, which can influence performance over time, sporadically or even permanently. Though this type of experimentation could not fit within the scope, its importance of it is clear to serverless computing in general. Future work enabled by our replication package could address this situation through techniques such as periodic, long-term measurements in a longitudinal study.

7. Related work

The distributed tracing approach in Section 4 in particular, and ServiTrace in general, advance the field of serverless benchmarks and, more generally, of performance analysis.

Distributed trace analysis. Our work has a new focus, on serverless computing. Distributed tracing is common in microservice architectures

Table 4
Summary of related serverless benchmarks.

Reference	Focus	Insights		Workloads		Scope				
		white-box	async	concurrent	trace-based	apps	func/app	micro	langs	services
faas-profiler [12]	Server-level overheads	✓	✗	✓	✗	5	1	28	2	0
vHive [13]	Cold-start breakdown	✓	✗	✗	✗	10	1	0	1	1
ServerlessBench [59]	Diverse test cases	✗	✗	✗	✗	4	1–7	10	4	1
SeBS [50]	Memory size impact	✗	✗	✓	✗	10	1	0	2	1
FunctionBench [39]	Diverse functions	✗	✗	✗	✗	8	1	6	1	1
FaaSDom [63]	Language comparison	✗	✗	✓	✗	0	–	5	4	0
BeFaaS [93]	Application-centric	(✓)	✗	✓	✗	1	17	0	1	1
ServiTrace (this work)	White-box analysis	✓	✓	✓	✓	10	1–21	0	5	7

but its practice and analysis are big challenges across all software engineering [81–83]. Current production systems such as Canopy [32] from Facebook are primarily used for ad hoc manual analysis [32,83] but research proposed several techniques for automated trace analysis. [84] introduce a formal notation for causality and time and survey approaches for detecting causal relationships in distributed systems. Pivot tracing [85] introduces an efficient *happened-before join* operator to facilitate cross-component event correlation. [86] present algorithms for critical path analysis and trace graph comparison based on their generalized graph representation of execution traces [87]. FIRM [20] combines critical path and critical component analysis with machine learning models to identify and mitigate service level objective (SLO) violations. [88] use graph clustering to characterize the call graph dependency structure and performance of production microservice at Alibaba.

Although *tracing for serverless computing* raises several new challenges, it has received little attention. GammaRay [77] augments AWS X-Ray to track casual ordering and Lowgo [89] proposes a tracing tool for multi-cloud serverless applications. A comparison study of different serverless tracing tools investigates how well they detect different types of faults [90] and Costradamus [91] uses distributed tracing to estimate per-request costs. However, serverless tracing is still emerging and trace analysis remains a largely manual process [92]. Provider-managed infrastructure limits access to 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 as performed in FIRM [20]. In contrast, ServiTrace offers an innovative and pragmatic solution for *detailed* latency-breakdown analysis of *asynchronous* serverless applications (details in Section 4.2).

Serverless benchmarks and measurement frameworks. Table 4 compares ServiTrace with the most important serverless benchmarks and performance frameworks. Our study

1. enables insights through fine-grained white-box analysis across a variety of situations common in production, including asynchronous invocations and external services,
2. adds realistic applications and workloads based on real-world characteristics [25,29,35],
3. has a wider scope in the design space of serverless applications (see Section 5.1), and
4. does not rely on low-level, server-side tracing, which is not available for public serverless platforms.

