ChainScope: Balancing Accuracy and Overhead in Non-intrusive Distributed Tracing of Microservices

RUIPENG HONG, School of Computer Science and Engineering, Sun Yat-sen University, China
GABRIELE CASTELLANO, Huawei Technologies Co. Ltd, France
JINGRUN ZHANG, Huawei Technologies Co. Ltd, China
CHEN SUN, Huawei Technologies Co. Ltd, China
PENGFEI CHEN*, School of Computer Science and Engineering, Sun Yat-sen University, China
MASSIMO GALLO, Huawei Technologies Co. Ltd, France

Distributed tracing is essential for monitoring and debugging services deployed on top of microservice architectures. However, existing solutions face a fundamental trade-off between non-intrusiveness, language/protocol independence, overhead, and accuracy. We present ChainScope, a distributed tracing system that leverages eBPF to establish a universal tracing framework within the kernel. By implementing kernel-level context propagation, ChainScope enables trace-level sampling in a non-intrusive manner, effectively controlling overhead while maintaining high accuracy across diverse service processing models. Evaluation results show that ChainScope achieves better accuracy-overhead trade-offs than existing non-intrusive tracing approaches with up to **2.2**× better accuracy, and **1.6**× higher end-to-end service throughput in complex scenarios such as asynchronous processing and RPC (Remote Procedure Call) protocols. Furthermore, we demonstrate that ChainScope accurately identifies the root cause of long-tail latency in a realistic use case.

CCS Concepts: • **Networks** → **Network performance evaluation**; • **Computer systems organization** → **Distributed architectures**.

Additional Key Words and Phrases: Microservice, Distributed Tracing, eBPF

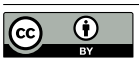
ACM Reference Format:
Ruipeng Hong, Gabriele Castellano, Jingrun Zhang, Chen Sun, Pengfei Chen, and Massimo Gallo. 2026. ChainScope: Balancing Accuracy and Overhead in Non-intrusive Distributed Tracing of Microservices. *Proc. ACM Netw.* 4, CoNEXT2, Article 14 (June 2026), 21 pages. https://doi.org/10.1145/3808662

1 Introduction

Cloud native applications are increasingly relying on microservice architecture to achieve scalability, maintainability, and fault tolerance [8, 24]. In these settings, a single request triggers a set of interdependent operations across several microservices, often implemented in diverse programming languages and run in distributed environments. Given this complexity, distributed tracing has become an essential observability tool for microservice architectures [2, 3, 14, 16, 20, 21, 28, 30, 31, 35, 36, 40, 42, 43, 45]. By capturing the complete execution path of individual requests as they

*Pengfei Chen is the corresponding author.

Authors' Contact Information: Ruipeng Hong, hongrp@mail2.sysu.edu.cn, School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China; Gabriele Castellano, gabriele.castellano@huawei.com, Huawei Technologies Co. Ltd, Paris, France; Jingrun Zhang, zhangjingrun2@h-partners.com, Huawei Technologies Co. Ltd, Shenzhen, China; Chen Sun, sunchen48@huawei.com, Huawei Technologies Co. Ltd, Beijing, China; Pengfei Chen, chenpf7@mail.sysu.edu.cn, School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China; Massimo Gallo, massimo.gallo@huawei.com, Huawei Technologies Co. Ltd, Paris, France.



ACM 2834-5509/2026/6-ART14
https://doi.org/10.1145/3808662

propagate, distributed tracing systems allow developers and operators to identify performance bottlenecks, diagnose failures, and gain a deeper understanding of the system behavior.

However, deploying effective distributed tracing in production is challenging because fundamental requirements such as (1) **non-intrusiveness** without code modification, (2) **coverage** supporting diverse programming languages and protocols, (3) minimal **performance overhead**, and (4) **accuracy** of reconstructed traces, cannot be achieved simultaneously. Existing distributed tracing approaches typically excel at some of these goals while compromising others. For example, application-level instrumentation frameworks like OpenTelemetry [30], Jaeger [20], achieve high accuracy and enable effective sampling via explicit application-level tagging but require tenants' cooperation for instrumenting microservices. In contrast, non-intrusive tracing systems like DeepFlow [35], and Deeptrace [12] eliminate the need for code instrumentation but face significant limitations: they struggle with accuracy in complex scenarios involving asynchronous processing or multiplexed protocols; they often rely on payload inspection, which is ineffective against encrypted traffic (e.g., HTTPS/TLS); and they require complete event reporting, which effectively precludes trace-level sampling. Hybrid approaches [14, 39, 42] primarily focus on minimizing overhead and providing language independence, but do so at the expense of protocol independence and accuracy.

