IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

A SURVEY OF INSTRUMENTATION OVERHEAD, SAMPLING TRADE-OFFS, AND TRACE QUALITY IN DISTRIBUTED TRACING SYSTEMS FOR APIS

[1]Farhat Khan, [2]Urjita Thakar, [3]Sudarshan Varma

[1]Research Fellow, [2]Professor Computer Engineering Department, SGSITS, Indore [3]Director of Technology, Advanz 101 Systems Pvt. Ltd., Indore

[1]Department of Computer Engineering,

[1]Shri Govindram Seksaria Institute of Technology and Science(SGSITS), Indore, India

Abstract: Distributed tracing of software application promises a clear view of what happens to a request as it moves through a software system, but that visibility comes at a cost. Once tracing is enabled, a practical question follows: how much should be collected, what resources will it consume, and how much useful information will the resulting traces provide? This survey follows that question through the evolution of distributed tracing, from Dapper's foundational work on tracing and sampling to modern OpenTelemetry-based evaluations, microbenchmark studies, and application-level measurements. The literature shows that this problem has been studied carefully, but largely in isolation. Sampling has been examined, yet its effects are less well understood in modern application-level settings; runtime overhead and telemetry volume have been measured, but are rarely connected to the quality of the traces produced; and trace quality is discussed extensively, but is seldom evaluated against an independent representation of the application's expected behaviour. The architectural question is similarly unresolved, as matched evaluations of equivalent monolithic and microservice implementations remain limited, while the contribution of different instrumentation mechanisms has not been systematically separated from other technology and workload effects. These gaps point to a broader problem: existing studies can tell us about individual costs or benefits of tracing, but provide less guidance on how these dimensions interact. The survey therefore concludes by outlining an integrated empirical study that brings sampling, performance overhead, telemetry volume, trace quality, architecture, and instrumentation mechanisms into a common experimental framework.

Index Terms **-** Distributed tracing, OpenTelemetry, microservices, monolithic architecture, trace sampling, instrumentation overhead, trace quality, telemetry volume, performance evaluation, observability.

I. INTRODUCTION

Distributed tracing has become a standard part of modern software systems. It is used to follow a request as it moves across services, to identify failures, and to understand performance problems. The basic model was established by Dapper [1] more than fifteen years ago: requests are represented as trees of spans connected through parent links, and sampling is used to limit the amount of data collected.

The tracing of the software applications has grown considerably since then. Additional dynamic information was shown to travel across component boundaries by Pivot Tracing [6], and tracing data was shown to be collected and analyzed at very large scale by Canopy [5]. A more basic question was raised by Sambasivan [7]: how much information should a tracing system actually record, and how accurately should it represent the application behaviour? Together, these works establish that collecting a trace is not enough, the trace must also contain enough correct information to support diagnosis. Standardization has further

changed the landscape, with trace context propagation supported by W3C Trace Context [9], vendor neutral instrumentation provided by OpenTelemetry [10], and OTLP adopted by Jaeger [11].

An important question, however, remains unanswered: how much tracing should an application use? Tracing is not free as instrumentation adds work, exporting spans consumes CPU, memory, and network resources, and data volume grows with the sampling rate. Head sampling keeps overhead low but can discard an important trace before a request's behaviour is known [1], [8]; tail sampling preserves interesting traces but requires data to be retained and evaluated first, at additional cost [8]. This decision is typically made without a complete quantitative picture.

Tracing overhead has been measured in prior research. Serialization was found to account for a large part of overhead when OpenTelemetry, inspectIT, and Kieker were compared using microbenchmarks [12], [13], [17], and this investigation was extended to application-level microservice and serverless systems by Nõu [16]. Several questions, however, remain open. Although sampling-rate effects were examined in foundational work such as Dapper [1]. They remain insufficiently explored in modern application-level studies that simultaneously evaluate performance, telemetry volume, and trace quality. The detailed application-level study by Nõu et al. [16], for example, used 100% sampling throughout its experiments. Overhead and diagnostic value are rarely studied together, so no clear overhead-versus-quality relationship is available. Trace quality remains difficult to measure objectively [7], [14], since collected traces are rarely checked against an independent reference a trace cannot be used as the sole basis for determining what is missing from itself, making such measurement circular.

