AutoScope: Code Knowledge Enhanced Span-level Sampling for Distributed Tracing

YULUN WU, The Chinese University of Hong Kong, China
GUANGBA YU*, The Chinese University of Hong Kong, China
ZHIHAN JIANG, The Chinese University of Hong Kong, China
YICHEN LI, The Chinese University of Hong Kong, China
MICHAEL R. LYU, The Chinese University of Hong Kong, China

Distributed tracing is essential for observing microservices but incurs prohibitive storage costs. Existing solutions primarily rely on trace-level sampling, which indiscriminately discards entire traces based on probability or tail latency. This coarse-grained approach often loses critical off-the-path anomalies and fails to retain normal baselines required for comparative diagnosis.

To address these limitations, we introduce the concept of span-level sampling, a fine-grained paradigm that retains only essential spans rather than full traces. However, we identify a critical challenge in this new approach where naive span retention leads to structural ambiguity, as the removal of intermediate spans disrupts the graph topology and renders the remaining data useless for analysis. To resolve this, we propose AutoScope , a novel framework that leverages static code knowledge to enable structurally safe span-level sampling. AutoScope constructs a *Bridged Call Site Control Flow Graph (CSCFG)* to map runtime spans to static execution logic. By identifying Dominant Span Sets (DSS), it exploits the "Sample One, Infer All" property to enable aggressive compression while preserving the structural skeleton of every request. Extensive evaluations on two benchmark microservice systems demonstrate that AutoScope achieves an 80.2% storage reduction while maintaining a state-of-the-art faulty span coverage of 98.1%. Furthermore, the reconstructed traces enhance downstream Root Cause Analysis (RCA) tasks, improving the Mean Reciprocal Rank (MRR) by an average of 8.3%.

CCS Concepts: • **Software and its engineering** → **Cloud computing**; • **General and reference** → **Reliability**; **Performance**.

Additional Key Words and Phrases: Distributed Tracing, Trace Sampling, Static Analysis

ACM Reference Format:
Yulun Wu, Guangba Yu, Zhihan Jiang, Yichen Li, and Michael R. Lyu. 2025. AutoScope: Code Knowledge Enhanced Span-level Sampling for Distributed Tracing. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (July 2025), 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Modern software architectures have evolved into distributed microservice systems [34, 55], making distributed tracing an essential observability tool for understanding application behavior and

*Corresponding author.

Authors' Contact Information: Yulun Wu, The Chinese University of Hong Kong, Hong Kong SAR, China; Guangba Yu, The Chinese University of Hong Kong, Hong Kong SAR, China; Zhihan Jiang, The Chinese University of Hong Kong, Hong Kong SAR, China; Yichen Li, The Chinese University of Hong Kong, Hong Kong SAR, China; Michael R. Lyu, The Chinese University of Hong Kong, Hong Kong SAR, China.

ACM 1557-7392/2025/7-ART
https://doi.org/XXXXXXX.XXXXXXX

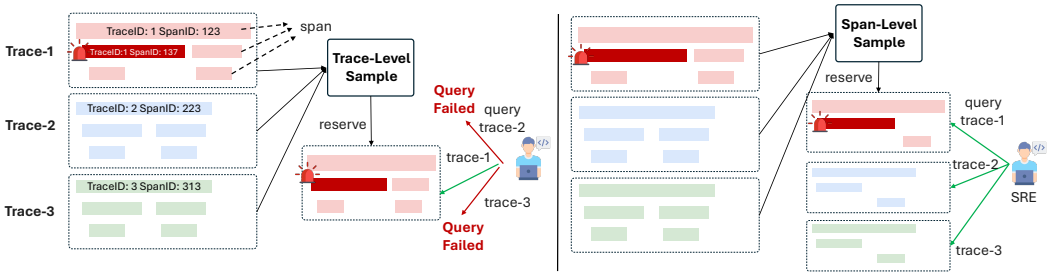

Fig. 1. Comparisons between trace-level and span-level sampling.

performance at scale [13, 18, 39]. As shown the *Trace-1* in Fig. 1, by capturing fine-grained spans along request paths across services, traces create a comprehensive execution path that includes service transitions, time distributions, and context propagation. These tracing systems enable Site Reliability Engineers (SREs) to effectively profile system performance [20, 54, 56], identify performance anomalies [12, 31, 46], and diagnose root causes [30, 33, 47]. Therefore, many industry leaders have embraced various tracing solutions, including OpenTelemetry [36], Skywalking [40], and Jaeger [22], demonstrating the technology's widespread adoption [6, 16, 54].

Although distributed traces offer valuable information for system analysis, their extensive volume and the resulting storage requirements create significant obstacles. Alibaba's e-commerce platform generates between 18.6 and 20.5 pebibytes (PB) of trace data each day [18]. This immense amount arises because developers seek to capture the full spectrum of application behaviors to facilitate the diagnosis of emerging issues. However, persistently storing such a large amount of trace data incurs substantial operational overhead. To address these challenges, trace-level sampling techniques [15, 19, 21, 27, 53] have been devised to selectively retain traces relevant to anomalous behavior [12, 31, 47].

As shown in Fig. 1-(a), the primary mechanism of trace-level sampling involves initially identifying the traces to be sampled and subsequently retaining only those selected traces while completely eliminating the remainder, a process we call the "1 or 0" strategy. These techniques are generally classified into head sampling [24, 39] or tail sampling [19, 21, 27], depending on when the sampling decision occurs and the criteria applied. However, this approach of wholly discarding unsampled traces reveals significant shortcomings in practical scenarios. Existing research [18] indicates that SREs may still need to access these discarded traces, as the traits of traces that require analysis are often unpredictable, leading to a query miss rate of up to 27.17%. Moreover, such trace query failures can substantially hinder diagnostic approaches based on comparing normal and abnormal traces [12, 31, 47].

To address the limitations of trace-level sampling and enhance sampling flexibility, we introduce the concept of span-level sampling. The fundamental observation behind span-level sampling is that most spans within a trace are irrelevant to explaining performance variations under scrutiny [11, 24, 55]. For example, a study of Alibaba's trace data indicates that 90% of spans contribute little meaningful information, while only 10% are critical for diagnosing performance issues [32]. This suggests the potential to balance trace cost and utility at the span level by preserving spans that aid fault diagnosis while discarding those that do not.

However, achieving a balance between trace cost and utility at the span level is far from straightforward. Transitioning from the trace-level sampling approach, an intuitive strategy might involve analyzing variations in span duration and retaining spans that significantly deviate from typical latency patterns. Nevertheless, we observed that relying solely on duration data overlooks critical invocation details within the trace, such as the specific services or functions traversed. For example,

as illustrated in Fig. 4-(a) and (b), *Trace-1* and *Trace-2* represent different request types, but after sampling, their trace structure becomes identical, making their request types indistinguishable, thus affecting downstream diagnostic tasks (§ 3).

To overcome the structural ambiguity inherent in purely latency-based approaches, we propose AutoScope , the first code-knowledge enhanced span-level sampling framework. The key insight of AutoScope is that while dynamic latency varies, the underlying execution logic defined by the source code remains invariant. By aligning dynamic traces with static code structures, we can "compress" the trace without losing its topological context.

Specifically, AutoScope bridges the gap between static analysis and runtime tracing. It first constructs a Bridged Call Site Control Flow Graph (CSCFG), which unifies intra-service control flows (via static analysis) and cross-service dependencies (via runtime topology). Based on this graph, AutoScope partitions the trace into Dominant Span Sets (DSS) which are logical units where the execution of one “dominant” span mathematically implies the execution of its subordinates. This allows for a “Sample One, Infer All” strategy: by retaining only the dominant span and significant anomalies (identified via a robust Z-score estimator), AutoScope can reconstruct the full trace skeleton with minimal storage.

We evaluated AutoScope on two widely-adopted microservice benchmarks, TrainTicket [43] and SocialNetwork [14]. The results demonstrate a significant breakthrough in the trade-off between cost and utility. AutoScope reduces storage overhead by 80.2%, comparable to aggressive trace-level sampling, which preserves the structural records of all requests. Consequently, it achieves a faulty span coverage of 98.1%, effectively eliminating the “missing baseline” problem common in traditional sampling. Extensive experiments on downstream tasks further confirm this high fidelity: AutoScope boosts the accuracy of four state-of-the-art Root Cause Analysis (RCA) frameworks by an average of 8.3%, proving that span-level sampling can serve as a superior drop-in replacement for existing solutions.

In summary, this paper makes the following contributions:

- **New Sampling Paradigm:** We introduce the concept of *span-level sampling*, a fine-grained approach that decouples data retention from the trace unit, allowing for the precise preservation of critical execution fragments while discarding redundancy (§ 3).
- **Code-Enhanced Methodology:** We design AutoScope , which innovatively leverages static code analysis to guide dynamic sampling. By constructing Bridged CSCFGs and DSS, AutoScope ensures structural completeness and anomaly retention without the overhead of full logging (§ 4).
- **Extensive Evaluation:** We demonstrate the superiority of AutoScope through rigorous experiments. It achieves an 80.2% compression ratio while maintaining 98.1% fault coverage, significantly outperforming traditional trace-level strategies in both sampling efficiency and downstream RCA utility (§ 5).

The paper is organized as follows: Section 2 provides the background on distributed tracing and static analysis. Section 3 introduces the concept of span-level sampling and highlights the challenges of ambiguity that motivate our code-enhanced approach. Section 4 details the design and implementation of AutoScope . Section 5 describes the experimental setup, followed by a comprehensive analysis of the evaluation results. Section 6 discusses practical applications and threats to validity. Finally, Section 7 reviews related work and Section 8 concludes the study.

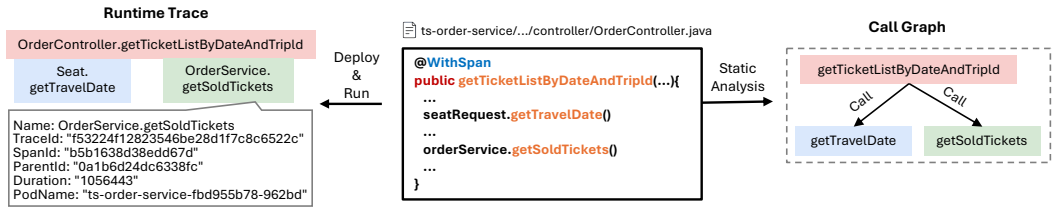

Fig. 2. The correspondence between user code, function call graphs, spans, and traces.

