Investigating Performance Overhead of Distributed Tracing in Microservices and Serverless Systems

Anders Nõu
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
anders.nou@gmail.com

Sacheendra Talluri
Alexandru Iosup
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
{s.talluri,a.iosup}@vu.nl

Daniele Bonetta
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
d.bonetta@vu.nl

Abstract

Distributed tracing is crucial to achieve observability in modern distributed systems. However, its adoption introduces performance trade-offs, impacting throughput and latency. This paper investigates the overhead of distributed tracing in microservices and serverless applications. We provide an analysis of the popular OpenTelemetry and Elastic APM distributed tracing frameworks, evaluating their performance impact on microservices and serverless workloads. We highlight and categorize the primary sources of overhead and measure their contribution to performance degradation. The results reveal significant throughput reductions (19-80%) and latency increases (up to 175%) depending on application configurations and execution environments. Our findings reveal that serializing trace data for export is the largest cause of overhead.

CCS Concepts

• **Computer systems organization → Cloud computing**; • **General and reference → Measurement**.

Keywords

distributed tracing, instrumentation, performance, open telemetry

ACM Reference Format:
Anders Nõu, Sacheendra Talluri, Alexandru Iosup, and Daniele Bonetta. 2025. Investigating Performance Overhead of Distributed Tracing in Microservices and Serverless Systems. In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE Companion '25), May 5–9, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3680256.3721316

1 Introduction

Modern cloud software often contains microservices and serverless applications [19, 23]. These new architectures have been adopted in various domains, including web development, data processing, and microservice architectures [10]. However, the complex interconnected nature of these architectures demands high observability to ensure reliability and performance [25].

Distributed tracing has emerged as a necessary tool for monitoring and diagnosing the performance of distributed systems, including microservices and serverless applications [22, 24]. By tracking requests as they flow through various services and components, distributed tracing provides insights into the behavior and interactions within applications [11, 26]. This level of observability is necessary to identify performance bottlenecks, understand system dependencies, and ensure the reliable operation of distributed applications [22]. Despite its benefits, distributed tracing introduces additional overhead, which can negatively impact the throughput and latency of the applications being monitored [14, 16, 20, 27, 28].

Understanding the performance implications of distributed tracing is crucial [11]. Applications that partition work into independently processed tasks can be particularly sensitive to the overhead introduced by tracing [17, 31]. These applications involve low-latency operations, where even small increases in latency can significantly impact overall performance. As a result, it is essential to quantify the performance overhead of distributed tracing and identify the main contributors to this overhead.

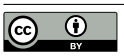
This paper aims to fill the gap in the understanding of distributed tracing overhead, providing a detailed evaluation of the impact of tracing on different system types.

This paper makes the following contributions:

(1) We introduce a systematic framework for measuring tracing overhead across diverse applications and environments in Section 3.
(2) We comparatively analyze the OpenTelemetry and Elastic APM instrumentation tools and analyze their performance for microservices in Section 4.
(3) We analyze the instrumentation overhead of OpenTelemetry for serverless applications in Section 5.
(4) We provide insights into the main contributors to tracing overhead, including the configuration, instrumentation, and export stages in Section 6.

2 Background: Distributed Tracing in Serverless Applications

Distributed tracing is a technique to monitor and track requests as they move through different elements in a distributed system (e.g., messages in a serverless application). Capturing a request's lifecycle provides insights into how services interact, the sequence of operations, and where time is spent within the system. The resulting visibility offers several key benefits. First, it enables *(*End-to-End Traceability), allowing comprehensive monitoring in complex distributed architectures. Subsequently, it enables *Performance Insights* by evaluating performance metrics, including request latency and execution time of specific operations within the applications. It favors *Root Cause Analysis*, as it provides additional context for



This work is licensed under a Creative Commons Attribution 4.0 International License.
ICPE Companion '25, May 5–9, 2025, Toronto, ON, Canada
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1130-5/2025/05
https://doi.org/10.1145/3680256.3721316

troubleshooting errors, making it easier to pinpoint the cause of a performance problem. Finally, it allows a better understanding of Service Dependencies, offering a clear picture of service relationships and interactions [22].