ServiTrace complements serverless *platform* benchmarks and greatly extends *application-level* benchmarks that interact with cloud services. Platform benchmarks such as vHive [13] enable detailed white-box analysis across the entire FaaS stack, including server-level analysis (e.g., branch prediction) as shown by faas-profiler [12]. Complementary, ServiTrace targets the interactions between applications (L5) and services (L4) as depicted in Fig. 1, enabling the analysis of realistic applications in large-scale production deployments, which is impossible with self-hosted platform benchmarks (as argued in Section 4). Most existing benchmarks focus on single-function applications [12,13,39,50] or artificial micro-benchmarks [12,39,59,63], and ServerlessBench [59] and BeFaaS [93] include larger applications but are limited to a single external service (i.e., database), apart from an API gateway used by all benchmarks for easy invocation. We observe similar computation times for individual functions as other works observe when they use functions with a large amount of memory. For example, our thumbnail generator requires 194 ms of computation, which is comparable to the 124.5 ms spent by the thumbnail function in SeBS [50]. The external service communication latency of applications that only use a database, F (142 ms) and G (54 ms), is higher than that reported by BeFaaS [93] (<10 ms), which utilizes an in-memory database. It is similar to ServerlessBench [59] (150 ms), which uses CouchDB, a database that writes to disk.

Further, ServiTrace is the first-of-its-kind to support trace-based workload generation and detailed insights into realistic applications; particularly, no prior benchmark supports detailed white-box tracing across asynchronous call boundaries and diverse external services. The white-box analysis of faas-profiler [12] and vHive [13] relies on low-level tracing and thus is not applicable to public serverless platforms, and BeFaaS [93] only supports coarse-grained, language-specific, function-level tracing.

Performance evaluation of serverless platforms. Performance is an important and commonly studied aspect [94,95] of serverless computing. Over 100 studies from academia and industry have already appeared according to several literature reviews [15,16,94,95]. Commonly investigated topics include scalability [14,96], cold starts [9,78], performance variability [10,97], instance recycling times [9,98], and the impact of parameters such as memory size [11,99,100], or programming language [63,101]. These studies tend to rely on single-purpose micro-benchmarks and most commonly target single-core CPU performance [15], whereas the ServiTrace application-level benchmarking suite includes 10 realistic and diverse applications (Section 5.1). Whereas prior work is limited to synchronous invocations and typically relies on client-side measurements with few exceptions that explore asynchronous FaaS [77,102] and distributed tracing [77] using simple scenarios, ServiTrace supports asynchronous invocations across diverse external services and contributes a novel approach for detailed latency breakdown analysis of distributed serverless traces (Section 4). Further, reproducibility [31] remains a big challenge in serverless performance studies, as analyzed recently [15], and ServiTrace is designed (Section 3) and implemented (Section 5) to mitigate reproducibility challenges through automated containerized benchmark orchestration.

8. Conclusion

Due to their compositional and asynchronous nature, serverless applications and the platforms executing them are challenging to

benchmark. We designed and implemented ServiTrace, an approach for fine-grained distributed trace analysis and an application-level benchmarking suite for serverless architectures. Unlike existing approaches, ServiTrace:

1. proposes a novel algorithm and heuristics to enable white-box analysis even for asynchronous applications and data produced by distributed traces of serverless applications,
2. leverages a suite of 10 diverse and realistic applications (importantly, including both synchronous and asynchronous cases) and generates trace-based workloads, and
3. supports end-to-end experiments, capturing fine-grained application-level performance and enabling reproducible results.

Using ServiTrace, we conducted a large-scale empirical investigation of the AWS serverless platform, collecting over 7.5 million execution traces. We observe that median end-to-end latency is most often dominated by external service calls, orchestration, or by waiting for asynchronous triggers. Excessive tail latency is similarly caused more by external services (particularly object storage) than any computation inherent to serverless applications. Regarding cold starts, our tracing results indicate that investment into simplifying runtime environments or slimming them down (e.g., as Golang does) is the most promising angle to speed up scaling. Finally, our experiments confirm the AWS platform can react effectively to workload differences, even to challenging bursty invocation patterns, for most applications.

ServiTrace, and the distributed tracing and general serverless benchmarking concepts demonstrated by it, can be used by practitioners for detailed performance analysis of their own applications (e.g., to evaluate trade-offs between alternative services). Platform engineers can use our approach and tooling to further improve serverless platforms. We envision ServiTrace to become an integral part of the evaluation of future serverless research contributions, also as an extensible basis.