2 Background

2.1 Distributed Tracing

Distributed tracing [39] serves as a fundamental observability mechanism for capturing causal relationships across process boundaries in distributed environments. By tracking the propagation of requests across diverse microservices, it facilitates the inference of system states and enables the precise identification of code regions responsible for performance bottlenecks [20, 54].

To elucidate the intrinsic mapping between static source code and runtime traces, we utilize a case study from TrainTicket [43], a widely adopted microservice benchmark. We adhere to the instrumentation standards of OpenTelemetry [36], the prevailing trace framework. As shown in Fig. 2, the static definition of `getTicketListByDateAndTripId()` invokes child functions `getTravelDate` and `getSoldTickets`. This static call hierarchy (Call Graph, right) is mirrored by the trace structure (Trace, left) captured during runtime execution.

Span. A **Span** represents the atomic unit of work within a trace, encapsulating a specific request-response interaction or function execution within a service instance. In Fig. 2, the `OrderService` function `getTicketListByDateAndTripId()` and its sub-functions are explicitly instrumented using the `@WithSpan` annotation [1]. This instructs the tracing agent to generate corresponding spans, visualized as blocks containing essential metadata (e.g., Span ID, timestamp, duration). Since spans map one-to-one with instrumented function calls, they serve as the runtime counterparts to static nodes in the function call graph.

Trace. A **Trace** constitutes a Directed Acyclic Graph (DAG) of spans, representing the end-to-end execution path of a single request. As depicted in Fig. 2, the trace aggregates spans from the three functions, preserving their causal dependencies. For instance, the parent-child relationship between the span `OrderController.getTicketListByDateAndTripId()` and `Seat.getTravelDate()` signifies a direct invocation in the underlying code. Consequently, a trace effectively acts as a runtime instantiation of the static execution flow; the logical structure of the application code strictly dictates the organization and causality of the generated spans.

2.2 Trace Sampling

Trace sampling is a pivotal mechanism for alleviating the storage and computational burdens in production environments, where capturing every request is often infeasible due to the sheer volume of data [19, 21, 27, 39]. Currently, sampling strategies deployed in industry, such as those in Jaeger [22] and OpenTelemetry [36], predominantly rely on *head-based sampling*. This approach employs a uniform random strategy (e.g., retaining 5% of traffic) where the decision to record a trace is made at the request's ingress. While efficient, head-based sampling ignores the runtime behavior of requests, often discarding traces of high analytical value [24, 39].

[1] https://opentelemetry.io/docs/zero-code/java/agent/annotations/

To overcome the blindness of head-based approaches, *tail-based sampling* [19, 21, 27] postpones the sampling decision until the request concludes. By leveraging complete execution information, such as latency or error codes, tail-based strategies prioritize retaining anomalous or significant traces.

Trace-level vs. Span-level Sampling. In this paper, we classify both head-based and tail-based approaches under the umbrella of ***Trace-level Sampling***. These methods adhere to a rigid **"1 or 0" strategy**: a trace is either fully retained or completely discarded. While effective for trace-granularity filtering, this binary approach lacks flexibility. To address this, we introduce ***Span-level Sampling***, a more granular paradigm that allows for the selective retention of specific spans within a trace. Figure 1 illustrates the structural distinction between these two paradigms. The specific motivation and implementation challenges of span-level sampling are detailed in Section 3.

2.3 Call-Site Control Flow Graph

Distributed traces are essentially dynamic instantiations of the application's static source code. To develop sampling strategies that leverage this deterministic logic, we must first bridge the domain gap between runtime traces and static code using static code analysis [10, 42]. The foundational representation in static analysis is the **Control Flow Graph (CFG)** [4], which models the execution flow within a function. As shown in the left panel of Fig. 3, a CFG consists of Basic Blocks (BBs) which are sequences of instructions executed in order connected by edges representing possible execution paths. Function calls connect these CFGs to form an Inter-Procedural Control Flow Graph (ICFG) [35]. In microservice architectures, these connections extend across service boundaries via RPC or HTTP calls [29].

However, a direct mapping from standard CFGs to traces is inefficient due to a granularity mismatch. Standard CFGs capture fine-grained instruction details (e.g., variable assignments like `int uid = ...` or boolean checks like `bool isVIP = ...` in Fig. 3), whereas traces primarily record high-level function invocations (Spans). To align static code knowledge with trace structure, we adopt the **Call-Site CFG (CSCFG)** [44]. Unlike traditional CFGs, a CSCFG preserves only the Basic Blocks containing function calls, effectively *pruning* non-tracing logic. As illustrated in the middle panel of Fig. 3, this representation transforms the code into a concise structural skeleton where each node (e.g., `User.verify`) acts as the static counterpart to a runtime span. This establishes a rigorous one-to-one correspondence between the static graph and the dynamic trace.

A crucial property of the CSCFG for sampling efficiency is the **Dominance Relation** [2]. If every path from the function entry to node B must pass through node A, then A *dominates* B. When two nodes A and B mutually dominate each other, meaning the execution of one deterministically implies the execution of the other. A and B form a **Dominant Span Set (DSS)**. For example, in Fig. 3, since `User.verify(uid)` and `Order.create()` execute sequentially without intervening branches, the presence of the former guarantees the latter. These two functions thus constitute a DSS (marked by the orange dashed box). This relationship is the theoretical foundation of AutoScope, allowing us to sample only one representative span (e.g., `Order.create`) to infer the entire set, thereby significantly reducing storage overhead. Note that `Notify.VIP()` is excluded from this DSS as it resides on a conditional branch.

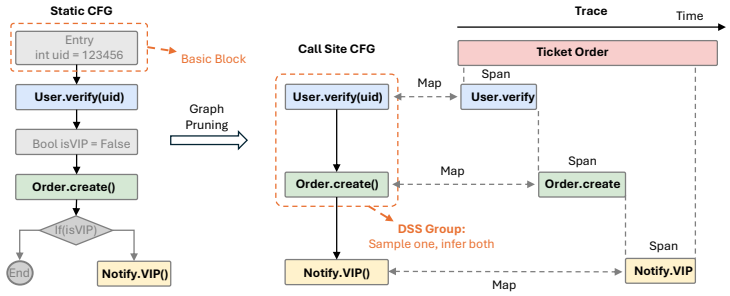

Fig. 3. The structural alignment between CFG, CSCFG, and Trace.

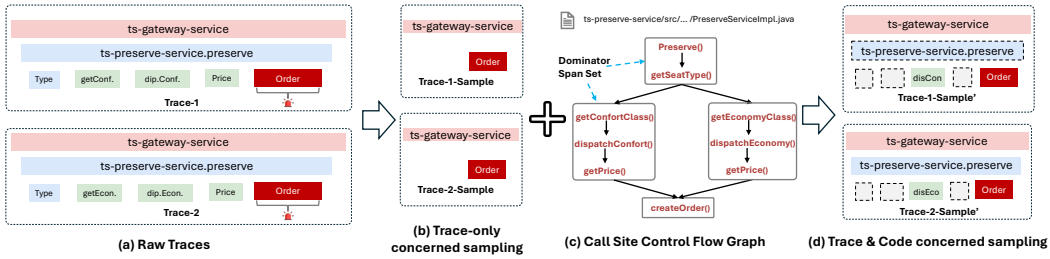

Fig. 4. An example of the importance of source code in span-level sampling.

3 Motivation

3.1 What is Span-level Sampling?

***Definition**: Span-level sampling is a fine-grained strategy that decouples data retention from the trace unit, selectively preserving specific spans within a distributed trace based on their diagnostic significance.*

In stark contrast to the rigid "1 or 0" decision-making of trace-level sampling, span-level sampling operates on the principle of **value-based retention**. Instead of discarding an entire trace due to storage constraints, this approach deconstructs the trace to distinguish between "signal" and "noise" at the span level. By retaining only critical segments—such as spans exhibiting high latency, error codes, or those belonging to key business paths—span-level sampling ensures that the system maintains a comprehensive record of *all* requests, albeit in a condensed form. For SREs, this granularity resolves the "all-or-nothing" dilemma: it substantially reduces storage overhead while preserving the essential context required for identifying anomalies and isolating failure domains. Consequently, span-level sampling maximizes the cost-utility ratio of observability data, supporting focused debugging without the burden of massive data redundancy.

Formally, let a distributed trace be represented as a set of spans $\mathcal{T} = \{s_1, s_2, \ldots, s_n\}$, where each s_i denotes a span unit. **Trace-level sampling** can be defined as a binary function $\Phi_{trace} : \mathcal{T} \rightarrow \{\mathcal{T}, \emptyset\}$, where:

$$\Phi_{trace}(\mathcal{T}) = \alpha \cdot \mathcal{T}, \quad \alpha \in \{0, 1\}. \tag{1}$$

Here, α is a boolean variable determined by a sampling probability or a specific rule (e.g., tail latency). This implies that storage cost is either $|\mathcal{T}|$ or 0.

Table 1. Comparison of retention capabilities between AutoScope and SOTA trace sampling approaches.

Method		Input Data			Retention Capability	
		Trace	Metric	Code	Normal Trace	Abnormal Trace
Trace-Level Sampling	Random (5%) [22]	✔			Partial	Negligible
	Perch [26]	✔			Negligible	Comprehensive
	Sifter [27]	✔			Negligible	Comprehensive
	TraceMesh [9]	✔			Negligible	Comprehensive
	TraStrainer [19]	✔	✔		Partial	Comprehensive
Span-Level Sampling	**AutoScope (Ours)**	✔		✔	**Comprehensive**	**Comprehensive**

Note: Negligible (< 1%), Partial (1%–20%), Substantial (20%–80%), Comprehensive (> 80%). Coverage percentages refer to the retention rate of normal/anomalous traces.