A further gap concerns architecture. Microservice applications have been shown to generate far more telemetry than equivalent monolithic applications [15], though the corresponding performance impact was not established. Because a request may cross several service boundaries in a microservice system but remain within one process in a monolith, tracing overhead may depend on decomposition as well as sampling rate. Within the studies reviewed here, there is limited evidence of a matched architectural evaluation of tracing overhead, telemetry volume, and trace quality.

These observations point to a central problem: a tracing configuration should be judged not only by whether it produces traces, but by what it costs and what it provides. This motivates an evaluation of tracing as a cost–quality trade-off across sampling rates, measured against an uninstrumented baseline, with trace quality assessed against an independent, code-derived operational inventory parsed using JavaParser [30] for Java and Tree-sitter [31] for JavaScript and with matched monolithic and microservice implementations used to isolate the effect of architecture, following established principles of rigorous performance evaluation [23]–[28].

The next section presents the literature review.

II. Literature Review

The literature review is organized as follows. It begins with the foundations of distributed tracing and its relationship to microservices, followed by an examination of sampling economics, instrumentation overhead, and trace quality. The section then discusses the architectural context and concludes with the measurement methodologies used in prior research.

A. Foundations of Distributed Tracing

Distributed tracing began with Google's Dapper. Dapper introduced a simple model for following requests through a system. A request was represented as a tree of spans. Each span was connected to its parent. Sampling was used to control the amount of collected data. Dapper also measured the cost of tracing itself [1]. This made performance overhead an important part of tracing research. Later systems extended this basic idea. Pivot Tracing allowed engineers to attach additional information, called “baggage,” to requests as they moved between services [6]. This provided greater flexibility but also introduced some additional overhead. Pivot Tracing extended this model by allowing dynamic monitoring queries and propagating contextual information across component and machine boundaries [6]. Its evaluation on the Hadoop stack demonstrated cross-tier analysis and low execution overhead, showing that tracing can support more flexible forms of distributed diagnosis.

Canopy focused on another challenge. At Facebook's scale, collecting trace events was only the first step. The events also had to be processed and presented in a useful form [5]. Sambasivan raised a more basic question: how much information should a tracing system collect, and how accurately should it represent application behaviour [7]? They highlighted the importance of this question, but did not provide a numerical measurement.

Parker later provided a practical guide to tracing concepts and trade-offs [8]. Two important standards now support modern tracing. W3C Trace Context provides a common way to pass trace information between

services [9]. OpenTelemetry provides a common framework for collecting and exporting telemetry data [10], [11].

B. Microservices as the Architectural Backdrop

Distributed tracing developed alongside microservices. Fowler, Lewis, and Newman described microservices as small and independently deployable services organised around business capabilities [2], [3]. Dragoni later reviewed the development of the microservice architecture [4]. They also identified several operational challenges, including observability. These studies did not directly measure tracing overhead. However, they explain an important architectural difference. A request in a monolith may remain inside one application. The same request in a microservice system may cross several service boundaries. This difference can affect tracing cost. Therefore, a tracing overhead measured in a monolithic application cannot automatically be assumed to apply to a microservice application.

C. Sampling and Its Economics

Sampling is one of the main ways to control tracing cost. Head sampling makes the decision at the beginning of a request. It is simple and inexpensive [1], [8]. However, it may discard an important request before its behaviour is known. Tail sampling makes the decision after the trace has been collected. This allows the system to keep unusual or failed traces [8]. The trade-off is higher resource usage because the trace information must be retained before the final decision is made.