Given the importance of distributed tracing, several frameworks have emerged in the last few years that integrate with the most popular programming languages used to implement serverless applications. In this paper, we focus on the two most prominent frameworks, namely OpenTelemetry and Elastic APM. In the following sections, we briefly introduce the main properties of both frameworks.

OpenTelemetry [8] is an open source observability framework that provides standardized methods for collecting, processing, and exporting telemetry data, including traces, metrics, and logs. OpenTelemetry provides standardized APIs and SDKs for trace data collection, supported in multiple languages and frameworks.

Elastic APM [1] is an application performance monitoring system designed to provide detailed performance data and end-to-end distributed tracing for applications. It is part of Elastic Stack [2], an observability framework designed to collect, analyze, and visualize logs, metrics, and traces.

3 Distributed Tracing Overheads Analysis

In this work, we aim to study the performance trade-offs associated with distributed tracing frameworks and evaluate their potential influence on system performance. To this end, we measured the overhead of tracing frameworks using controlled experiments in a variety of popular applications and deployments. We focus on quantifying the impact of tracing on throughput, latency, and resource utilization, as well as on identifying the main sources of overhead. We have designed and executed three sets of experiments that focused on different applications and purposes. First, we focused on the performance of standalone microservices. This had the goal of measuring the instrumentation overhead of a single web service (e.g., a single HTTP request handler). Second, we focused on serverless applications and platforms. This second experiment had the goal of measuring the impact of tracing and instrumentation in more realistic cloud deployment scenarios. Finally, we expanded on the two previous experiments and analyzed the sources of the performance overhead measured in our experiments with the goal of identifying the root causes of performance degradation.

3.1 Experimental Workloads and Frameworks

We based our microservice performance evaluation on the TechEmpower Framework [7]. For serverless applications, we used the Serverless Benchmark Suite [13] (SeBS). Both benchmarking frameworks provide a variety of workloads and implementations in different programming languages. In every experiment, the baseline consisted of unmodified applications without instrumentation. We utilized OpenTelemetry and Elastic APM to instrument all services, allowing us to evaluate throughput and latency across different workloads. The experiments were executed within a Kubernetes environment, using containerization to ensure consistent and repeatable configurations. The experimental setup involved multiple iterations of tests to account for variability in system performance. Each iteration measured key metrics such as request latency, throughput, and CPU utilization in different tracing configurations. The results were aggregated and analyzed to identify patterns and outliers.

3.2 Experimental setup

All experiments were conducted on a dedicated cluster node with the following hardware specifications: Intel® Xeon® Silver 4416+ Processor at 2.00 GHz, with 20 cores and 40 threads; 256.0GB of DDR4 RAM, consisting of four RAM units configured to operate at 1200 MHz (2400 MT/s); 1.5 TB of SSD storage; Ubuntu 22.04.4 LTS. Regarding langauge runtimes, the versions used are: Python v3.9, Node.js v18, Java 17, and Go 1.21. All experiments have been deployed on a Kubernetes cluster (v 1.20) running Docker 27.2.1. We have used the most recent stable versions of OpenTelemetry APIs (1.25.1 for JavaScript and Python) and likewise for Elastic APM (6.22.0) with a recent installation of the Flask framework (3.0.3).