In contrast, **span-level sampling** is defined as a subset selection function $\Phi_{span} : \mathcal{T} \rightarrow \mathcal{T}'$, where $\mathcal{T}' \subseteq \mathcal{T}$. The objective is to determine an indicator function $\mathbb{I}(s_i) \in \{0, 1\}$ for each span:

$$\Phi_{span}(\mathcal{T}) = \{s_i \in \mathcal{T} \mid \mathbb{I}(s_i) = 1\}. \tag{2}$$

The goal of span-level sampling is to minimize the storage cost $|\mathcal{T}'|$ (i.e., $|\mathcal{T}'| < |\mathcal{T}|$) while maximizing the preservation of diagnostic information (e.g., retaining causal structures and anomalies).

3.2 Necessity of Span-level Sampling

Figure 1-(a) illustrates the inherent limitation of trace-level sampling. Consider a scenario with three traces: *Trace-1* exhibits a performance anomaly (caused by *Span-137*), while *Trace-2* and *Trace-3* represent normal execution paths. Conventional tail-based sampling strategies [19, 21, 27] rigidly enforce a "keep or drop" decision, typically retaining only the anomalous *Trace-1* while discarding the normal ones. However, this binary elimination creates a critical **observability gap**. In complex fault diagnosis scenarios, such as "off-the-path" resource contention [45], the root cause often lies in shared resource interference rather than direct call errors. To identify such issues, SREs require *Trace-2* and *Trace-3* as baselines for **differential diagnosis** [31, 47]. The discarding of these normal traces severs the necessary context, thereby hampering effective root cause analysis.

Table 1 empirically quantifies this limitation by comparing the retention capabilities of state-of-the-art approaches. Although tail-based methods (e.g., Perch [26], Sifter [27], TraceMesh [9]) excel at capturing anomalies (coverage > 80%), they exhibit a severe deficiency in retaining normal traces, preserving negligible amounts (< 1%). This skewed retention policy results in a high **Query Miss Rate** during operational troubleshooting. Empirical studies on large-scale industrial systems reveal that query miss rates can reach 27.17% [18], significantly undermining the reliability of trace-based diagnosis. This non-trivial miss rate highlights the urgent need for a sampling paradigm that balances the retention of both anomalous and normal contexts.

In contrast, Figure 1-(b) demonstrates the efficacy of span-level sampling. By shifting the granularity from the trace level to the span level, this approach decouples data retention from the request unit. It allows for the storage of the structural skeleton and critical spans of *all* traces, including *Trace-2* and *Trace-3*, without exceeding the storage budget. Consequently, span-level sampling bridges the observability gap, offering a novel perspective on the balance of cost and utility. Its key structural advantages include:

- **Granular Selectivity**: By filtering at the span level, the system can retain only critical paths or spans relevant to specific anomalies, avoiding the rigid "1 or 0" discarding of entire traces.

- **Universal Trace Accessibility**: Unlike trace-level sampling that completely deletes unselected traces, span-level sampling retains a skeletal form of *every* trace. This ensures that SREs can retrieve the context of any request ID, effectively eliminating "trace not found" query failures.
- **Optimized Cost-Utility Ratio**: Span-level sampling maintains a comprehensive system view (High Utility) while strictly adhering to storage constraints (Low Cost).

Insight 1: Current trace-level sampling creates a dichotomy where retaining anomalies comes at the expense of discarding normal baselines, creating an observability gap for differential diagnosis. Span-level sampling resolves this by offering granular retention, preserving the structural context of all traces to maximize diagnostic utility within storage limits.

3.3 Challenges in Span-level Sampling

Implementing span-level sampling necessitates a paradigm shift from trace-wide retention to granular span selection. A naive approach is *latency-based filtering*, which retains only spans exhibiting significant performance deviations (e.g., high latency). However, this strategy introduces a critical side effect: the loss of execution context.

Challenge 1: Contextual Ambiguity in Fault Localization. Relying solely on performance metrics (e.g., latency) for span selection often discards the clues required to differentiate distinct business logic, leading to diagnostic ambiguity. For example, consider the TrainTicket scenario in Fig. 4-(a). Two distinct user requests (i.e., purchasing a comfort seat versus an economy seat) trigger different underlying processing logic but share a common service call, `ts-preserve-service`. In this scenario, a performance issue occurs within the shared `createOrder()` method, causing excessive latency. If the sampling strategy blindly retains only the high-latency span (`createOrder()`), as shown in Fig. 4-(b), the resulting partial trace becomes indistinguishable between the two business cases. Consequently, an SRE analyzing the sampled data faces a dilemma: *Did the failure originate from the premium validation logic or the standard query path?* This ambiguity significantly hinders effective root cause analysis, as the differentiating contexts (e.g., spans unique to comfort processing) are discarded.

Insight 2: Span-level sampling driven solely by performance anomalies creates contextual ambiguity, stripping away the causal chain required to distinguish between different execution paths that share common bottlenecks.

Challenge 2: Structural Blindness of Purely Dynamic Tracing. The root cause of the aforementioned ambiguity lies in the inherent limitation of runtime trace data: it records *what happened* but lacks the knowledge of *structural logic*.

Traces are inherently stochastic and driven by variable user behaviors. A collection of traces represents only a subset of executed paths, which does not capture the comprehensive control flow of the application. In the "Comfort vs. Economy" example, distinguishing the failure requires more than observing the slow node; it requires understanding the Control Flow Control (CFG). SREs need to know whether the execution traversed the path containing `getComfortClass()` or the path containing `getEconomyClass()`. However, purely dynamic tracing lacks this global perspective. It cannot inherently identify that `getComfortClass()` is a critical discriminator span that determines the request type. Without this structural prior knowledge, sampling algorithms remain blind to critical logic branches, inevitably leading to the omission of essential diagnostic contexts.

Insight 3: Trace data alone is insufficient to reconstruct the program's critical paths due to its **structural blindness**. To resolve ambiguity, sampling strategies must transcend dynamic traces and incorporate static knowledge of the application's control flow.

The Opportunity: Code-Enhanced Trace Sampling. The evolution of static analysis tools [38, 42] provides a compelling solution to these challenges. With source code increasingly accessible in DevOps pipelines, static analysis allows us to extract the structural blueprint of microservices. As illustrated in Fig. 4-(c), by analyzing the code's control flow, we can statically discern that the comfort and economy classes occupy distinct execution branches. This prior knowledge empowers the sampling algorithm to identify structural discriminator spans (e.g., `getComfortClass()` and `dispatchComfort()`) that are critical for disambiguation. As shown in Fig. 4-(d), retaining these informative spans alongside the anomalous ones resolves the ambiguity, yielding a refined trace that preserves both the symptom (latency) and the cause (context). This observation motivates our proposal: leveraging code insights to guide span-level sampling.

4 Span-level Sampling Method: AutoScope

4.1 Overview

Figure 5 presents the system architecture of AutoScope . To fundamentally resolve the contextual ambiguity and structural blindness identified in Section 3.3, AutoScope moves beyond the limitations of purely dynamic tracing. Instead of treating traces as isolated data points, we anchor them to the static code structure. The core idea of this work is to **leverage the deterministic execution logic extracted from static source code as prior knowledge to eliminate redundancy in dynamic runtime traces**, thereby achieving precise, structure-aware span-level sampling. To implement this, AutoScope operates in two distinct phases: *Offline Knowledge Construction* and *Online Span-level Sampling*.

Phase I: Offline Knowledge Construction. The objective of this phase is to build a global structural blueprint of the microservice system. It begins with *Service-Internal Analysis*, where a static analyzer (e.g., Soot) parses the source code to construct local CSCFGs. Since standard static analysis cannot resolve network boundaries, we introduce a *Cross-Service Bridging* mechanism. By analyzing protocol-specific definitions (e.g., RPC IDLs or HTTP annotations), this step links isolated local graphs into a global bridged CSCFG. Finally, AutoScope performs *DSS Identification* on this global graph. By computing dominance relationships, it statically groups mutually dependent functions into DSSs. These identified CSCFG and DSSs constitute the core "Code Knowledge" that guides the online phase.

Phase II: Online Span-level Sampling (Bottom Layer). This phase executes the sampling logic on the live data stream. As raw traces flow in during trace collection, AutoScope performs *Trace Partition*. Instead of treating a trace as a black box, it utilizes the pre-identified DSSs from Phase I to decompose each trace into inferable structural units. Subsequently, the *Span Selection* module applies a topology-aware cascade strategy to determine retention. It calculates priorities based on the structural importance of each DSS and runtime anomaly scores. The result is a set of highly compressed fragments stored in the tace database.

To ensure compatibility with SRE workflows, AutoScope includes an *On-Demand Span Reconstruction* module. When a query is triggered, this module retrieves the fragmented spans and combines them with the Code Knowledge. By leveraging the mutual dominance property of DSS, it restores the missing structural skeleton and imputes attributes for discarded spans, presenting a complete trace view to the user.

4.2 Phase I: Offline Knowledge Construction

The primary objective of this phase is to construct a global structural blueprint of the microservice system and identify the DSS to guide the online sampling process.

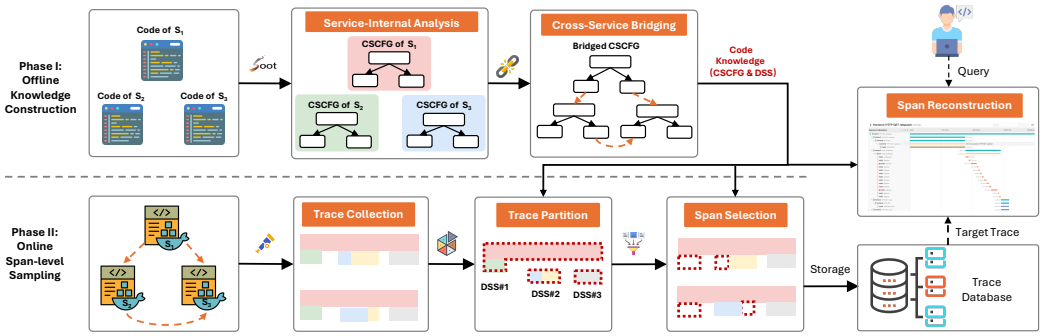

Fig. 5. The Overview of AutoScope .