Both approaches are well established. However, their performance effects have received less attention. Nõu conducted one of the most realistic tracing overhead studies, but used 100% sampling throughout their experiments [16]. This made the tracing cost easier to observe. At the same time, it left the effect of different sampling rates largely unexplored.

D. Instrumentation Overhead and Performance Cost

Reichelt, Kühne, and Hasselbring studied OpenTelemetry, inspectIT, and Kieker using controlled microbenchmarks [12]. Their experiments covered five different machines. They found that about 90% of the observed overhead was related to data serialization before export. Their later work investigated configurations that could reduce this cost [13]. Yang also benchmarked a newer version of Kieker [17]. These studies provided useful information about tracing overhead. However, they mainly used microbenchmarks rather than complete applications.

Nõu, Talluri, Iosup, and Bonetta moved closer to real-world applications [16]. They studied four web frameworks. These included Java Spring, Go, Python Flask, and Node.js. They used OpenTelemetry and Elastic APM. Their results showed substantial performance effects. Throughput decreased by 20% to 80%, depending on the framework. Latency increased by as much as 179% in the worst case. Serverless functions were affected strongly because tracing introduced a fixed cost to short-running tasks.

The researchers also examined where the overhead came from. Configuration and data export were major contributors. This finding was consistent with the earlier results from Reichelt. However, the study used only 100% sampling. It also did not measure the storage volume produced by tracing.

Gomes examined this other side of the problem [15]. They showed that microservice applications can produce much more telemetry than equivalent monolithic applications. Their work focused on telemetry storage rather than runtime overhead.

As a result, two important questions remain separated. One concerns the runtime cost of tracing. The other concerns the amount of telemetry produced. Few studies connect these two questions.There is also a major architectural gap. Previous studies generally do not trace the same application in both monolithic and microservice forms. Therefore, the effect of architecture itself is difficult to isolate.

E. Trace Quality and Completeness

Tracing performance is only one part of the problem. The quality of the collected traces is equally important. Bento reviewed tools used to analyse and understand traces [14]. They identified common challenges such as large amounts of data, limited tooling, and application-specific configuration. However, their work largely assumes that the collected traces are already reliable. It does not measure how much of the application's actual behaviour appears in the trace. This creates an important measurement problem. A trace cannot easily be judged only by looking at the trace itself. An independent description of the expected application behaviour is needed.

Instrumentation methods can also affect trace quality. For example, a Java agent can automatically instrument bytecode. A Node.js library may instead instrument selected modules during application startup. These approaches may observe different parts of an application. Trace context can also be lost when requests cross asynchronous boundaries. Such losses may not be obvious from the final trace.

The existing overhead studies do not normally measure these quality issues. Nõu, for example, focused on performance rather than trace completeness or correctness [16]. Therefore, tracing overhead and trace quality have largely been studied as separate problems.

F. The Architectural Context

The wider microservice literature also highlights observability as an important concern. Systematic reviews identify observability as one of the challenges faced by microservice practitioners [21], [22]. However, identifying a concern does not explain its measurable cost. Rodrigues, Rito Silva, and Avritzer provided another important observation [20]. They showed that comparisons between monolithic and microservice systems can depend strongly on how the workload is deployed. Zhao, De Matteis, and Bogner came closer to a controlled architectural comparison [19]. They studied how microservice granularity affected a monitoring metric. However, their metric was not tracing overhead or trace quality. DeathStarBench is another widely used benchmark suite [18]. It provides realistic microservice workloads. However, it does not provide a matched monolithic version of the same applications.

This creates a limitation for architectural comparison. Without equivalent monolithic and microservice versions, it is difficult to attribute differences specifically to the architecture.