4 Experiment 1: Microservices

In our first experiment, we explored how distributed tracing influences the performance of simple microservices implemented using popular programming languages. This first experiment aimed to quantify the overhead across different popular frameworks used for microservices development, namely Python Flask, Java Spring, Go http, and Node.js. For this purpose, we assessed these four web frameworks in relation to four separate microservices. By testing these diverse frameworks, we aim to cover a wide range of use cases prevalent in software development due to their widespread use [3, 4, 6]. The variety also ensures in-depth analysis of how distributed tracing performs across different programming paradigms and runtime environments, and reduces the risk that overhead measurements are impacted by language-specific aspects. In this experiment, we evaluated the performance impact of distributed tracing by measuring Requests Per Second (RPS) under realistic conditions simulating peak traffic. We established the impact of instrumentation on the applications by determining the maximum RPS each application can handle while maintaining a constant CPU utilization.

4.1 Applications under test

This first experiment is based on four different microservices included in TechEmpower [5]. TechEmpower is a comprehensive benchmarking suite designed to measure the performance of web frameworks. It provides a standardized methodology for a fair comparison between different web technologies. The suite includes seven test types: JSON serialization, single and multiple database queries, server-side templating, database updates, plain text processing, and caching mechanisms. These endpoints cover various application workflows, enabling a robust evaluation of applications under different workloads. We selected four endpoints for our experiments, covering multiple use cases to thoroughly compare the applications under various workloads. The implemented endpoints are json, db, object, and queries. These endpoints are described in the Techempower benchmark documentation [5].

In this experiment, we compared OpenTelemetry and Elastic APM with a baseline uninstrumented application. Automatic instrumentation was applied to OpenTelemetry and Elastic APM setups across all web frameworks assessed. Additionally, we employed

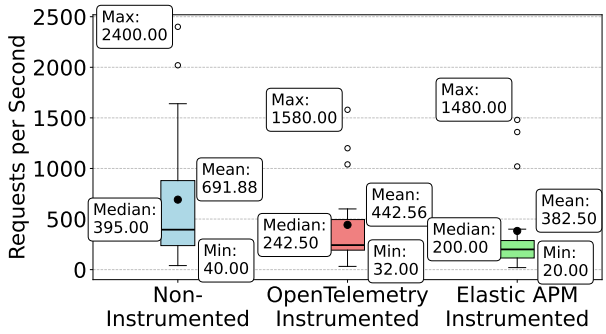


Figure 1: Throughput of microservices benchmarks with different instrumentation frameworks.

PostgreSQL5, a widely-used open-source object-relational database system. Its prevalent usage and support within the TechEmpower Framework made it an appropriate choice for our experimental architecture.

4.1.1 Load generation. To generate the experimental requests workload, we used k6 [9], an open source load testing tool designed to test the performance and reliability of web applications. k6 offers scripting capabilities and a variety of executors to simulate real-world traffic patterns and stress test web applications. We selected k6 because it can generate consistent and customizable load patterns and supports distributed and large-scale performance testing. In our experiment, we configured k6 to use scenarios and the constant-arrival-rate executor to run the load tests. This configuration allowed us to maintain a consistent request rate throughout the duration of the test, ensuring that the applications are subjected to a steady load for stable results. We defined a k6 test scenario for each endpoint, instrumentation method, and combination of framework. Each scenario is configured with duration, number of virtual users, warmup period, and cooling period.

4.2 Results overview

The main findings for this experiment can be summarized as follows.

- Distributed tracing reduces throughput across all frameworks, with declines ranging from 19.55% to 80.18%.
- Java Spring exhibits the lowest overhead with microservice applications, while Node.js shows the most significant impact on throughput.
- OpenTelemetry generally delivers higher throughput than Elastic APM across most frameworks.

The results underscore the importance of selecting appropriate tracing configurations based on application characteristics and programming languages. For example, frameworks with high-level abstractions, such as Python Flask, exhibited greater sensitivity to tracing overhead. In contrast, lower-level frameworks like Go http demonstrated more stable performance under similar conditions.