4.2.1 Service-Internal Analysis. We first employ static analysis tools (e.g., Soot [41], Angr [5]) to parse the bytecode (e.g., `.class` files) or binaries of individual microservices. This step extracts the intra-procedural control flow and method invocation relationships at the basic block level. By filtering out non-tracing logic (e.g., arithmetic operations), we generate independent *CSCFG fragments* for each service, where nodes specifically represent function call sites.

4.2.2 Cross-Service Bridging. Constructing a global CSCFG requires bridging these independent service fragments. However, general-purpose static analysis tools are inherently confined to process boundaries, treating remote calls merely as opaque library invocations (as shown in the "Static Analyze" box in Fig. 6). To overcome this isolation, we introduce a Protocol-Enhanced Bridging Strategy.

Extensive research on microservice architectures [3, 25] confirms that the inter-service communication landscape is overwhelmingly dominated by two fundamental paradigms: RPC frameworks (e.g., gRPC, Thrift) and RESTful HTTP APIs (e.g., Spring MVC). These categories represent the *de facto* standards for production environments. Consequently, by designing generic bridging mechanisms for these two paradigms, AutoScope ensures broad applicability across the vast majority of industrial architectures rather than being limited to specific tools.

Therefore, our strategy adapts to the nature of the underlying contracts:

- **Explicit Contracts (RPC-based):** Frameworks like Thrift and gRPC rely on strict Interface Description Languages (IDLs). We parse definition files (e.g., `.thrift`, `.proto`) to deterministically map client-side stubs to server-side skeletons. Since the IDL serves as a canonical source of truth, this linkage enables high-confidence bridging independent of the implementation language.
- **Implicit Conventions (HTTP-based):** Frameworks like Spring Boot rely on convention-based RESTful patterns. Here, we perform semantic matching between client-side request artifacts (e.g., `RestTemplate` URL patterns) and server-side routing annotations (e.g., `@RequestMapping`). This effectively captures the loose coupling characteristic of HTTP communications.

Figure 6 illustrates this mechanism using a real-world case from *TrainTicket*. In `OrderService`, the caller method `fetchOrder` initiates an HTTP request via `restTemplate` targeting a specific URL pattern. While standard static analysis fails to resolve the control flow beyond this library call, AutoScope extracts the string literal URL path (`/api/v1/orders/{id}`). It then matches this path against the `@RequestMapping` annotations in the target `OrderController`. By resolving the path variable `{id}`, our approach successfully links the call site directly to the remote `getOrder` implementation.

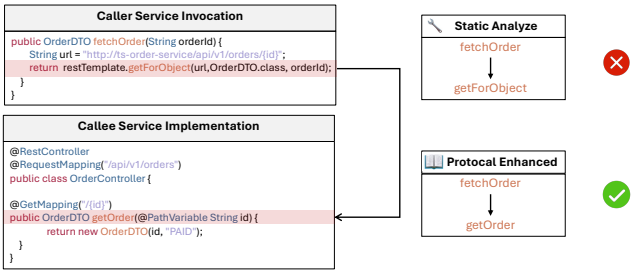

Fig. 6. Example of cross-service invocation bridging in TrainTicket.

To quantify the effectiveness of this procedure, we conducted a preliminary evaluation on cross-service calls within the TrainTicket and SocialNetwork benchmarks. We established a ground truth by manually annotating all cross-service invocations. As expected, general-purpose static analysis frameworks (i.e., Soot and Angr) failed to capture any cross-service edges, resulting in an F1-score of 0. In contrast, our protocol-aware bridging achieved an F1-score of 0.93 on TrainTicket and 0.98 on SocialNetwork. The few remaining inaccuracies primarily stem from highly dynamic configuration-driven URL constructions that evade static resolution. Nevertheless, compared to the baseline, AutoScope provides a fundamental improvement, establishing a solid structural foundation for span-level sampling.

4.2.3 DSS Identification. Based on the global bridged CSCFG, we perform the final step of Phase I: identifying the DSS. The DSS serves as the fundamental unit for our sampling strategy, leveraging the graph-theoretic principle of control flow inference. We leverage the Dominance Relation [2]. In a CSCFG, node u *dominates* node v if every path from the entry to v must traverse u. Furthermore, if v also *post-dominates* u (i.e., every path from u to the exit must traverse v), then u and v are *mutually dominant*. A set of functions that are mutually dominant constitutes a DSS. This implies a deterministic execution dependency: if one function in a DSS is executed, all others in the set must inevitably execute.

To illustrate this intuitively, consider the ticket booking workflow in Figure 3:

- **The DSS Group:** The functions `User.verify(uid)` and `Order.create()` are connected by a linear control flow without intervening branches. This guarantees that the execution of `verify` is always followed by `create`. Consequently, they satisfy the mutual dominance condition and are grouped into a single DSS (marked by the orange dashed box).
- **Branch Exclusion:** In contrast, `Notify.VIP()` resides on a conditional branch (governed by the `isVIP` check). Since the execution of `Order.create()` does not guarantee that `Notify.VIP()` will be called (it depends on runtime data), they are not mutually dominant. Thus, `Notify.VIP()` forms a separate sampling unit.

By statically partitioning the global graph into disjoint DSSs, AutoScope obtains a powerful property: **"Sample one, infer all."** For the DSS in Fig. 3, retaining only the span of `Order.create()` allows us to logically infer the existence and structural context of `User.verify(uid)`. This static partitioning directly counters the structural blindness (Challenge 2 § 3.3). While dynamic traces are stochastic and incomplete, the DSS provides a deterministic structural guarantee, ensuring that the system is aware of the complete execution graph even when runtime data is sparse.

4.3 Phase II: Online Span-level Sampling

Guided by the code knowledge (global CSCFG & DDSs) constructed in Phase I, Phase II executes the sampling logic on the continuous stream of raw traces. As illustrated in the bottom layer of Figure 5, this phase transforms the sampling problem from a naive random selection into a structure-aware decision process. It comprises two sequential modules: *Trace Partition*, which maps dynamic traces to static structures, and *Span Selection*, which executes the final retention decision.

4.3.1 Trace Partition. *Trace Partition* module acts as the bridge between the runtime data plane and the static control plane. Its objective is to map the raw incoming traces onto the CSCFG and to decompose them into DSS instances. This process overcomes the semantic gap between dynamic traces and static code through three progressive steps.

(1) Function-Span Matching. The first challenge is to anchor runtime spans to static CSCFG nodes. Typically, we match a span's operation name (formatted as `Class.Method`) and metadata (e.g., service name) to the corresponding function signature. However, ambiguity arises with shared library functions. For instance, utility functions in `ts-common` (a shared library in TrainTicket) are not bound to a specific service but inherit metadata from the caller. This causes direct mapping to fail as the caller's service does not physically contain the library code. To resolve this, we construct a global library dictionary during the offline phase. When a standard mapping fails, AutoScope queries this dictionary to correctly locate shared functions, significantly enhancing mapping accuracy.

(2) Incremental Path Alignment. After identifying individual functions, we must reconstruct the execution path on the CSCFG. A naive approach of pre-computing all possible graph paths is computationally infeasible due to the path explosion phenomenon in complex microservices. Instead, we adopt an incremental path-matching strategy. AutoScope dynamically traverses the CSCFG, advancing the pointer on the graph as it iterates through the chronologically ordered spans of a trace. This effectively aligns the trace with a valid walk on the graph without storing exponential path combinations.

(3) Robust Handling of Unmapped Spans. A critical issue in production tracing is the presence of "Ghost Spans" where runtime spans that have no static counterparts. This frequently occurs in frameworks like `Spring`, where middleware generates spans for URL mappings or filters that do not correspond to concrete method definitions in the service code. Strict graph traversal would break upon encountering such spans. To ensure robustness, we employ a dynamic programming approach to compute the optimal alignment, treating unmapped spans as insertions that do not disrupt the overall graph matching.

Once the trace is optimally aligned, AutoScope partitions the trace at the boundaries of the pre-identified DSSs. To minimize the latency overhead of this matching process, we implement a trace pattern cache, prioritizing lookups for frequently recurring execution paths.

4.3.2 Span Selection. After partitioning traces into DSS instances, the final challenge is to decide *which* spans to retain. AutoScope approaches this as a resource allocation problem: given a finite storage budget, how do we maximize the diagnostic value of the retained data? We posit that, in a microservice graph, not all spans are created equal. Functions acting as structural hubs (i.e., those bridging critical service interactions) possess higher diagnostic leverage than peripheral nodes. Therefore, a uniform budget distribution would be suboptimal. Guided by this insight, AutoScope adopts a *Topology-Aware Cascade Sampling* strategy. The core logic is to **first allocate the budget based on the static structural importance derived from Phase I, and then fill this budget using a multi-dimensional priority mechanism.**

(1) Topology-Aware Budget Allocation. The allocation process operates in two logical steps: defining the global constraint and distributing the quota. First, we determine the total storage allowance for the current trace. Given a user-defined sampling ratio R_{target} (e.g., 5%) and the set of DSS instances $\mathcal{D} = \{D_1, \ldots, D_n\}$ appearing in the trace, the global span budget B_{total} is defined as:

$$B_{total} = \left\lceil \sum_{i=1}^{n} (R_{target} \times |D_i|) \right\rceil, \tag{3}$$

where $|D_i|$ represents the span count of the i-th DSS.

With B_{total} defined, we distribute it among the n DSS instances. A simple proportional distribution implies that low-volume nodes might receive zero quota. To prevent this structure loss, we adopt a Two-Pass Allocation Strategy:

- **Pass 1 (Minimum Retention):** To guarantee graph connectivity, we first assign a baseline quota $Q_i^{base} = 1$ to every DSS instance. This ensures that no functional node is "starved" to zero, preserving the topological skeleton of the trace.
- **Pass 2 (Weighted Enrichment):** If the budget allows ($B_{total} > n$), the remaining surplus $S = B_{total} - n$ is distributed. We leverage the topological weight $W(D_i)$ derived from the Betweenness Centrality [37] in the bridged CSCFG to bias the allocation towards structural hubs. The additional quota ΔQ_i is calculated as:

$$\Delta Q_i = \left\lfloor S \times \frac{|D_i| \cdot (1 + \alpha \cdot W(D_i))}{\sum_{j=1}^{n} [|D_j| \cdot (1 + \alpha \cdot W(D_j))]} \right\rfloor, \tag{4}$$

where $\alpha \geq 0$ is a sensitivity factor that regulates the influence of static topology on the dynamic allocation. In this work, we set $\alpha = 1$ as the default to achieve a balanced trade-off between execution volume and structural importance. The final quota is $Q_i = 1 + \Delta Q_i$. In extreme cases where the budget is tight ($B_{total} < n$), AutoScope strictly enforces Pass 1 ($Q_i = 1$), accepting a minor budget overflow to prioritize structural integrity over strict ratio adherence.