In both practice and literature, current tracing systems face a critical accuracy-overhead trade-off while trying to satisfy the requirements mentioned above. In this paper, we introduce ChainScope, a novel distributed tracing system that takes advantage of the programmability of eBPF (Extended Berkeley Packet Filter) [25] to build an in-kernel universal tracing plane that operates independently of programming languages and network protocols while supporting trace-level sampling mechanisms to regulate data collection overhead, achieving a better accuracy-overhead trade-off. Our approach integrates explicit inter-service context propagation via kernel-level packet tagging with advanced intra-service correlation techniques that can adapt to different service types.

The key insight behind ChainScope is that by performing trace context propagation within the network stack, and entirely within the kernel space, we can achieve non-intrusiveness and high coverage, even for encrypted traffic, while maintaining high trace-level accuracy. This is accomplished through careful design choices in hook placement, sampling, and context propagation mechanisms, all validated through systematic analysis and microbenchmarks. Results show that ChainScope outperforms state-of-the-art methods in both accuracy and overhead. By maintaining high fidelity under load, ChainScope serves as a crucial zero-effort observability baseline to identify the root causes of rare, long-tail events, thereby limiting the need for complex manual instrumentation to the problematic microservice(s).

Our contributions can be summarized as follows.

- We propose a baseline in-kernel universal tracing plane that operates independently of programming languages and application-level communication protocols while supporting sampling to control overhead without sacrificing trace-level accuracy.
- We motivate our design choices with simple high-load microbenchmarks, showing how these affect the system and service overhead.
- We describe extended methods via a plugin architecture for supporting complex microservices that rely on coroutines and RPC.
- We show that ChainScope significantly improves the accuracy–overhead balance for diverse microservice applications, outperforming the state-of-the-art[1] on high load and concurrency use cases.
- We present a complete end-to-end latency diagnosis use-case demonstrating ChanScope effectiveness in root cause analysis.

[1]benchmark code available at https://github.com/hrpccs/chain-scope

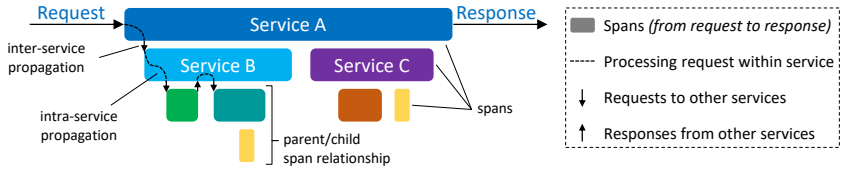


Fig. 1. Breakdown of a trace composed of spans organized in a parent/child hierarchy. Colored boxes represent individual spans (the larger the box, the longer the service took to process the request). Arrows show how the request context is propagated both *across* services (inter-service) and *within* a service (intra-service) to establish causal relationships.

2 Background and motivation

2.1 Distributed tracing

Distributed tracing systems provide visibility into the end-to-end execution path of individual requests as they propagate through a microservice architecture. A *trace* represents the complete lifetime of a single request, from the moment it enters the system until a response is returned. A trace is composed of *spans*, each representing a logical unit of work performed by a particular service during the request lifetime, such as processing an incoming request, performing computation, or issuing an outgoing call to another service. Each span primarily captures the execution duration, measured from the initial receipt of a request to the dispatch of the corresponding response. Spans within a trace are organized into a tree-shaped hierarchy through *parent/child* relationships: a span that triggers a new unit of work becomes the *parent*, and the resulting work unit is its *child*. Figure 1 illustrates the trace that originates when Service A handles an incoming request. A *root span* represents the total time Service A spends processing the request and generating a response. During this processing, Service A issues two outgoing requests to Services B and C, each corresponding to a child span that establishes a parent/child relationship with the root span. Within Service B, the received request may trigger further internal processing steps (such as parsing, computation), and calls to other services (such as a database query); any further outgoing request is represented by its own span with additional parent/child relationships. The complete trace forms a tree of spans, where the edges capture causal dependencies both across and within the service boundaries.