4.2.1 Throughput. Figure 1 shows the aggregated throughput in requests per second across three configurations. The non-instrumented configuration has the highest throughput, with a median of 395 RPS and a mean of 691.88 RPS, reaching a maximum of 2,400 RPS. When

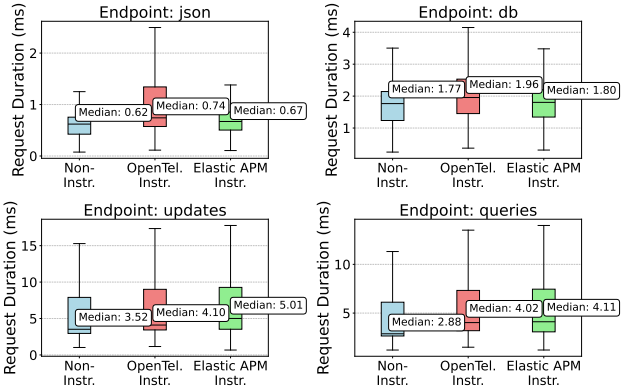


Figure 2: Latency of microservice endpoints with different instrumentation frameworks.

tracing is enabled, throughput decreases significantly. With OpenTelemetry, the median throughput decreases by 38.6% compared to the non-instrumented median. The mean throughput drops by 36.0% and the maximum throughput drops to 1,580 RPS (decreased by 34.20%). The elastic APM instrumented configuration has an even greater impact on throughput, with the median falling to 200 RPS, a 49.4% decrease. The mean throughput drops to 382.5 RPS, marking a reduction of 44.7%, while the maximum value of RPS is reduced to 1,480 RPS. We observe that elastic APM introduces a more significant performance overhead with respect to throughput than OpenTelemetry.

4.2.2 Latency. section 4.2.2 illustrates the distribution and median latency in milliseconds for different instrumentation configurations (non-instrumented, OpenTelemetry instrumented, and Elastic APM instrumented) for the four benchmark endpoints (json, db, updates, and queries). The data is aggregated over the evaluated frameworks to present a general view of latency overhead across different requests. The figure shows that enabling distributed tracing consistently increases latency compared to the non-instrumented configuration. The extent of overhead highly varies depending on the endpoint. For the json endpoint, latency increases by approximately 19%, from a median of 0.62 to 0.74 ms with OpenTelemetry, while elastic APM introduces a lower increase 7%. The upper quartile value is also significantly higher for OpenTelemetry. The impact is more significant for the endpoints for updates and queries. The median latency for the update endpoint increases by 16.5% for OpenTelemetry and approximately 42% for Elastic APM. For queries, the performance overhead for both instrumentation tools is similar: 39.6% for OpenTelemetry and 42.7% for Elastic APM. Overall, the less intensive endpoints json and db incurred less overhead overall and less overhead with Elastic APM compared to OpenTelemetry. However, more intensive benchmark updates and queries exhibited less overhead with OpenTelemetry in comparison to Elastic APM

5 Experiment 2: Serverless Applications

We performed a second experiment to evaluate the performance overhead introduced by distributed tracing in serverless applications. We examine how tracing impacts application latency and compare these effects across different programming frameworks.

The analysis aims to provide insight into the trade-offs involved in implementing tracing in serverless applications.

5.1 Application under test

We designed a series of test scenarios across multiple configurations and environments to evaluate the performance overhead of tracing in serverless applications. The experiments are centered on two of the most popular programming frameworks for microservices, namely Python Flask and Node.js, using five distinct benchmark applications. Each application is executed in two configurations: instrumented (with tracing enabled) and non-instrumented (without tracing). The setup allows us to analyze the impact of tracing across different applications and frameworks, resulting in 20 individual benchmark executions (10 per framework: 5 instrumented and five non-instrumented).

The benchmark applications represent a range of microservices workloads, including low- and high-latency services. We aim to isolate the overhead introduced by tracing and assess how it varies across different frameworks and workloads.