CRediT authorship contribution statement

Joel Scheuner: Software, Project administration, Data curation, Conceptualization; **Simon Eismann:** Software, Methodology, Investigation, Data curation; **Sacheendra Talluri:** Software, Methodology, Investigation; **Erwin van Eyk:** Software, Investigation; **Cristina Abad:** Supervision, Investigation; **Philipp Leitner:** Investigation, Supervision, Funding acquisition; **Alexandru Iosup:** Supervision, Funding acquisition, Investigation.

Data availability

The data and code is publicly available on Github.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Sacheendra Talluri reports financial support was provided by EU Framework Programme for Research and Innovation Future and Emerging Technologies. Sacheendra Talluri reports financial support was provided by EU Framework Programme for Research and Innovation Marie Sklodowska-Curie Actions. Sacheendra Talluri reports financial support was provided by Dutch National Growth Fund 6G FNS. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] E.V. Eyk, L. Toader, S. Talluri, L. Versluis, A. Uta, A. Iosup, Serverless is more: from paas to present cloud computing, IEEE Internet Comput. 22 (5) (2018) 8–17.

[2] P.C. Castro, V. Ishakian, V. Muthusamy, A. Slominski, The rise of serverless computing, Commun. ACM 62 (12) (2019) 44–54.

[3] E.V. Eyk, A. Iosup, S. Seif, 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), the 2nd International Workshop on Serverless Computing (WOSC), 2017, pp. 1–4.

[4] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N.J. Yadwadkar, R.A. Popa, J.E. Gonzalez, I. Stoica, D.A. Patterson, What serverless computing is and should become: the next phase of cloud computing, Commun. ACM 64 (2021) 76–84.

[5] D. Ustiugov, T. Amariucai, B. Grot, Analyzing tail latency in serverless clouds with stellar, in: IEEE International Symposium on Workload Characterization (IISWC), (2021) , pp. 51–62. http://www.iiswc.org/iiswc2021/index.html.

[6] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, C. Delimitrou, An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems, in: Proceedings of the 24Th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019, pp. 3–18.

[7] I. Pelle, J. Czentye, J. Dóka, B. Sonkoly, Towards latency sensitive cloud native applications: a performance study on AWS, in: Proceedings of the 12th IEEE International Conference on Cloud Computing (CLOUD), the 12th IEEE International Conference on Cloud Computing (CLOUD), 2019, pp. 272–280.

[8] S. Quinn, R. Cordingly, W. Lloyd, Implications of alternative serverless application control flow methods, in: Proceedings of the Seventh International Workshop on Serverless Computing (WoSC), the Seventh International Workshop on Serverless Computing (WoSC), 2021, pp. 17–22.

[9] L. Wang, M. Li, Y. Zhang, T. Ristenpart, M. Swift, Peeking behind the curtains of serverless platforms, in: Proceedings of the USENIX Annual Technical Conference (ATC), the USENIX Annual Technical Conference (ATC), 2018, pp. 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang.

[10] H. Lee, K. Satyam, G.C. Fox, Evaluation of production serverless computing environments, in: Proceedings of the 11th IEEE CLOUD: 3rd International Workshop on Serverless Computing (WoSC), the 11th IEEE CLOUD: 3rd International Workshop on Serverless Computing (WoSC), 2018, pp. 442–450.

[11] K. Figiela, A. Gajek, A. Zima, B. Obrok, M. Malawski, Performance evaluation of heterogeneous cloud functions, Concurrency Comput. Pract. Exp. 30 (23) (2018).

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

[13] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, B. Grot, Benchmarking, analysis, and optimization of serverless function snapshots, in: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021, pp. 559–572.

[14] J. Kuhlenkamp, S. Werner, M.C. Borges, D. Ernst, 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), the 35th ACM/SIGAPP Symposium on Applied Computing (SAC), 2020, pp. 1576–1585.