To reconstruct these transaction paths, tracing systems must establish causal relationships between requests at two levels: (1) **inter-service propagation**, where causal relationships are tracked as requests go through services, and (2) **intra-service propagation**, where relationships need to be established within a single service, such as across threads or internal function calls, as depicted in Figure 1. In the following, we outline key mechanisms for both and highlight their trade-offs.

Inter-service propagation. The causal relationship between requests as they traverse service boundaries is traditionally established via *explicit* or *implicit* context propagation. Explicit context propagation consists of injecting tracing information (e.g., Trace ID) into request headers or payloads, enabling decentralized correlation of distributed traces. This enables precise causality and selective data collection (i.e., sampling), but requires complex service instrumentation or middleware integration. In contrast, implicit context propagation determines causal relationships by analyzing network-level metadata, such as TCP/IP tuples and sequence numbers. This approach eliminates the need for explicit context propagation and code instrumentation. However, it relies on a centralized analysis to reconstruct the traces, increasing computational costs, and potentially to accuracy loss. This is even more challenging with complex multi-threaded or asynchronous request handling, making it difficult to accurately infer causality.

Intra-service propagation. Similarly , there exist two main approaches, namely *active* and *passive* intra-service context propagation. With active context propagation, developers are required to explicitly encode causal relationships using mechanisms such as thread-local storage (TLS), coroutine contexts (e.g., Go's `context.Context`), or callback parameters. This ensures correctness by design but burdens developers with challenging manual instrumentation and maintenance. Frameworks like OpenTelemetry alleviate this by using SDKs, but still require explicit integration. With passive context propagation, instead, the causality can be inferred by observing application behavior such as shared thread IDs, coroutine execution, or event timing. Although this approach removes the need for code instrumentation, it can lead to wrong context propagation if the underlying assumptions about application behavior are not met. Moreover, with modern microservice applications, efficient concurrent serving models like user-space threads (Golang coroutines), non-blocking I/O, and thread pools make passive context propagation more challenging.

To address performance, storage, and computation overhead, distributed tracing systems often leverage sampling. A first approach, *head* sampling [30, 36], consists of deciding to trace a request at the beginning. While this approach is effective at controlling overhead, it often misses rare or unexpected issues because these problems cannot be predicted in advance. A second approach, *tail* sampling [17, 18, 23, 41], makes the sampling decision at the end of request processing. By doing so, it can specifically trace requests that show anomalies. However, it does not reduce system overhead because it requires constructing and potentially transferring trace data for all requests before making the final sampling choice. Recently, the concept of Retroactive Sampling [45] was introduced, which aims to combine the benefits of both methods. Trace data is stored locally but is only preserved and processed if a problem is detected after the fact. This method allows for on-demand retrieval of traces for problematic requests with minimal overhead.

2.2 Design Goals and Motivation

Effective distributed tracing systems for microservices architectures require:

Goal 1 — Non-intrusiveness. Operate independently of the application-level, without instrumentation or redeployment, ensuring seamless integration and resilience to versioning.

Goal 2 — Coverage. Achieve broad observability, tracing should be language and protocol-agnostic. While tools like OpenTelemetry provide broad coverage, challenges remain in handling diverse languages or multiplexed protocols.

Goal 3 — Lightweight. Maintain minimal overhead to avoid impacting service performance. This can be achieved by avoiding complex processing [6, 22] or frequent event export [27].

Goal 4 — Accuracy. The tracing system should reliably reconstruct end-to-end request execution paths with high fidelity, precisely capturing causal relationships both intra and inter services. This includes correctly attributing latency, request flow, and failure points, even in the presence of asynchronous processing and concurrency. High accuracy is essential to avoid misleading or incomplete traces that could hinder troubleshooting and root cause analysis.