(2) Cascade Priority Selection. Within the allocated quota Q_i, AutoScope selects specific spans using a three-tier cascade mechanism. This strategy prioritizes deterministic faults over statistical deviations, and temporal diversity over redundancy.

- **Tier 1: Semantic and Structural Failures.** This tier targets spans carrying the highest diagnostic value, capturing both explicit errors and implicit execution interruptions. (1) Explicit Errors: We scan standard span tags (e.g., `http.status_code` $\geq$ `500`, `error=true`) to identify logic exceptions. (2) Structural Interruptions (Early Termination): A critical risk in sampling is inferring the execution of downstream spans that never ran due to crashes or timeouts. To prevent this, AutoScope performs a Structural Consistency Check against the CSCFG. If a span belongs to a DSS but fails to spawn the statically expected child spans (i.e., a mismatch between the runtime children count and the static call degree), it is flagged as a Premature Termination. Any span exhibiting either explicit errors or structural interruptions is assigned maximum priority and retained immediately. This ensures that the tail of a crashed trace is preserved, preventing false positive inferences during reconstruction.
- **Tier 2: Performance Anomalies.** If the quota is not exhausted by Tier 1, we target performance bottlenecks. To handle the multimodal distribution of microservice latencies, we employ a robust Z-score based on the Median and Median Absolute Deviation (MAD). The anomaly score Z_i for a span s_i is defined as:

$$Z_i = \frac{T_{ex}(s_i) - \text{median}(W)}{\text{MAD}}, \tag{5}$$

where $T_{ex}(s_i)$ denotes the exclusive latency (self-execution time)[54], and W represents the sliding window of historical latencies. The MAD is computed as $\text{median}(|w - \text{median}(W)|)$ for all $w \in W$. Unlike standard deviation, MAD provides a robust measure of variability that is resilient to outliers. Spans with Z_i exceeding a dynamic threshold θ_z are ranked by severity. We dynamically maintain θ_z at the 90th percentile of the historical score distribution using the P^2 algorithm [23] to filter out statistical noise. From these candidates, the top-k spans are selected to fill the remaining quota (i.e., $k = Q_i - |S_{tier1}|$), ensuring the most severe anomalies are prioritized.

- **Tier 3: Temporal Diversity.** In stable systems where errors and latency anomalies are rare, we must avoid filling the remaining quota with redundant samples from a single burst. AutoScope employs a least recently sampled strategy (LRS) as a relapse. Spans are prioritized based on their arrival timestamps, favoring those that occupy under-represented time slots within the window. This ensures that the retained spans are uniformly distributed over time, capturing diverse execution contexts rather than repetitive patterns.

4.4 On-Demand Trace Reconstruction

A primary concern with span-level sampling is the resulting fragmentation of trace data, which may impede visualization in standard tools (e.g., Jaeger UI) and obstruct graph-based downstream tasks. To ensure seamless compatibility with existing observability ecosystems, AutoScope incorporates a *Trace Reconstruction* module. Crucially, we design this as a "Lazy Evaluation" process: reconstruction is triggered strictly on-demand when an SRE queries a specific trace ID or an automated RCA tool requests data. This architectural choice shifts the computational burden from the high-throughput ingestion path to the low-frequency query path, ensuring zero overhead on the critical write path.

When a query is issued, the reconstruction proceeds in two steps using the stored fragments and the offline bridged CSCFG:

- **Structural Restoration (Deterministic):** Upon retrieving the retained spans of a trace, AutoScope maps them to their corresponding DSS nodes in the CSCFG. Leveraging the mutual dominance property, we infer the existence of un-sampled functions within the same DSS. Furthermore, by traversing the static graph edges between retained nodes, we regenerate the missing topological links (Parent-Child relationships), restoring the complete skeleton of the call graph. Crucially, this inference is halted if the retained span is flagged as a *Premature Termination* (Tier 1), ensuring that the reconstructed trace accurately reflects the crash point rather than hallucinating a complete execution.
- **Attribute Imputation (Approximate):** While the structure is exact, the runtime attributes of discarded spans are imputed. We employ a statistical imputation strategy using historical attributes (e.g., moving average duration) maintained for each DSS. Let $T_{rec}(s')$ be the reconstructed duration of a missing span s'. It is estimated as $T_{rec}(s') = \mu_{hist}(DSS(s'))$. We deliberately choose the mean over stochastic generation (e.g., Gaussian sampling) to maintain diagnostic safety. Since these spans were explicitly filtered out by the Span Selection module (Tier 2) for lacking significant deviation, their actual latency is statistically guaranteed to cluster around the baseline. Using the mean acts as a noise-suppression mechanism, preventing the reconstruction process from hallucinating synthetic anomalies that could mislead downstream RCA algorithms.

5 Evaluation

To comprehensively evaluate the effectiveness and efficiency of AutoScope , we address the following four research questions:

- **RQ1 (Sampling Efficiency):** To what extent can AutoScope reduce storage overhead? We investigate the relationship between trace complexity and reduction ratio, specifically analyzing the theoretical *Lowest Sampling Ratio (LSR)* required to maintain structural integrity.
- **RQ2 (Sampling Quality):** How effectively does AutoScope capture critical anomalous spans? We compare the faulty span coverage of AutoScope against state-of-the-art sampling strategies.
- **RQ3 (Downstream Utility):** Can the sampled traces support effective Root Cause Analysis (RCA)? We evaluate whether the traces reconstructed from AutoScope 's fragments maintain sufficient fidelity to sustain automated diagnosis algorithms compared to full traces.
- **RQ4 (Sampling Efficiency):** What is the runtime overhead of AutoScope ? We assess the computational latency introduced by the online trace partitioning and selection process to verify its suitability for production environments.

5.1 Experiment Setup

5.1.1 Data Collection. We conducted our evaluation on two widely-adopted microservice benchmarks: TrainTicket (TT) [43] and SocialNetwork (SN) [14]. TrainTicket is A train booking system comprising 41 microservices communicating via RESTful APIs. SocialNetwork is a social media application built with 36 microservices using Thrift RPC and MongoDB. These benchmarks represent diverse communication paradigms (HTTP vs. RPC) and complexity levels, consistent with previous studies [28, 49]. We instrumented all service functions with OpenTelemetry [36] and collected traces using Grafana Tempo [7].

To validate sampling quality under realistic failure scenarios, we utilized ChaosBlade [8] to inject a spectrum of faults into randomly selected services. The fault set includes: (1) resource faults: CPU contention (> 80% load) and network latency (> 500ms delay), (2) code exceptions: logic errors and forced exception returns injected via bytecode modification. Each fault injection lasted 3 minutes to mimic the typical break-fix cycle. In total, we collected 33,255 traces for TT and 12,421 for SN, with an average anomaly rate of 4.12%.

5.1.2 Baselines. We compare AutoScope against five state-of-the-art sampling strategies, categorized into trace-level and span-level approaches. All baselines are configured to match the same sampling budget as AutoScope for a fair comparison.

- **Trace-level Sampling (Tail-based):**
  - **Perch** [26]: An offline approach that groups traces using hierarchical clustering based on structural features and selects representatives from each cluster.
  - **Sifter** [27]: A biased sampling method that prioritizes traces with high prediction errors by maintaining a low-dimensional model of execution paths.
  - **TraceMesh** [9]: A diversity-aware sampling method leveraging Locality Sensitive Hashing (LSH) to dynamically adjust retention probabilities.
- **Span-level Sampling:**
  - **Random Sampling (Uniform)**: A naive baseline where spans are selected with a fixed probability.
  - **Log^2** [11]: Originally proposed as a cost-aware mechanism for logging instrumentation (i.e., deciding which logging statements to execute), it detects anomalous code regions by comparing runtime latency with historical distributions. We adapted its latency-centric filtering logic to the context of span sampling.

5.1.3 Evaluation Metrics. We employ three metrics to assess compression efficiency, data quality, and downstream utility:

- **Sampling Ratio (Compression):** Defined as #Sampled Spans/#Total Spans. Since AutoScope is constrained by the DSS structure, we define the *Lowest Sampling Ratio (LSR)* as the theoretical lower bound required to maintain structural connectivity.
- **Faulty Span Coverage (Quality):** The percentage of ground-truth anomalous spans retained in the sampled dataset. Higher coverage indicates better fidelity in capturing failures.
- **Acc@N & MRR (Utility):** To evaluate the utility for RCA, AutoScope reconstructs traces from the sampled traces and feeds into four SOTA RCA algorithms: TraceRCA [30], TraceAnomaly [31], MicroRank [47], and TraceContrast [50]. We measure the diagnostic accuracy using Accuracy at Top-N (*Acc@N*) and Mean Reciprocal Rank (*MRR*) [48, 49].
- **Processing Latency (Overhead):** We measure the average time consumed by the online sampling pipeline (Trace Partition + Span Selection) per trace.

5.2 Evaluation Results

5.2.1 RQ1: Sampling Efficiency. To answer this question, we analyze the *Lowest Sampling Ratio (LSR)* achieved by AutoScope across different trace complexities. The LSR reflects the theoretical minimum storage required to maintain structural recoverability. Figure 7 visualizes the correlation between trace length (Span Count), structural complexity (DSS Count), and the sampling ratio.

AutoScope demonstrates significant compression capabilities, achieving an average sampling ratio of 14.7% on TrainTicket and 24.9% on SocialNetwork. This translates to a storage reduction of approximately 75% to 85%. Crucially, unlike random sampling strategies that discard data blindly, this reduction is achieved while preserving the complete topological skeleton of every single request, ensuring that no structural context is lost during compression.