[15] J. Scheuner, P. Leitner, Function-as-a-service performance evaluation: a multivocal literature review, J. Syst. Softw. (JSS) 170 (2020).

[16] A. Raza, I. Matta, N. Akhtar, V. Kalavri, V. Isahagian, Sok: function-as-a-service: from an application developer's perspective, J. Syst. Res. (JSys) 1 (1) (2021).

[17] J. Mace, End-to-end tracing: Adoption and use cases, Technical Report, Brown University, 2017. https://cs.brown.edu/jcmace/papers/mace2017survey.pdf.

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

[19] B.H. Sigelman, L.A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C. Shanbhag, Dapper, a large-scale distributed systems tracing infrastructure, 2010. https://research.google/pubs/pub36356/.

[20] H. Qiu, S.S. Banerjee, S. Jha, Z.T. Kalbarczyk, 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), 2020, pp. 805–825. https://www.usenix.org/conference/osdi20/presentation/qiu.

[21] S. Satija, C. Ye, R. Kosgi, A. Jain, R. Kankaria, Y. Chen, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, K. Srinivasan, Cloudscape: a study of storage services in modern cloud architectures, in: H.S. Gunawi, V. Tarasov (Eds.), 23Rd USENIX Conference on File and Storage Technologies, FAST 2025, Santa Clara, CA, 2025, pp. 103–121. USENIX Association, https://www.usenix.org/conference/fast25/presentation/satija.

[22] P.G. López, A. Arjona, J. Sampé, A. Slominski, L. Villard, Triggerflow: trigger-based orchestration of serverless workflows, in: The 14th ACM International Conference on Distributed and Event-based Systems (DEBS), 2020, pp. 3–14.

[23] Wilkinson, The FAIR guiding principles for scientific data management and stewardship, Nature SciData 3 (2016).

[24] E.V. Eyk, A. Iosup, J. Grohmann, S. Eismann, A. Bauer, L. Versluis, L. Toader, N. Schmitt, N. Herbst, C.L. Abad, The SPEC-RG reference architecture for FaaS: from microservices and containers to serverless platforms, IEEE Internet Comput. 23 (6) (2019) 7–18.

[25] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, R. Bianchini, Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider, in: USENIX Annual Technical Conference (ATC), 2020, pp. 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad.

[26] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. Warke, H. Ludwig, D. Hildebrand, A.R. Butt, Improving docker registry design based on production workload analysis, in: 16th USENIX Conference on File and Storage Technologies (FAST), 2018, pp. 265–278. https://www.usenix.org/conference/fast18/presentation/anwar.

[27] J. Kistowski, J.A. Arnold, K. Huppler, K. Lange, J.L. Henning, P. Cao, How to build a benchmark, in: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPC), the 6th ACM/SPEC International Conference on Performance Engineering (ICPC), 2015, pp. 333–336.

[28] W. Hasselbring, Benchmarking as empirical standard in software engineering research, in: Evaluation and Assessment in Software Engineering (EASE), 2021, pp. 365–372.

[29] S. Eismann, J. Scheuner, E.V. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C.L. Abad, A. Iosup, The state of serverless applications: collection, characterization, and community consensus, 2021.

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

[31] B.N. Taylor, C.E. Kuyatt, Guidelines for evaluating and expressing the uncertainty of NIST measurement results, Technical Report, National Institute of Standards and Technology, 1994. https://emtoolbox.nist.gov/Publications/NISTTechnicalNote1297s.pdf.

[32] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'neill, K.W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, Y.J. Song, Canopy: an end-to-end performance tracing and analysis system, in: Proceedings of the 26th Symposium on Operating Systems Principles (SOSP, the 26th Symposium on Operating Systems Principles (SOSP, 2017, pp. 34–50.

[33] A. Najafi, A. Tai, M. Wei, Systems research is running out of time, in: Workshop on Hot Topics in Operating Systems (HotOS '21), 2021, pp. 65–71.