G. Measurement Methodology in Prior Work

Reliable performance measurement requires careful experimental design. Jain provided foundational guidance for performance analysis [24]. Georges described important practices for fair Java benchmarking [23]. Schroeder, Wierman, and Harchol-Balter discussed the differences between open and closed workload models [27]. Statistical methods are also important. Arcuri and Briand [25] and Romano [26] discussed approaches for analysing experimental results. Nõu followed several of these principles [16]. They calibrated their workload to approximately 75% CPU utilisation. They repeated experiments under controlled conditions. Their serverless experiments also used 10,000 repetitions. Reichelt controlled the hardware environment across five machines [12], [13]. These practices helped reduce the influence of unrelated factors. However, many tracing studies provide limited information about confidence intervals and effect sizes. This makes it harder to understand whether reported differences are practically meaningful. Felter compared containers with virtual machines [28]. Their findings support the use of containers in controlled experiments because container overhead is relatively small. This allows researchers to focus more directly on the cost of the system being studied. For workload generation, k6 has also become a widely used tool for performance testing [29].

H. Serverless and Task-Based Tracing

Serverless tracing provides another perspective on tracing overhead. Nõu found that the effect of tracing depends strongly on task duration [16]. Short-running functions experienced latency increases of up to 175%. Longer-running functions experienced a much smaller increase of about 6.7%. The difference occurs because the fixed cost of tracing becomes less important as the task performs more work. This finding is important beyond serverless systems. It shows that tracing overhead is not necessarily a single fixed value. It can depend on how an application executes. Most microservice studies do not examine this issue. Benchmarks such as DeathStarBench mainly focus on request-based workloads [18].

Future studies therefore need to consider workload and task duration when measuring tracing overhead. A single overhead value may not accurately represent different application architectures or execution patterns.

I. Summary of Reviewed Literature

The reviewed literature shows that distributed tracing has evolved from foundational request-tracking systems such as Dapper into a broader observability mechanism for understanding distributed applications [1]. Later systems extended tracing through richer contextual information and large-scale trace processing [5], [6], while W3C Trace Context and OpenTelemetry established common mechanisms for trace propagation and telemetry collection [9], [10]. The literature also highlights the importance of architecture, as microservice applications introduce multiple service boundaries that can affect instrumentation, context propagation, and telemetry generation [2]–[4]. Sampling is widely recognised as a mechanism for controlling tracing overhead and telemetry volume [1], [8], but existing studies provide limited evidence on how different sampling rates simultaneously affect runtime performance, trace quality, and telemetry volume. Studies by Reichelt et al. [12], [13], Yang et al. [17], and Nõu et al. [16] demonstrate that tracing overhead can vary with instrumentation technology, configuration, application characteristics, and workload, while Gomes et al. [15] show that tracing can also generate substantial telemetry storage requirements. Trace-analysis research has examined challenges associated with processing and interpreting large trace datasets [14], but comparatively less work quantitatively measures how completely collected traces represent the expected application behaviour. Existing benchmarking literature further emphasises the

importance of controlled workloads, repeated measurements, consistent execution environments, and appropriate statistical analysis [23]–[29]. Overall, the literature provides strong foundations for distributed tracing, sampling, performance evaluation, and trace analysis, but these dimensions have largely been studied separately. In particular, limited work evaluates **runtime overhead, sampling effects, trace quality, and telemetry volume together**, while relatively few studies compare equivalent applications implemented in both monolithic and microservice architectures under comparable workloads. These gaps motivate an integrated evaluation of OpenTelemetry instrumentation that considers performance overhead, sampling trade-offs, trace quality, and telemetry volume across monolithic and microservice architectures.

The next section contains the comparison related to the different factors such as storage cost, sampling rate between various studies.

III. Comparative Analysis

Table I contains the comparison of the five questions: did the study measure a full application or just a microbenchmark, did it vary sampling rate, did it measure storage, did it check trace quality against something independent, and did it compare a monolith to microservices. The answer to that last question, for every existing study, is no.