All experiments are based on the Serverless Benchmark Suite (SeBS), a framework designed to evaluate the performance and cost-efficiency of serverless platforms and functions [13]. SeBS facilitates the deployment, execution, and measurement of serverless functions across multiple cloud platforms, including AWS Lambda, Azure Functions, Google Cloud Functions, and Apache OpenWhisk. The framework provides various benchmark applications that reflect typical serverless workloads. We used five SeBS applications, each selected to represent different types of workload to assess the performance impact of tracing comprehensively. The benchmarks include web application tasks, multimedia processing, and a scientific computation task. This diversity helps to evaluate tracing overhead across a wide range of scenarios.

All applications are implemented in Python and Node.js. For benchmarks without a native Node.js version, we developed Node.js implementations based on the Python version and existing code examples. Each application has both instrumented (tracing enabled) and non-instrumented versions, resulting in 20 different configurations (10 per programming language).

The applications used in this experiment are dynamic HTML, uploader, thumbnailer, video processing, and graph pagerank. These applications are described in the SeBS paper [13].

By evaluating these benchmarks, we can analyze how tracing affects various serverless tasks, providing insights into its suitability for different applications.

5.2 Results overview

Serverless functions exhibited different and diverse overhead patterns. Short-duration tasks, often reliant on cold starts, faced latency increases up to 175%. This overhead mainly stemmed from the tracing configuration and data export stages. Long-duration tasks, on the contrary, showed modest increases of 6.7%, highlighting a relative marginal impact of tracing when applications perform long operations.

Cold-start latency was particularly pronounced in environments with extensive configuration requirements: the initialization of tracing components contributed significantly to cold start overhead,

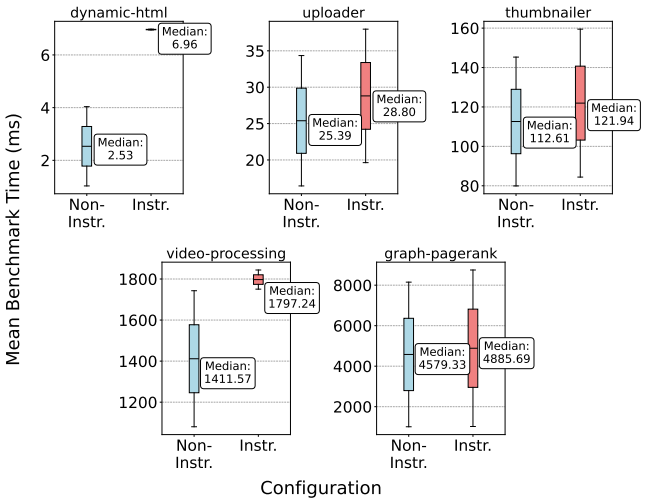


Figure 3: Mean execution time of different applications from the SeBS benchmark.

and optimizing these initialization processes could mitigate the impact on short-duration tasks.

5.2.1 Impact on different applications. Figure 3 presents box plots for five serverless workloads across non-instrumented and instrumented configurations. The results of the benchmarks are aggregated over both Python and Node.js evaluations.

The results indicate that the performance impact of tracing instrumentation varies widely depending on the workload type. For example, dynamic-html workload, with a relatively low baseline latency of 2.53 ms, shows an increase in median latency to 6.96 ms when instrumented, showing an overhead of approximately 175%.

As the median request duration increases, the overhead generally decreases. For example, the uploader and thumbnailer with moderate median request durations show a more modest increase - 13.43% overhead for the uploader and 8.29% for the thumbnailer. However, the video-processing workload with a median latency of approximately 1,411 ms exhibits a substantial increase of 27.32%. This considerable increase is the result of the 70.69% overhead in Python as Table 4.3 shows. The graph pagerank workload, which has the highest baseline latency among workloads with a median of 4,579.33 ms, shows the lowest overhead with 6.69%.

The data highlight that tracing introduces a considerable performance impact across workloads. The functions with lower request duration show the highest increase but still the lowest increase in absolute values. The more compute-intensive workloads incur lower overhead in percentage but higher in terms of absolute values.