[34] T. Young, Open Telemetry: Sync and async children (follows_from), 2019. https://github.com/open-telemetry/opentelemetry-specification/issues/65.

[35] A. Williams, Guide to serverless technologies, 2018. https://thenewstack.io/ebooks/serverless/guide-to-serverless-technologies/.

[36] W. Amazon, Services, Minimal Baseline, 2022. https://serverlessland.com/patterns/apigw-lambda-cdk.

[37] V. Yussupov, U. Breitenbücher, F. Leymann, 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), the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC), 2019, pp. 273–283.

[38] W. Amazon, Services, Facial Recognition, 2021. https://image-processing.serverlessworkshops.io.

[39] J. Kim, K. Lee, FunctionBench: A suite of workloads for serverless cloud function service, Technical Report, CLOUD WIP, 2019.

[40] A. Karandikar, Real-world Backend, 2022. https://github.com/anishkny/realworld-dynamodb-lambda.

[41] Nordstrom, Hello Retail, 2017. https://web.archive.org/web/20210119044741 https://acloudguru.com/blog/engineering/serverless-event-sourcing-at-nordstrom-ea.

[42] A.V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. Kistowski, A. Ali-Eldin, C.L. Abad, J.N. Amaral, P. Tuma, A. Iosup, Methodological principles for reproducible performance evaluation in cloud computing, IEEE Trans. Softw. Eng. (TSE) 47 (8) (2019) 93–94.

[43] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, Y. Cheng, FaaSNet: scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute, in: USENIX Annual Technical Conf. (ATC), 2021, pp. 443–457. https://www.usenix.org/conference/atc21/presentation/wang-ao.

[44] M. Crovella, A. Bestavros, Self-similarity in world wide web traffic: evidence and possible causes, IEEE/ACM Trans. Netw. 5 (6) (1997) 835–846.

[45] C. Avin, M. Ghobadi, C. Griner, S. Schmid, On the complexity of traffic traces and implications, Proc. ACM Meas. Anal. Comput. Syst. 4 (1) (2020) 29.

[46] Aws, AWS Lambda Developer Guide, Pdf page 225, https://docs.aws.amazon.com/pdfs/lambda/latest/dg/lambda-dg.

[47] R. Satish, S. Talluri, S. Sivakumar, M. Jansen, A. Iosup, Performance characterization of data store event trigger mechanisms for serverless computing, in: Proceedings of the 25th IEEE International Symposium on Cluster, Cloud, and Internet Computing (CCGRID 2025), the 25th IEEE International Symposium on Cluster, Cloud, and Internet Computing (CCGRID 2025)Troms, Norway, IEEE, 2025.

[48] J. Spillner, M. Al-Ameen, Serverless literature dataset, 2019.

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

[50] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, T. Hoefler, Sebs: a serverless benchmark suite for function-as-a-service computing, in: Proceedings of the 22Nd International Middleware Conference, the 22nd International Middleware Conference, 2021, pp. 64–78.

[51] D. Barcelona-Pons, P.G. á López, Benchmarking parallelism in faas platforms, Future Generation Computer Systems 124 (2021) 268–284.

[52] J. Wen, Y. Liu, Z. Chen, J. Chen, Y. Ma, Characterizing commodity serverless computing platforms, 2021.

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

[54] W. Amazon, Services, Configuring function memory (console), in: Lambda Documentation, 2022. https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.htm#configuration-memory-console.

[55] A. Casalboni, AWS Lambda Power Tuning, 2019. https://github.com/alexcasalboni/aws-lambda-power-tuning.

[56] N. Akhtar, A. Raza, V. Ishakian, I. Matta, COSE: Configuring serverless functions using statistical learning, in: 39th IEEE Conference on Computer Communications (INFOCOM), 2020, pp. 129–138.