TABLE I : Comparison of Prior Observability-Cost Studies

Study	Year	Full App?	Sampling Verifed?	Storage Cost?	Quality vs Independent Ref?	Monolithic vs Microservices
Dapper [1]	2010	Production	Yes	Partial	No	No
Reichelt et al. [12]	2021	Microbenchmark	No	No	No	No
Reichelt et al. [13]	2023	Microbenchmark	No	No	No	No
Gomes et al. [15]	2024	Yes	Yes	Yes	No	No
Nõu et al. [16]	2025	Yes	No (fixed 100%)	No	No	No
Proposed direction	2026	Yes (multi-app)	Yes (multi-rate)	Yes	Yes (static inventory)	Yes

Table I lines up Dapper, Reichelt et al. (2021, 2023), Gomes et al., and Nõu et al. against five questions: full-application vs. microbenchmark, sampling varied, storage measured, quality checked against an independent reference, and monolith-vs-microservice comparison. Every existing study answers "no" to the last question, and most answer "no" to several others. Only the proposed direction checks all five boxes. The storage cost refers specifically to telemetry volume or backend storage generated under the experimental workload, rather than general resource or network measurements.

Table II compares the tools and techniques themselves, what drives their overhead, what they are good at, and where they fall short.

TABLE II : Frameworks, Instrumentation Mechanisms, and Sampling Strategies

Framework / mechanism	Main overhead driver	Strengths	Limitations
OpenTelemetry Java agent [10]	Serialization / export [12]	Method-level detail; no source changes	Weaving cost not separated from export cost
OpenTelemetry Node.js [10]	Similar serialization cost, less studied	Simple setup; broad package coverage	Coarser visibility; async context can be lost
inspectIT / Kieker [12], [17]	Serialization (~90% of overhead) [12]	Long research lineage	Microbenchmark-only in most evaluations
OpenTelemetry multi-framework [16]	Configuration & export dominate	Real cross-framework evidence	Sampling fixed at 100%; no storage data
Elastic APM [16]	Export/serialization; cold-start-dependent	Directly compared to OpenTelemetry	19–80% throughput / up to 175% latency

Head sampling [1], [8]	Negligible per-trace cost	Cheap, simple	Drops interesting traces at random

Table II compares the tools, OpenTelemetry (Java and Node.js), inspectIT/Kieker, Elastic APM, and head vs. tail sampling by their main overhead driver, strengths, and limitations. It shows that serialization and export consistently dominate overhead across tools, while sampling strategies trade cost against the risk of losing interesting traces.

Table III zooms into the most detailed dataset available, Nõu framework-level numbers, because it speaks directly to how much instrumentation mechanism alone can matter.

TABLE III : Framework-Level Overhead Evidence Reported by Nõu et al. [16] (100% Sampling)

Framework	Throughput overhead (OTel)	Throughput overhead (Elastic APM)	Median latency overhead
Java Spring	~19.6%	~22.6%	Lowest of the four
Go http	~38.8%	~34.1%	Moderate
Python Flask	~35.0%	~46.3%	Moderate–high (up to 166%)
Node.js	~52.4%	~80.2%	Highest (up to 179%)

Table III reports throughput and latency overhead for Java Spring, Go http, Python Flask, and Node.js under OpenTelemetry and Elastic APM. Overhead ranges roughly four-fold across frameworks from ~20% to ~80% throughput loss showing that instrumentation cost depends heavily on the technology stack, not just on tracing itself.

The tables show a field where every individual piece has been measured carefully by the use of overhead, storage and sampling separately. But no single study measures all of them together. Table III adds a further detail that overhead varies by roughly four times just between frameworks, from 20% to 80% throughput loss, under otherwise identical conditions. That means any fair comparison between a monolith and a microservice implementation needs the technology stack fixed.

IV. Research Gaps

The literature review identifies five interconnected gaps that motivate the present study:

Gap 1: Sampling-rate gap. Existing microbenchmark studies provide detailed measurements of instrumentation overhead but use relatively controlled workloads [12], [13], [17], while the detailed application-level study by Nõu et al. evaluates tracing primarily under 100% sampling [16]. Consequently, there is limited empirical evidence on how a range of sampling rates affects the performance and trace characteristics of the same application under controlled conditions.