Achieving these goals simultaneously is challenging. As applications grow in complexity, designing a non-intrusive (G1), high coverage (G2) distributed tracing system that is also lightweight and accurate (G3,4) becomes difficult. Table 1 lists representative distributed tracing frameworks classified according to their approaches and design goals. In the following, we only review non-intrusive tracing systems because they eliminate the need for costly manual instrumentation and code changes, which is a major barrier to adoption and maintenance in production environments.

The inter-service propagation dimension in Table 1 divides implicit approaches (top group) from explicit approaches (bottom group). Statistics-based tracing systems and offline data analysis

Table 1. Frameworks for distributed tracing.

	Methodologies			Properties			
Framework	**Inter-service**	**Intra-service**	**Sampling**	**Non-intrusive**	**Coverage**	**Lightweight**	**Accurate**
TraceWeaver [3]	implicit	passive	tail	✓	cross-language/ protocol	≈ *complex reconstruction*	Fair
REPTracer [43]	implicit	passive	no (tail possible)	≈ *LD_PRELOAD*	cross-language/ protocol	≈ *complex reconstruction, no sampling*	Fair (Good with *Job ID*)
DeepFlow, Rake [35, 47]	implicit	passive	no (tail possible)	✓	cross-language/ protocol	≈ *complex reconstruction, no sampling*	Good (Bad for gRPC)
DeepTrace [12]	implicit	passive	no	✓	cross-language/ protocol	≈ *complex reconstruction, no sampling*	Good
Hindsight [45]	explicit (protocol header)	active	retroactive	× *require APIs*	language/ protocol-specific	✓	Good (only Edge)
Otel-Auto[30]	explicit (protocol header)	passive	head	✓	language/ protocol-specific	× *uprobes*	Good (Bad for gRPC)
ZeroTracer[42]	explicit (HTTP header)	passive	head	✓	cross-language, only HTTP	× *uprobes for coroutines*	Good
Beyla [14]	explicit (HTTP header, TCP/IP for TLS)	passive	head	✓	cross-language, limited-protocol	× uprobes	Good (Bad for gRPC)
ChainScope (base)	explicit (TCP/IP header)	passive	head (retroact. possible)	✓	cross-language no multiplexing	✓	Good
ChainScope (advanced)	explicit (TCP/IP header)	passive	head (retroact. possible)	✓	lang-specific plugins, cross-protocol	≈ few uprobes *for coroutines / gRPC*	Good

systems both correspond to the *implicit* inter-service propagation category, as they infer causality from network-level observations rather than modifying application traffic. TraceWeaver [3] and other statistics-based tracing systems [1, 33] infer event causality from timestamps and service dependency graphs. While they achieve high accuracy with simple, non-invasive instrumentation, they must preserve all data for analysis, consuming significant network and computational resources, violating the lightweight goal. Similarly, other distributed tracing systems relying on offline data analysis [35, 37, 43, 47] and even payload inspection [12] are either resource-heavy or sacrifice accuracy for scalability. In contrast, systems that rely on *explicit* context propagation (bottom group in Table 1), such as [14, 42], attach a unique end-to-end request identifier to all relevant data. While they use sampling to reduce network and storage overhead, they often overlook the significant computational cost of exporting events, which undermines the lightweight goal. This approach also presents a fundamental challenge for accurate context propagation in complex applications. It often necessitates expensive user-space instrumentation [2, 42, 43], undermining lightweight and high coverage goals.

This review highlights a critical gap: no existing non-intrusive tracing system supports sampling while maintaining good coverage and limited overhead simultaneously. This limitation is the core motivation for Chainscope. In Section 3, we detail our design for a universal sampling mechanism that works across all TCP/IP-based protocols that satisfy some fundamental simplifying processing assumptions. We further explain how we balance accuracy and overhead for complex applications in Section 4, which requires dedicated application-level tracing.

3 The ChainScope system

Our analysis in Section 2 shows that the design of a distributed tracing framework inherently involves trade-offs. In the following, we present ChainScope by introducing and validating each design choice via simple microbenchmarks. Specifically, we describe two variants of ChainScope. In this Section, we detail a baseline architecture that efficiently establishes inter-service causality by explicit in-network context propagation and uses simple passive intra-service propagation with minimal overhead. In Section 4 we extend the baseline system with some complementary mechanisms, which, at the price of additional overhead, accurately handle the complex behavior of some modern microservices using thread pools, coroutines, and RPC protocol.