6 Overhead Analysis

In distributed tracing, various sources contribute to the overall overhead observed in the application and its instrumentation. In a final experiment, we evaluated the overhead sources in the computational part of distributed tracing [21]. The primary contributors to overhead can be categorized and quantified as follows:

- **Configuration:** Initialization and setup of tracing components, including metadata generation, contributed significantly to cold-start latency in serverless environments.

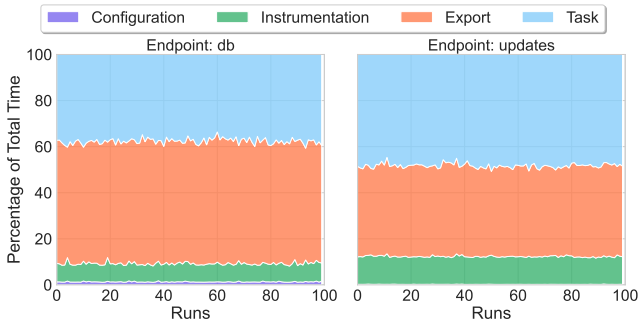


Figure 4: Contribution of different tracing stages to the instrumentation overhead for microservice applications.

- **Instrumentation:** Capturing trace points and enriching metadata introduced computational overhead, particularly in frameworks relying on runtime instrumentation. Detailed profiling revealed that the frequency and granularity of trace points were key factors influencing this overhead.
- **Export:** Transmitting trace data to external storage systems, such as Elasticsearch, accounted for the majority of network-related performance costs. Batch processing and compression techniques were explored as potential optimizations to reduce export overhead.

An overview of the different impact of such overhead sources on the microservices of Section 4 is depicted in Figure 4. As the figure shows, exporting data to an external service is the largest contributor to overhead for microservices. For serverless applications the sources of overhead depend very much on workload-specific aspects of an application, and are not directly influenced by the framework being used.

7 Related Work

Several studies measured either throughput or latency [15, 17, 20, 28–30] of distributed tracing. However, many studies do not report the absolute performance values, making it difficult to perform direct comparisons between tools.

Given the impact of instrumentation overhead, several mitigation strategies have been proposed. Google Dapper [28] samples 0.1% of requests, resulting in 16.3% latency and 1.5% throughput overhead. Canopy [20] uses sampling, with an 8.15% overhead for short duration applications and 0.76% for long ones. NanoLog [29] leads to a 4% increase in latency and a 19% reduction in throughput.

Popular tools such as OpenTelemetry, Jaeger, Zipkin, and Elastic APM vary significantly in the overhead they introduce and in the integration complexity with different systems [12, 18, 22].

8 Conclusion

Distributed tracing is essential for observability but introduces significant performance overhead. Our study quantifies these impacts, categorizes the sources of overhead, and provides actionable insights for mitigating performance penalties while maintaining system visibility. Future work includes exploring lightweight tracing methodologies, dynamic sampling strategies, and integrating machine learning models to predict and optimize tracing configurations in real-time. Our experiment code is open-source and available at https://github.com/atlarge-research/serverless-tracing-overhead.

Acknowledgements

This work was supported by the EU Horizon Graph Massivizer and the EU MSCA Cloudstars projects. This research was partly supported by a National Growth Fund through the Dutch 6G flagship project "Future Network Services".