[57] W. Lloyd, M. Vu, B. Zhang, O. David, G. Leavesley, Improving application migration to serverless computing platforms: latency mitigation with keep-alive workloads, in: Companion of the 11th IEEE/ACM UCC: 4th International Workshop on Serverless Computing (WoSC), 2018, pp. 195.

[58] H. Wang, D. Niu, B. Li, Distributed machine learning with a serverless architecture, in: IEEE Conference on Computer Communications (INFOCOM), 2019, pp. 1288–1296.

[59] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, H. Chen, Characterizing serverless platforms with serverlessbench, in: Proceedings of the ACM Symposium on Cloud Computing (SoCC), the ACM Symposium on Cloud Computing (SoCC), 2020, pp. 30–44.

[60] S. Salvador, P. Chan, Toward accurate dynamic time warping in linear time and space, Intell. Data Anal. 11 (5) (2007) 561–580. http://content.iospress.com/articles/intelligent-data-analysis/ida00303.

[61] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, D. Popa, Firecracker: lightweight virtualization for serverless applications, in: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2020, pp. 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache.

[62] M. Brooker, A.C. Catangiu, M. Danilov, A. Graf, C. Maccarthaigh, A. Sandu, Restoring uniqueness in microvm snapshots, Technical Report, arXiv preprint, 2021. https://arxiv.org/abs/2102.12892.

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

[64] B. Minic, Improving cold start times of Java AWS Lambda functions using GraalVM and native images, 2021. https://shinesolutions.com/2021/08/30/improving-cold-start-times-of-java-aws-lambda-functions-using-graalvm-and\protect\penalty-\@M-native-images/.

[65] J. Dean, L.A. Barroso, The tail at scale, Commun. ACM 56 (2013) 74–80. http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext.

[66] A. Ali, R. Pinciroli, F. Yan, E. Smirni, Batch: machine learning inference serving on serverless platforms with adaptive batching, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, the International Conference for High Performance Computing, Networking, Storage and Analysis, 2020, pp. 1–15.

[67] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, C. Kozyrakis, Pocket: elastic ephemeral storage for serverless analytics, 13th USENIX Symposium on Operating Systems Design and Implementation 18 (2018) 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic.

[68] T. Zhang, D. Xie, F. Li, R. Stutsman, Narrowing the gap between serverless and its state with storage functions, in: Proceedings of the ACM Symposium on Cloud Computing, the ACM Symposium on Cloud Computing, 2019, pp. 1–12.

[69] Q. Pu, S. Venkataraman, I. Stoica, Shuffling, fast and slow: scalable analytics on serverless infrastructure, in: Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI), the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2019, pp. 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu.

[70] A. Mahgoub, E.B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, S. Chaterji, WISEFUSE: Workload characterization and DAG transformation for serverless workflows, Proc. ACM Meas. Anal. Comput. Syst 6 (2) (2022) 28.

[71] C. Lin, H. Khazaei, Modeling and optimization of performance and cost of serverless applications, IEEE Trans. Parallel Distrib. Syst. 32 (3) (2020) 615–632.

[72] A. Filichkin, GraalVM AWS Lambda or solving Java cold start problem, 2021. https://filia-aleks.medium.com/graalvm-aws-lambda-or-solving-java-cold-start-problem.

[73] B. Schaatsbergen, Pre-jitting in AWS Lambda functions, 2021. https://web.archive.org/web/20210807184839/https://www.bschaatsbergen.com/pre-jitting-in-lambda.

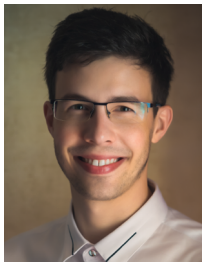
[74] S. Kuenzer, V. Badoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, S. Teodorescu, C. Raducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, F. Huici, Unikraft: fast, specialized unikernels the easy way, in: 16th European Conference on Computer Systems (EuroSys), 2021, pp. 376–394.