Gap 2: Cost–quality gap. Runtime overhead has been investigated in several studies [12], [13], [16], [17], while telemetry storage and volume have been examined separately [15]. These dimensions have rarely been evaluated together with trace quality under the same experimental conditions, leaving the relationship between tracing cost and the usefulness of the resulting traces insufficiently characterised.

Gap 3: Objective trace-quality gap. Trace quality and trace analysis have been discussed in prior work [7], [8], [14], but comparatively little work quantitatively evaluates trace completeness against an independent representation of the application's expected behaviour. This limits the ability to determine which application operations are actually captured by automatic instrumentation.

Gap 4: Architectural gap. Existing studies provide evidence about tracing in microservice applications and about performance differences between monolithic and microservice architectures, but the reviewed literature provides limited evidence from controlled experiments in which the same application is evaluated in both architectural forms under matched workloads. This makes it difficult to isolate the contribution of architectural style to tracing overhead, telemetry volume, and trace quality.

Gap 5: Instrumentation-mechanism gap. Existing work has compared tracing performance across different languages, frameworks, and observability technologies [16], but the specific contribution of the underlying instrumentation mechanism has received less systematic attention. In particular, the relative effects of mechanisms such as Java bytecode instrumentation and Node.js module-level instrumentation have not been clearly isolated under otherwise comparable application and workload conditions. Taken together, these gaps do not indicate that the individual dimensions of tracing have been ignored. Rather, they show that sampling, runtime overhead, telemetry volume, trace quality, architecture, and instrumentation mechanism have largely

been investigated as separate dimensions. The present study addresses this missing connection by evaluating these dimensions together under controlled and comparable experimental conditions.

V. Future Directions And Research Implications

The identified gaps motivate an integrated empirical study that evaluates distributed tracing across multiple dimensions rather than treating them independently. The study examines a real application under multiple sampling configurations rather than focusing only on 100% sampling, allowing the relationships among runtime overhead, telemetry volume, and trace quality to be evaluated together. It also evaluates equivalent monolithic and microservice implementations of the same application. So that the effect of architectural style can be examined under comparable workloads and experimental conditions. In addition, trace quality is evaluated against an independent, code-derived inventory of expected application operations rather than against the collected traces alone. This makes trace coverage and completeness quantitatively measurable. The objective is not to replace existing approaches, but to connect the dimensions examined separately in prior work within a single controlled experimental framework. By evaluating sampling, performance overhead, telemetry volume, trace quality, architecture, and instrumentation mechanisms together, the study provides a more comprehensive assessment of the practical costs and diagnostic value of OpenTelemetry-based distributed tracing.

VI. Conclusion

This survey addresses a central question in distributed tracing: what are the practical costs of tracing, and what diagnostic value does it provide in return? Runtime overhead has been measured, but often using controlled microbenchmarks or fixed sampling configurations. Telemetry volume and storage costs have also been investigated, but are rarely evaluated alongside runtime overhead and trace quality. Trace quality has been discussed extensively, but is less frequently evaluated against an independent representation of the application's expected behaviour. Similarly, architectural differences between monolithic and microservice systems have been studied, but relatively few studies evaluate equivalent applications under matched conditions while simultaneously measuring tracing performance, telemetry volume, and trace quality.

The gap in the existing literature is therefore not a lack of individual studies, but a lack of an integrated evaluation that connects these dimensions within a common experimental framework. Addressing this gap can provide a more complete understanding of the practical trade-offs associated with enabling distributed tracing. Such an evaluation can help practitioners assess not only the performance and telemetry costs of tracing, but also the extent to which the collected traces provide useful and complete information about application behaviour.

References

[1] B. H. Sigelman et al., “Dapper, a large-scale distributed systems tracing infrastructure,” Google Technical Report dapper-2010-1, 2010.

[2] M. Fowler and J. Lewis, “Microservices,” martinfowler.com, 2014.