References

[1] 2024. Application Performance Monitoring (APM). https://www.elastic.co/observability/application-performance-monitoring. Accessed: 2024-05-20.
[2] 2024. Elastic Stack. https://www.elastic.co/elastic-stack. Accessed: 2024-09-22.
[3] 2024. Flask STackoverflow. https://stackoverflow.com/questions/tagged/flask. Accessed: 2024-09-28.
[4] 2024. Node.js Stackoverflow. https://stackoverflow.com/questions/tagged/node.js. Accessed: 2024-09-28.
[5] 2024. Project Information Framework Tests Overview. https://github.com/TechEmpower/FrameworkBenchmarks/wiki/Project-Information-Framework-Tests-Overview. Accessed: 2024-05-05.
[6] 2024. Spring Framework Stackoverflow. https://stackoverflow.com/questions/tagged/spring. Accessed: 2024-09-28.
[7] 2024. Web Framework Benchmarks. https://www.techempower.com/benchmarks. Accessed: 2024-09-05.
[8] 2024. What is OpenTelemetry? https://opentelemetry.io/docs/what-is-opentelemetry/. Accessed: 2024-09-21.
[9] 2025. k6 load testing tool. https://k6.io/.
[10] Gojko Adzic and Robert Chatley. 2017. Serverless computing: economic and architectural impact. In *FSE 2017*. 884–889.
[11] Eric Anderson et al. 2009. Efficient tracing and performance analysis for large distributed systems. In *IEEE MASCOTS 2009*. IEEE, 1–10.
[12] Andre Bento et al. 2021. Automated analysis of distributed tracing: Challenges and research directions. *Journal of Grid Computing* 19, 1 (2021), 9.
[13] Marcin Copik et al. 2021. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Middleware 2021*. 64–78.
[14] Úlfar Erlingsson et al. 2012. Fay: Extensible distributed tracing from kernels to clusters. *ACM TOCS* 30, 4 (2012), 1–35.
[15] Rodrigo Fonseca et al. 2007. {X-Trace}: A pervasive network tracing framework. In *NSDI 2007*.
[16] Loïc Gelle et al. 2021. Combining distributed and kernel tracing for performance analysis of cloud applications. *Electronics* 10, 21 (2021), 2610.
[17] Francis Giraldeau and Michel Dagenais. 2015. Wait analysis of distributed systems using kernel tracing. *IEEE TPDS* 27, 8 (2015), 2450–2461.
[18] Andrea Janes, Xiaozhou Li, and Valentina Lenarduzzi. 2023. Open tracing tools: Overview and critical comparison. *Journal of Systems and Software* (2023), 111793.
[19] Eric Jonas et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).
[20] Jonathan Kaldor et al. 2017. Canopy: An end-to-end performance tracing and analysis system. In *SOSP 2017*. 34–50.
[21] Pedro Las-Casas et al. 2019. Sifter: Scalable sampling for distributed traces, without feature engineering. In *SoCC 2019*. 312–324.
[22] Jonathan Mace. 2017. *End-to-End Tracing: Adoption and Use Cases*. Survey. Brown University.
[23] Garrett McGrath and Paul R Brenner. 2017. Serverless computing: Design, implementation, and performance. In *ICDCS Workshops 2017*. IEEE, 405–410.
[24] Austin Parker and other. 2020. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media.
[25] Raja R Sambasivan et al. 2014. So, you want to trace your distributed system? Key design insights from years of practical experience. *CMU-PDL* 14 (2014).
[26] Bo Sang et al. 2010. Precise, Scalable, and Online Request Tracing for Multitier Services of Black Boxes. *IEEE TPDS* 23 (2010), 1159–1167.
[27] Junxian Shen et al. 2023. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In *SIGCOMM 2023*. 420–437.
[28] Benjamin H Sigelman et al. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).
[29] Stephen Yang, Seo Jin Park, and John Ousterhout. 2018. {NanoLog}: A Nanosecond Scale Logging System. In *ATC 2018*. 335–350.
[30] Lei Zhang et al. 2023. The Benefit of Hindsight: Tracing {Edge-Cases} in Distributed Systems. In *NSDI 2023*. 321–339.
[31] Zoltán Zvara, Péter GN Szabó, Barnabás Balázs, and András Benczúr. 2019. Optimizing distributed data stream processing by tracing. *FGCS* 90 (2019), 578–591.