[75] C. Soto-Valero, T. Durieux, N. Harrand, B. Baudry, Coverage-based debloating for java bytecode, 2022.

[76] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, S. Lanka, Sequoia: enabling quality-of-service in serverless computing, in: Proceedings of the ACM Symposium on Cloud Computing (SoCC), the ACM Symposium on Cloud Computing (SoCC), 2020, pp. 311–327.

[77] W. Lin, C. Krintz, R. Wolski, M. Zhang, X. Cai, T. Li, W. Xu, Tracking causal order in AWS lambda applications, in: Proceedings of the IEEE International Conference on Cloud Engineering (IC2E), the IEEE International Conference on Cloud Engineering (IC2E), 2018, pp. 50–60.

[78] J. Manner, M. Endreß, T. Heckel, 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), 2018, pp. 181–188.
[79] C. D, .-C.W. Erquinigo, A. Tang, Reverse debugging at scale, 2021. https://engineering.fb.com/2021/04/27/developer-tools/reverse-debugging/.
[80] L. Zhang, Z. Xie, V. Anand, Y. Vigfusson, J. Mace, The benefit of hindsight: tracing edge-cases in distributed systems, in: M. Balakrishnan, M. Ghobadi (Eds.), 20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, 2023, pp. 321–339. USENIX Association, https://www.usenix.org/conference/nsdi23/presentation/zhang-lei.
[81] M. Waseem, P. Liang, M. Shahin, A.D. Salle, G. Márquez, Design, monitoring, and testing of microservices systems: the practitioners' perspective, J. Syst. Softw. (JSS) 182 (2021).
[82] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, X. Liu, Enjoy your observability: an industrial survey of microservice tracing and analysis, Empirical Softw. Eng. (EMSE) 27 (1) (2021) 25.
[83] A. Bento, J. Correia, R. Filipe, F. Araújo, J.S. Cardoso, Automated analysis of distributed tracing: challenges and research directions, J. Grid Comput. 19 (1) (2021) 9.
[84] R. Schwarz, F. Mattern, Detecting causal relationships in distributed computations: in search of the holy grail, Distrib. Comput. 7 (3) (1994) 149–174.
[85] J. Mace, R. Roelke, R. Fonseca, Pivot tracing: dynamic causal monitoring for distributed systems, Commun. ACM 63 (3) (2020) 94–102.
[86] M. Hendriks, J. Verriet, T. Basten, B.D. Theelen, M. Brassé, L.J. Somers, Analyzing execution traces: critical-path analysis and distance analysis, Int. J. Softw. Tools Technol. Transf. 19 (4) (2017) 487–510.
[87] M. Hendriks, F.W. Vaandrager, Reconstructing critical paths from execution traces, in: 15th IEEE International Conference on Computational Science and Engineering, CSE, 2012, pp. 524–531.
[88] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, C. Xu, Characterizing microservice dependency and performance: alibaba trace analysis, in: ACM Symposium on Cloud Computing (SoCC), 2021, pp. 412–426.
[89] W. Lin, C. Krintz, R. Wolski, Tracing function dependencies across clouds, in: 2018 IEEE 11Th International Conference on Cloud Computing (CLOUD), 2018, pp. 253–260.
[90] M.C. Borges, S. Werner, A. Kilic, Faaster troubleshooting - evaluating distributed tracing approaches for serverless applications, in: IEEE International Conference on Cloud Engineering (IC2E), 2021, pp. 83–90.
[91] J. Kuhlenkamp, M. Klems, Costradamus: a cost-tracing system for cloud-based software services, Proceedings of the 15th International Conference on Service-Oriented Computing (ICSOC) 10601 (2017) 657–672.
[92] V. Lenarduzzi, A. Panichella, Serverless testing: tool vendors' and experts' points of view, IEEE Softw. 38 (1) (2021) 54–60.
[93] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, D. Bermbach, Befaas: an application-centric benchmarking framework for faas platforms, in: IEEE International Conference on Cloud Engineering (IC2E), 2021, pp. 1–8.
[94] V. Yussupov, U. Breitenbücher, F. Leymann, 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), the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC), 2019, pp. 229–240.
[95] H.B. Hassan, S.A. Barakat, Q.I. Sarhan, Survey on serverless computing, J. Cloud Comput. 10 (1) (2021) 39.
[96] G. Mcgrath, P.R. Brenner, Serverless computing: design, implementation, and performance, in: Proceedings of the 37th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW), the 37th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW), 2017, pp. 405–410.
[97] R. Cordingly, W. Shu, W.J. Lloyd, Predicting performance and cost of serverless computing functions with SAAF, in: IEEE International Conference on Cloud and Big Data (CBDCom), 2020, pp. 640–649.
[98] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, S. Pallickara, Serverless computing: an investigation of factors influencing microservice performance, in: Proceedings of the IEEE International Conference on Cloud Engineering (IC2E), the IEEE International Conference on Cloud Engineering (IC2E), 2018, pp. 159–169.
[99] M. Zhang, Y. Zhu, C. Zhang, J. Liu, Video processing with serverless computing: a measurement study, in: Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), 2019, pp. 61–66.
[100] T. Back, V. Andrikopoulos, Using a microbenchmark to compare function as a service solutions, Proceedings of the 7th European Service-Oriented and Cloud Computing (ESOCC) 11116 (2018) 146–160.
[101] D. Jackson, G. Clynch, An investigation of the impact of language runtime on the performance and cost of serverless functions, in: Companion of the 11th IEEE/ACM UCC: 4th International Workshop on Serverless Computing (WoSC), 2018, pp. 154–160.
[102] D. Balla, M. Maliosz, C. Simon, Performance evaluation of asynchronous FaaS, in: 14th IEEE International Conference on Cloud Computing (CLOUD), 2021, pp. 147–156.