[3] S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA, USA: O’Reilly Media, 2021.

[4] N. Dragoni et al., “Microservices: Yesterday, today, and tomorrow,” in Present and Ulterior Software Engineering, Cham, Switzerland: Springer, 2017, pp. 195–216.

[5] J. Kaldor et al., “Canopy: An end-to-end performance tracing and analysis system,” in Proc. 26th ACM SOSP, 2017, pp. 34–50.

[6] J. Mace, R. Roelke, and R. Fonseca, “Pivot Tracing: Dynamic causal monitoring for distributed systems,” in Proc. 25th ACM SOSP, 2015, pp. 378–393.

[7] R. R. Sambasivan et al., “So, you want to trace your distributed system?,” CMU Parallel Data Lab, Tech. Rep. CMU-PDL-14-102, 2014.

[8] A. Parker et al., Distributed Tracing in Practice. Sebastopol, CA, USA: O’Reilly Media, 2020.

[9] W3C, “Trace Context,” W3C Recommendation. https://www.w3.org/TR/trace-context/

[10] OpenTelemetry documentation. https://opentelemetry.io/docs/ (accessed Jul. 2026).

[11] Jaeger documentation (v2). https://www.jaegertracing.io/docs/ (accessed Jul. 2026).

[12] D. G. Reichelt, S. Kühne, and W. Hasselbring, “Overhead comparison of OpenTelemetry, inspectIT and Kieker,” SSP, CEUR Workshop Proc. vol. 3043, 2021.

[13] D. G. Reichelt, S. Kühne, and W. Hasselbring, "Towards solving the challenge of minimal overhead monitoring," Companion of ICPE, 2023.

[14] A. Bento et al., "Automated analysis of distributed tracing: Challenges and research directions," J. Grid Computing, vol. 19, no. 9, 2021.

[15] F. A. Gomes et al., "Impact of OpenTelemetry configuration on observability and telemetry storage cost," ADVANCE Workshop, 2024.

[16] A. Nõu, S. Talluri, A. Iosup, and D. Bonetta, "Investigating performance overhead of distributed tracing in microservices and serverless systems," Companion of ICPE'25, 2025, pp. 162–166. Full detail in A. Nõu, M.S. thesis, VU Amsterdam / UvA, Jan. 2025.

[17] S. Yang et al., "The Kieker observability framework version 2," Companion of ICPE, 2025, arXiv:2503.09189.

[18] Y. Gan et al., "An open-source benchmark suite for microservices," Proc. ASPLOS, 2019, pp. 3–18.

[19] Y. Zhao, T. De Matteis, and J. Bogner, "How does microservice granularity impact energy consumption and performance?," ICSA, 2025, arXiv:2502.00482.

[20] H. Rodrigues, A. Rito Silva, and A. Avritzer, "Performance comparison of monolith and microservice architectures," ECSA 2023, LNCS vol. 14590, 2024, pp. 185–199.

[21] S. Li et al., "Understanding and addressing quality attributes of microservices architecture," Information and Software Technology, vol. 131, 106449, 2021.

[22] M. Söylemez, B. Tekinerdogan, and A. Kolukısa Tarhan, "Challenges and solution directions of microservice architectures," Applied Sciences, vol. 12, no. 11, 5507, 2022.

[23] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," OOPSLA, 2007, pp. 57–76.

[24] R. Jain, The Art of Computer Systems Performance Analysis. New York, NY, USA: Wiley, 1991.

[25] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms," ICSE, 2011, pp. 1–10.

[26] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data," Florida AIR, 2006.

[27] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: A cautionary tale," NSDI, 2006, pp. 239–252.

[28] W. Felter et al., "An updated performance comparison of virtual machines and Linux containers," ISPASS, 2015, pp. 171–172.

[29] Grafana Labs, "k6 documentation." https://grafana.com/docs/k6/latest/ (accessed Jul. 2026).