A cross-dataset analysis reveals a distinct scale effect, where the sampling ratio exhibits an inverse correlation with trace complexity. As traces become longer, the effective sampling ratio decreases, indicating higher efficiency. For instance, on SocialNetwork, the ratio drops from 30.9% (short traces) to 20.4% (long traces), with a similar downward trend observed in *TrainTicket*. This efficiency gain is driven by the intra-DSS sampling capability of AutoScope . Long traces typically consist of deep, linear execution chains or repetitive loops where multiple functions map to the same DDSs. Due to the mutual dominance property, a single DSS instance may encapsulate a large number of spans (e.g., 10 sequential calls). Thanks to the "sample one, infer all" mechanism, AutoScope only needs to retain a small number of representative spans to reconstruct the entire group, resulting in a low sampling ratio.

In contrast, short traces often consist of fragmented, shallow call paths where each DSS contains only a few spans (often 1 or 2). To satisfy the minimum retention rule ($Q_i \geq 1$) for structural connectivity, AutoScope must retain a larger proportion of these spans. Consequently, the efficiency of AutoScope effectively improves as the trace length increases, because the fixed cost of retaining the structural skeleton is amortized over a larger volume of spans in complex requests.

***Answer to RQ1**: AutoScope achieves highly efficient sampling (avg. 19.8%) by exploiting structural redundancy. It demonstrates a desirable scale effect where efficiency significantly improves on longer traces, as the fixed cost of structural retention is amortized over a larger volume of spans.*

5.2.2 RQ2: Sampling Quality. In this section, we evaluate the quality of the sampled data by measuring faulty span coverage. This metric quantifies the proportion of ground-truth anomalies retained by the sampling strategy. We configured all methods with equivalent target sampling constraints. As analyzed in RQ1, AutoScope is structurally bound by a LSR (14.7% for TT and 24.9% for SN). To align the experimental settings, we set the target sampling probability (R_{target}) for all baseline methods to match these structural floors (rounded to 15% for TT and 25% for SN,

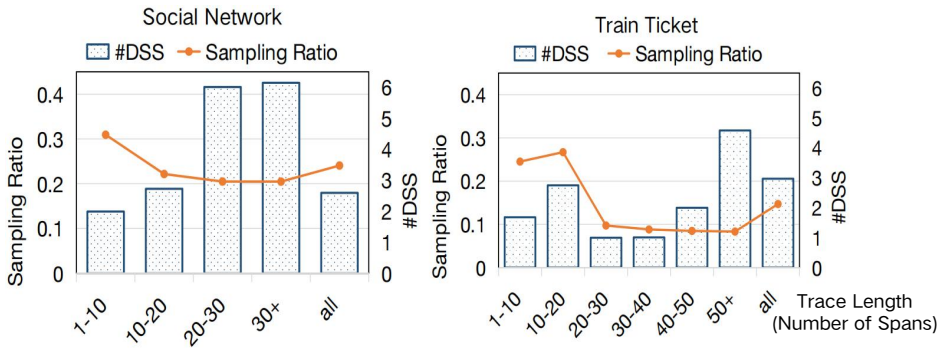

Fig. 7. The sampling ratio with different span and DSS number across two datasets

Table 2. Faulty Span Coverage (%) of Different Sampling Methods

Dataset	Perch [26]	Sifter [27]	TraceMesh [9]	Uniform	Log^2 [11]	AutoScope
TT (15%)	85.4	82.5	89.7	15.2	85.3	**96.8**
SN (25%)	89.7	88.7	94.8	24.7	91.1	**99.3**
Average	87.6	85.6	92.3	20.0	88.2	**98.1**

respectively). While the exact storage consumption may vary slightly due to the stochastic nature of trace-level decisions versus span-level precision, this setup ensures that all methods operate under the same user-defined budget constraint.

As presented in Table 2, AutoScope demonstrates superior efficacy in anomaly retention, achieving an average faulty span coverage of 98.1% across both datasets. This performance significantly surpasses the best-performing baseline, TraceMesh (92.3%), and consistently outperforms traditional trace-level methods. The performance gap highlights the inherent limitation of coarse-grained sampling: baselines like *Perch* and *Sifter* operate on a "1 or 0" trace-level decision. Consequently, if a trace contains a subtle anomalous span but the aggregate trace latency remains statistically normal, the entire trace is discarded. In contrast, AutoScope decouples retention from the trace unit, allowing it to surgically retain specific faulty spans even within otherwise healthy traces. Furthermore, while *TraceMesh* achieves competitive results (>90%), it relies on heavy offline training to build reference models from fault-free data. This imposes a significant cold-start burden, making it brittle to rapid software iterations. AutoScope avoids this dependency, achieving robust performance by leveraging only static code structures without prior training.

A deeper analysis reveals that AutoScope excels in handling distinct categories of anomalies where other methods falter. For performance anomalies, AutoScope calculates anomalies based on exclusive latency (subtracting child span durations) and employs a robust median/MAD estimator. By isolating the local execution time, AutoScope eliminates interference from downstream service delays (sub-span delays). Furthermore, the median-based scoring offers superior stability compared to the mean-variance statistics used in *Log*^{2}, which are easily skewed by the high variability and outliers typical of microservice environments. Moreover, AutoScope demonstrates a critical advantage in capturing structural anomalies caused by logic errors. Consider the `getToken` function in the *TrainTicket* benchmark, where an invalid ID triggers an early return: `if (!id) { return ...; }`. This results in a "short-circuit" trace that is structurally different from a normal execution. Purely latency-based methods often miss these fast-failing spans because their duration appears

normal. However, AutoScope leverages the DSSs derived from the CSCFG. Since the early-return branch corresponds to a distinct DSS node in the control flow graph, AutoScope naturally identifies and retains this structural variant, ensuring that logic errors are not overlooked.

***Answer to RQ2**: AutoScope achieves state-of-the-art faulty span coverage (98.1%) by decoupling retention from trace-level decisions. It excels in capturing both latency anomalies (via robust statistics) and structural anomalies (via code-aware DSS partitioning).*

Table 3. RCA performance under different sampling methods.

RCA	Sampling	Acc@1	Acc@2	Acc@3	MRR
TraceRCA	Perch	41.1	51.6	70.6	0.574
	Sifter	43.7	54.8	74.1	0.599
	TraceMesh	51.9	62.4	78.7	0.661
	AS w/o C	28.2	47.1	63.6	0.522
	AS	**57.9**	**72.7**	**88.3**	**0.715**
Trace Anomaly	Perch	54.4	60.77	76.1	0.670
	Sifter	55.1	63.9	77.2	0.688
	TraceMesh	62.1	69.8	81.0	0.734
	AS w/o C	31.4	48.3	62.3	0.536
	AS	**71.9**	**79.9**	**91.1**	**0.812**
Trace Contrast	Perch	40.9	52.9	71.1	0.573
	Sifter	45.3	56.4	75.1	0.609
	TraceMesh	54.1	66.3	80.7	0.678
	AS w/o C	25.6	41.2	68.4	0.518
	AS	**56.3**	**76.3**	**89.9**	**0.715**
MicroRank	Perch	42.0	52.4	70.2	0.580
	Sifter	43.6	54.9	71.2	0.595
	TraceMesh	51.4	62.9	78.1	0.658
	AS w/o C	24.4	45.8	65.2	0.513
	AS	**57.2**	**70.5**	**88.0**	**0.716**

5.2.3 RQ3: Downstream Utility. In this section, we evaluate whether the traces sampled (and reconstructed) by AutoScope can sustain effective RCA compared to traditional methods. Table 3 details the diagnostic accuracy (Acc@N) and MRR across four SOTA RCA frameworks. **AS** denotes our full method, while **AS w/o** C represents an ablated version (pure span sampling) without code-guided reconstruction

As shown in Table 3, AutoScope consistently outperforms all baselines across four RCA frameworks, achieving an average MRR improvement of 8.3%. The gain is most pronounced in TraceAnomaly (MRR +10.6%). TraceAnomaly employs a VAE-based model to learn normal latency patterns, which requires a large volume of diverse training data. Trace-level sampling (e.g., Perch) discards normal traces, starving the VAE model. In contrast, AutoScope retains the structural skeleton of *all*

requests and reconstructs their attributes via imputation. This provides high-quality, continuous training data for VAE, significantly boosting anomaly detection accuracy. Similarly, for TraceContrast and MicroRank, which rely on comparative analysis between normal and abnormal traces, AutoScope ensures the "normal baseline" is always available, eliminating the data sparsity issue.

The row *AS w/o C* reveals a critical insight: span-level sampling *without* code-guided reconstruction performs disastrously, yielding results even worse than the weakest trace-level baseline (Acc@1 $\approx$ 27.4%, MRR $\approx$ 0.52). This sharp decline confirms that purely latency-based span selection destroys the graph connectivity required by RCA algorithms (e.g., PageRank in MicroRank). Without the bridged CSCFG to connect the dots and restore missing links, the sampled spans become isolated fragments, rendering graph-based diagnosis ineffective. This validates our core hypothesis: code knowledge is not just an optimization, but a necessity for making span-level sampling viable for downstream tasks.

***Answer to RQ3**: AutoScope significantly enhances downstream RCA performance (avg. 8.3% MRR gain) by ensuring structural completeness via reconstruction. The ablation study confirms that code knowledge is the cornerstone of this capability, bridging the gap between fragmented sampling and graph-based diagnosis.*

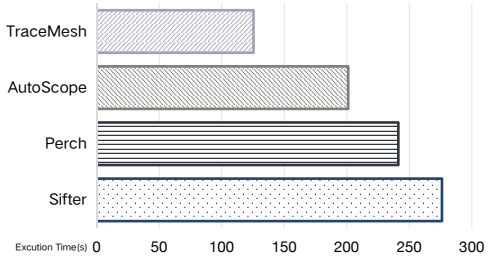

Fig. 8. Time Cost between different sampling methods.

5.2.4 RQ4: Sampling Efficiency. To validate AutoScope 's suitability for high-throughput production environments, we evaluate its runtime overhead. We measure the total execution time required to process the full dataset and calculate the average latency per trace. The results are compared in Figure 8.