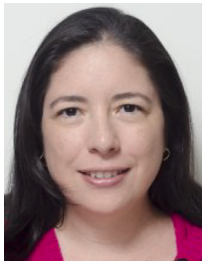


Joel Scheuner is a software engineer at LocalStack, building the world's best open-source local cloud development experience. He has a PhD from Chalmers University of Technology focusing on reproducible performance evaluation of serverless applications and their underlying cloud infrastructure. He is also known for his work with Cloud WorkBench (CWB), a cloud benchmarking toolkit to provision and execute performance benchmarks in cloud environments automatically.

Simon Eismann is a performance engineer at MongoDB. He has a PhD from University of Würzburg focusing on statistical prediction of serverless application characteristics. His work has won the best student paper award at Middleware 2021.

Sacheendra Talluri is a PhD student at Vrije Universiteit Amsterdam. His work focuses on fault tolerance in the cloud and the structure of cloud resource managers.

Erwin van Eyk is a senior software engineer at Temporal. He has previously worked at Platform9 on Kubernetes and serverless workflows.



Cristina Abad is a Professor at ESPOL University in Guayaquil, Ecuador, where she leads the Distributed Systems lab. She has a PhD from the University of Illinois at Urbana-Champaign on a Fullbright fellowship. She was a member of the Hadoop Core team at Yahoo.

Philipp Leitner is an Associate Professor of Software Engineering at Chalmers and University of Gothenburg, Sweden. He leads the Internet Computing and Emerging Technologies Lab (ICET-lab). He holds a PhD degree from Vienna University of Technology focussing on cost-aware service composition. His current research interests are software performance engineering, especially in the cloud computing and machine learning contexts.

Alexandru Iosup is a Full Professor at Vrije Universiteit Amsterdam and leads the AtLarge research group. He has been awarded the Netherlands ICT-Researcher of the Year award, member of the Royal Netherlands Young Academy of Arts and Sciences, and has been knighted by the Romanian Government.