AutoScope demonstrates competitive throughput, completing the sampling task in 201.1 seconds, which translates to an average processing latency of approximately 4.4 ms per trace. When compared to the baselines, AutoScope is significantly faster than sophisticated methods like *Sifter* (276.1s) and *Perch* (241.2s), validating its efficiency in complex decision-making scenarios. While *TraceMesh* achieves the lowest latency (125.4s) due to its use of lightweight Locality Sensitive Hashing (LSH), AutoScope incurs only a marginal increase ($\approx$ +1.7ms/trace). We argue that this slight latency cost is a necessary trade-off for the precise structural alignment provided by graph matching, which enables the superior anomaly coverage observed in RQ2. Consequently, AutoScope strikes an optimal balance, delivering high-fidelity sampling with a latency overhead that remains negligible for real-time stream processing.

A deeper decomposition of the runtime cost reveals that the Trace Partition phase accounts for 88% of the total computation. This concentration of overhead is expected, as mapping dynamic traces to the static CSCFG involves graph traversal and path alignment (specifically, the edit distance calculation). In contrast, the Span Selection phase, which involves Z-score calculation and ranking, contributes only 12%, confirming the computational efficiency of our robust statistical implementation.

Answer to RQ4*: AutoScope maintains a high sampling quality while keeping computational overhead at a reasonable level. The primary cost lies in trace partition.*

6 Discussion

6.1 Practical Application

The proposed AutoScope is not merely a theoretical construct but a practical solution designed to address real-world observability challenges. Its application value manifests in three key dimensions:

Cost-Effective Full Coverage for Service Providers. A primary dilemma for cloud service providers is balancing storage costs with data fidelity. Traditional "store-all" policies are financially unsustainable, while random sampling risks discarding critical audit trails. AutoScope breaks this trade-off by achieving an average compression ratio of 19.8% (as demonstrated in RQ1), directly translating to an 80% reduction in storage infrastructure costs (e.g., Elasticsearch or Cassandra clusters). Unlike risky random sampling, this reduction is achieved without data blindness because the structural skeleton of *every* request is preserved. This capability is particularly valuable for compliance auditing and long-term trend analysis, where retaining a complete record of service interactions is mandatory.

Eliminating the Observability Gap for SREs. For SREs, a common frustration with tail-based sampling is the "missing baseline" problem: the system captures the error trace but discards the normal traces required for comparative diagnosis. AutoScope fundamentally resolves this by ensuring that the minimum viable structure of normal requests is always available. When an anomaly occurs, SREs can immediately retrieve structurally identical "healthy" traces reconstructed by AutoScope to perform differential analysis (e.g., comparing span latencies or execution paths). This eliminates the "Trace Not Found" blind spot, significantly accelerating the MTTR.

Seamless Ecosystem Integration. AutoScope is designed as an orthogonal enhancement rather than a disruptive replacement.

- **Pipeline Compatibility:** It operates as a flexible plugin in the tracing pipeline. It can function as a standalone sampler or be chained after a coarse-grained head-based sampler (e.g., applying AutoScope on the 10% traffic allowed by a load balancer) to further refine data density.
- **Tool Agnosticism:** Since AutoScope includes a reconstruction mechanism (§ 4.4), the output data remains standard trace format (e.g., OpenTelemetry). This ensures zero-friction compatibility with existing visualization dashboards (like Jaeger UI or SkyWalking) and automated RCA tools, requiring no modifications to the downstream analysis infrastructure.

6.2 Threats to Validity

We identify potential threats to the validity of our study and discuss the mitigation strategies employed.

Internal Validity. The primary threat to internal validity relates to the accuracy of the ground truth used for evaluation, specifically the correctness of the static-to-dynamic mapping and the labels for faulty spans. To avoid bias from synthetic or unrealistic anomalies, we utilized the industrial-grade tool ChaosBlade to inject faults. This ensures that the generated anomalies (e.g., CPU burn, network delay, exceptions) reflect real-world failure patterns rather than arbitrary random noise.

External Validity. External validity concerns the generalizability of our findings to other systems and languages.

- *Language Specificity:* Our current prototype is implemented in Java using the Soot framework. A potential threat is whether AutoScope applies to other ecosystems (e.g., Go or Python). We

argue that the core contributions (*i.e.*, Dominant Span Set (DSS) partitioning and Topology-Aware Budget Allocation) are mathematical and graph-theoretical concepts, independent of the underlying language. Extending AutoScope to polyglot environments only requires replacing the static analysis frontend (e.g., using SSA form for Go) while keeping the core sampling engine intact.

- *Subject Systems:* We evaluated AutoScope on two open-source benchmarks. While they may not capture the full scale of massive industrial clusters, TrainTicket and SocialNetwork are widely accepted de facto standards in microservices research. They cover diverse communication paradigms (RESTful HTTP and Thrift RPC) and possess sufficient complexity (40+ microservices), providing confidence in the method's applicability to broader production scenarios.

6.3 Limitation

Although AutoScope demonstrates significant advantages, we acknowledge certain limitations inherent to its design philosophy and implementation.

Structural Constraint on Compression Ratio. A fundamental trade-off of AutoScope is that its compression capability is bounded by the complexity of the application's control flow. Unlike random sampling, which can arbitrarily reduce data retention to near-zero (e.g., 1%), AutoScope enforces an LSR to preserve the structural skeleton of traces (as analyzed in RQ1 (§ 5.2.1)). In scenarios requiring extreme storage optimization (e.g., hyper-scale systems where only 1% retention is permitted), AutoScope may exceed the strict budget constraint to guarantee graph connectivity. We prioritize diagnostic utility over raw compression because we argue that retaining disconnected unrecoverable trace fragments provides little value to RCA.

Boundaries of Static Analysis. AutoScope relies on the completeness of the CSCFG. While our protocol-enhanced bridging strategy covers the majority of RPC and HTTP interactions, highly dynamic behaviors (such as reflection-based invocations, complex dynamic proxies, or URLs constructed entirely at runtime via external configurations) may elude static detection. In such cases, the *Trace Partition* module falls back to the dynamic insert strategy. While this ensures robustness, it may lead to suboptimal structural inference compared to fully resolved static paths. Future work could explore integrating lightweight dynamic symbolic execution to improve graph coverage for these dynamic features.

7 Related Work

Trace Sampling approaches. With the exponential increase in trace volume in production systems, trace sampling has become a critical technique to manage data overload and ensure system efficiency. Traditional tracing systems such as Dapper [39], Jaeger [22], and Zipkin [1] have employed uniform random sampling to mitigate storage overhead. However, this approach fails to guarantee the representativeness and quality of the sampled traces, potentially leading to incomplete or misleading insights. To address these limitations, recent studies have explored biased sampling techniques, leveraging methods such as tree-based models [21], clustering algorithms [26], and neural language processing techniques [27]. These approaches primarily focus on identifying and preserving anomalous traces while minimizing the storage of normal traces. For instance, STEAM [17] employs Graph Neural Networks (GNNs) to represent traces and sample mutually dissimilar traces, thereby enhancing system observability. Similarly, Hindsight [53] introduces the concept of retroactive sampling, which aims to capture traces of symptomatic edge cases retrospectively. Despite their advantages, as discussed in Sec. 3.2, these trace-level sampling methods compromise trace queryability and fault diagnosis by disregarding entire normal traces.

Trace-based analysis approaches. In distributed system performance diagnosis, traces play a critical role, serving as the foundation for numerous analyses. For instance, in anomaly detection,

DeepTraLog [51] integrates traces with logs into a graph structure and employs Graph Neural Networks (GNNs) for training, enabling cross-service anomaly detection Similarly, TraceCRL [52]. leverages contrastive learning on operation-call graphs to obtain rich trace representations, significantly improving detection performance. Many automated trace-based RCA approaches have also been developed [30, 31, 47], with some studies [28, 49] incorporating multidimensional information to enhance fault localization. These methods rely on high-quality trace data. However, traditional sampling strategies often result in significant trace loss, reducing analytical accuracy. In contrast, AutoScope adopts a span-level sampling approach that preserves all trace records while effectively minimizing storage overhead, ensuring high-quality data for trace-based analysis.

8 Conclusion

In this paper, we introduce the concept of span-level sampling strategy that preserves structural integrity while reducing storage costs. To implement the concept, we design and develop AutoScope , which samples critical spans from traces while leveraging static analysis to extract execution logic from the application in the form of CSCFG, ensuring that the trace structure is retained. Extensive evaluations demonstrate that this approach achieves an average compression ratio of 19.8% while maintaining a state-of-the-art faulty span coverage of 98.1% and improving downstream RCA performance by 8.3%. These results confirm that integrating static code knowledge into dynamic sampling effectively resolves the conflict between storage efficiency and observability fidelity, with future work planned to extend this capability to highly dynamic and polyglot environments.

References

[1] 2025. Zipkin. https://zipkin.io

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley. https://www.worldcat.org/oclc/12285707

[3] Isil Karabey Aksakalli, Turgay Çelik, Ahmet Burak Can, and Bedir Tekinerdogan. 2021. Deployment and communication patterns in microservice architectures: A systematic literature review. *J. Syst. Softw.* 180 (2021), 111014. doi:10.1016/J.JSS.2021.111014

[4] Frances E. Allen. 1970. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970.* ACM, 1–19. doi:10.1145/800028.808479

[5] Angr. 2025. Angr: A powerful and user-friendly binary analysis platform. https://github.com/angr/angr

[6] Zhengong Cai, Wei Li, Wanyi Zhu, Lu Liu, and Bowei Yang. 2019. A Real-Time Trace-Level Root-Cause Diagnosis System in Alibaba Datacenters. *IEEE Access* 7 (2019), 142692–142702. doi:10.1109/ACCESS.2019.2944456

[7] Mainak Chakraborty and Ajit Pratap Kundan. 2021. Grafana. In *Monitoring cloud-native applications: Lead agile operations confidently using open source software.* 187–240.

[8] ChaosBlade. 2025. ChaosBlade:a cloud-native chaos engineering platform that supports multiple environments, clusters, and languages. https://chaosblade.io/

[9] Zhuangbin Chen, Zhihan Jiang, Yuxin Su, Michael R. Lyu, and Zibin Zheng. 2024. Tracemesh: Scalable and Streaming Sampling for Distributed Traces. In *Proceedings of the 17th IEEE International Conference on Cloud Computing, CLOUD 2024, Shenzhen, China, July 7-13, 2024.* IEEE, 54–65. doi:10.1109/CLOUD62652.2024.00016

[10] Brian Chess and Gary McGraw. 2004. Static Analysis for Security. *IEEE Secur. Priv.* 2, 6 (2004), 76–79. doi:10.1109/MSP.2004.111

[11] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis. In *Proceedings of the 2015 USENIX Annual Technical Conference, USENIX ATC Santa Clara, CA, USA, July 8-10, 2015.* USENIX Association, 139–150. https://www.usenix.org/conference/atc15/technical-session/presentation/ding

[12] François Doray and Michel Dagenais. 2017. Diagnosing Performance Variations by Comparing Multi-Level Execution Traces. *IEEE Transactions on Parallel and Distributed Systems* 28, 2 (2017), 462–474.

[13] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-Trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation NSDI 2007, Cambridge, Massachusetts, USA, April 11-13, 2007.* USENIX. http://www.usenix.org/events/nsdi07/tech/fonseca.html

[14] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine

Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. ACM, 3–18. doi:10.1145/3297858.3304013

[15] Alim Ul Gias, Yicheng Gao, Matthew Sheldon, José A. Perusquía, Owen O'Brien, and Giuliano Casale. 2023. SampleHST: Efficient On-the-Fly Selection of Distributed Traces. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium, NOMS 2023, Miami, FL, USA, May 8-12, 2023*. IEEE, 1–9. doi:10.1109/NOMS56928.2023.10154383

[16] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. 2020. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, Virtual Event, USA, November 8-13, 2020*. ACM, 1387–1397. doi:10.1145/3368089.3417066

[17] Shilin He, Botao Feng, Liqun Li, Xu Zhang, Yu Kang, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. 2023. STEAM: Observability-Preserving Trace Sampling. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. ACM, 1750–1761. doi:10.1145/3611643.3613881

[18] Haiyu Huang, Cheng Chen, Kunyi Chen, Pengfei Chen, Guangba Yu, Zilong He, Yilun Wang, Huxing Zhang, and Qi Zhou. 2025. Mint: Cost-Efficient Tracing with All Requests Collection via Commonality and Variability Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. ACM, 683–697. doi:10.1145/3669940.3707287

[19] Haiyu Huang, Xiaoyu Zhang, Pengfei Chen, Zilong He, Zhiming Chen, Guangba Yu, Hongyang Chen, and Chen Sun. 2024. TraStrainer: Adaptive Sampling for Distributed Traces with System Runtime State. *Proc. ACM Softw. Eng.* 1, FSE (2024), 473–493. doi:10.1145/3643748

[20] Lexiang Huang and Timothy Zhu. 2021. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *Proceedings of the ACM Symposium on Cloud Computing, Seattle, SoCC 2021, WA, USA, November 1 - 4, 2021*. ACM, 76–91. doi:10.1145/3472883.3486994

[21] Zicheng Huang, Pengfei Chen, Guangba Yu, Hongyang Chen, and Zibin Zheng. 2021. Sieve: Attention-based Sampling of End-to-End Trace Data in Distributed Microservice Systems. In *2021 IEEE International Conference on Web Services, ICWS 2021, Chicago, IL, USA, September 5-10, 2021*. IEEE, 436–446. doi:10.1109/ICWS53863.2021.00063

[22] Jaeger. 2025. CNCF Jaeger, a Distributed Tracing Platform. https://github.com/jaegertracing/jaeger

[23] Raj Jain and Imrich Chlamtac. 1985. The P2 algorithm for dynamic calculation of quantiles and histograms without storing observations. *Commun. ACM* 28, 10 (1985), 1076–1085.

[24] Jonathan Kaldor, Jonathan Mace, Michal Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP 2017, Shanghai, China, October 28-31, 2017*. ACM, 34–50. doi:10.1145/3132747.3132749

[25] Prajwal Kiran Kumar, Radhika Agarwal, Rahul Shivaprasad, Dinkar Sitaram, and Subramaniam Kalambur. 2021. Performance Characterization of Communication Protocols in Microservice Applications. In *SmartNets 2021*. IEEE, 1–5. doi:10.1109/SMARTNETS50376.2021.9555425

[26] Pedro Henrique B. Las-Casas, Jonathan Mace, Dorgival O. Guedes, and Rodrigo Fonseca. 2018. Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*. ACM, 326–332. doi:10.1145/3267809.3267841

[27] Pedro Henrique B. Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. 2019. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 312–324. doi:10.1145/3357223.3362736

[28] Cheryl Lee, Tianyi Yang, Zhuangbin Chen, Yuxin Su, and Michael R. Lyu. 2023. Eadro: An End-to-End Troubleshooting Framework for Microservices on Multi-source Data. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1750–1762. doi:10.1109/ICSE48619.2023.00150

[29] Yichen Li, Yulun Wu, Jinyang Liu, Zhihan Jiang, Zhuangbin Chen, Guangba Yu, and Michael R. Lyu. 2025. COCA: Generative Root Cause Analysis for Distributed Systems with Code Knowledge. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, Ontario, Canada, April 4-27, 2025*. ACM.

[30] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, Zhekang Chen, Wenchi Zhang, Xiaohui Nie, Kaixin Sui, and Dan Pei. 2021. Practical Root Cause Localization for Microservice Systems via Trace Analysis. In *Proceedings of the 29th IEEE/ACM International Symposium on Quality of Service, IWQOS 2021, Tokyo, Japan, June 25-28, 2021*. IEEE, 1–10. doi:10.1109/IWQOS52092.2021.9521340

[31] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, and Dan Pei. 2020. Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks. In *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020*. IEEE, 48–58. doi:10.1109/ISSRE5003.2020.00014

[32] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2021, Seattle, WA, USA, November 1 - 4, 2021*. ACM, 412–426. doi:10.1145/3472883.3487003

[33] Vijayaraghavan Murali, Edward Yao, Umang Mathur, and Satish Chandra. 2021. Scalable Statistical Root Cause Analysis on App Telemetry. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 288–297. doi:10.1109/ICSE-SEIP52600.2021.00038

[34] Sam Newman. 2021. *Building microservices*. " O'Reilly Media, Inc.".

[35] Flemming Nielson and Hanne Riis Nielson. 1999. Interprocedural Control Flow Analysis. In *Proceedings of the 8th European Symposium on Programming, ESOP'99, Amsterdam, The Netherlands, 22-28 March, 1999*, Vol. 1576. Springer, 20–39. doi:10.1007/3-540-49099-X_3

[36] Opentelemetry. 2025. High-quality, ubiquitous, and portable telemetry to enable effective observability. https://github.com/open-telemetry

[37] Dimitrios Prountzos and Keshav Pingali. 2013. Betweenness centrality: algorithms and implementations. In *PPoPP 2013*. Association for Computing Machinery, New York, NY, USA, 35–46. doi:10.1145/2442516.2442521

[38] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 693–706. doi:10.1145/3192366.3192418

[39] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper: a large-scale distributed systems tracing infrastructure. (2010).

[40] Skywalking. 2025. APM, Application Performance Monitoring System. https://github.com/apache/skywalking

[41] Soot. 2025. Soot: A framework for analyzing and transforming Java and Android applications. https://soot-oss.github.io/soot/

[42] Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*. ACM, 1093–1105. doi:10.1145/3597926.3598120

[43] TrainTicket. 2025. Train Ticket: A Benchmark Microservice System. https://github.com/FudanSELab/train-ticket

[44] Rongxin Wu, Xiao Xiao, Shing-Chi Cheung, Hongyu Zhang, and Charles Zhang. 2016. Casper: an efficient approach to call trace collection. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, 678–690. doi:10.1145/2837614.2837619

[45] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. 2019. Zeno: Diagnosing Performance Problems with Temporal Provenance. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*. USENIX Association, 395–420. https://www.usenix.org/conference/nsdi19/presentation/wu

[46] Yong Xu, Yaokang Zhu, Bo Qiao, Hongshu Che, Pu Zhao, Xu Zhang, Ze Li, Yingnong Dang, and Qingwei Lin. 2021. TraceLingo: Trace representation and learning for performance issue diagnosis in cloud services. In *CloudIntelligence 2021*. 37–40.

[47] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments. In *Proceedings of the Web Conference 2021, WWW 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*. ACM / IW3C2, 3087–3098. doi:10.1145/3442381.3449905

[48] Guangba Yu, Pengfei Chen, Zilong He, Qiuyu Yan, Yu Luo, Fangyuan Li, and Zibin Zheng. 2024. ChangeRCA: Finding Root Causes from Software Changes in Large Online Systems. *Proc. ACM Softw. Eng.* 1, FSE, Article 2 (July 2024), 23 pages. doi:10.1145/3643728

[49] Guangba Yu, Pengfei Chen, Yufeng Li, Hongyang Chen, Xiaoyun Li, and Zibin Zheng. 2023. Nezha: Interpretable Fine-Grained Root Causes Analysis for Microservices on Multi-modal Observability Data. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. ACM, 553–565. doi:10.1145/3611643.3616249

[50] Chenxi Zhang, Zhen Dong, Xin Peng, Bicheng Zhang, and Miao Chen. 2024. Trace-based multi-dimensional root cause localization of performance issues in microservice systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[51] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-Log Combined Microservice Anomaly Detection through Graph-based Deep Learning. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 623–634. doi:10.1145/3510003.3510180

[52] Chenxi Zhang, Xin Peng, Tong Zhou, Chaofeng Sha, Zhenghui Yan, Yiru Chen, and Hong Yang. 2022. TraceCRL: Contrastive Representation Learning for Microservice Trace Analysis. In *ESEC/FSE 2022*. ACM, 1221–1232.

[53] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. 2023. The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*. USENIX Association, 321–339. https://www.usenix.org/conference/nsdi23/presentation/zhang-lei

[54] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 655–672. https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou

[55] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Trans. Software Eng.* 47, 2 (2021), 243–260. doi:10.1109/TSE.2018.2887384

[56] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 683–694. doi:10.1145/3338906.3338961