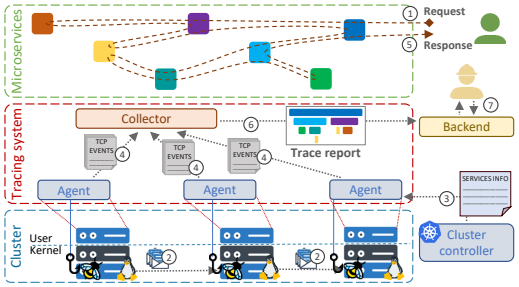


Fig. 2. High-level design of ChainScope.

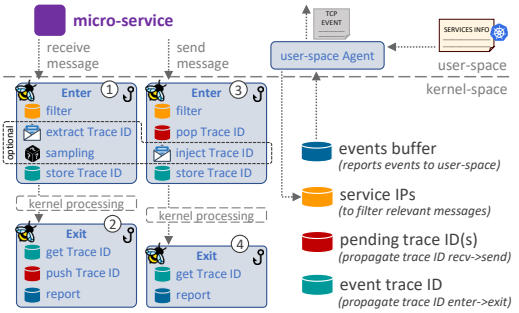


Fig. 3. Simplified Agent architecture.

3.1 System overview

ChainScope consists of two main components: the **Agent(s)** and the **Collector** (cf. Figure 2). Agents are deployed at each server in the cluster and are responsible for transparently capturing inter- and intra-service context propagation. ChainScope Agents leverage the minimal number of eBPF hooks to capture low-level network events, assigning each event a Trace ID that propagates along the same request. This establishes a full in-kernel universal tracing plane with limited overhead. Events are then reported to user-space only when required to reduce processing overhead, where they are processed before being sent to the Collector. The latter reconstructs end-to-end traces from events, exporting them to a backend (e.g., Jaeger [20]) for storage and analysis.

Figure 2 illustrates the steps ChainScope takes to reconstruct a microservice call chain. When the system is started, based on the specific cluster controller (e.g., Kubernetes), Agents are dynamically instructed with the current microservices to be traced and the valid entry points. When a request enters the monitored system ①, the Agent on the first node intercepts the corresponding event and assigns it a trace ID. The trace ID, and additional metadata, is then propagated between services (inter-service) ② by tagging upstream network packets associated with the microservice request. At the Agents, events (or spans) are enriched with cluster-related data ③ (e.g., services names), and then eventually exported to the Collector ④. When the response to the service request is received ⑤, the Collector aggregates events/spans into a trace and eventually sends it to a backend ⑥ that can be queried for further investigation ⑦. In the following, we detail the architectures of ChainScope's Agent and Collector and motivate our main design choices with microbenchmarks.

3.2 Agent

In this section, we describe the baseline architecture of the ChainScope agent while discussing and validating each design choice with microbenchmarks. This baseline captures microservices' universal behavior by making a set of assumptions that simplify intra-service context propagation while remaining applicable to a wide range of microservices and programming languages. First, (1) we assume the thread receiving a request remains responsible for it until the response is sent downstream, i.e., run-to-completion. In cases where threads are spawned to handle the request, additional kernel probes track the association between the receiver and the handler thread, similar to [42]. Second, (2) we assume each incoming request can be mapped to a unique set of network-level events without request aggregation or multiplexing, as opposed to advanced protocols like gRPC (cf. Section 4). Finally, we differentiate between microservices and middleware (e.g., envoy [26]). Middlewares often process requests asynchronously (e.g., `epoll_wait`). To account for this behavior, the ChainScope Agent uses two distinct intra-service context propagation algorithms according to the traced process using the information obtained by the cluster controller.

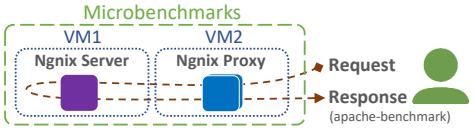
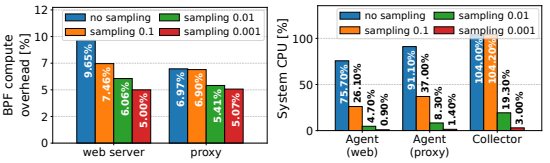


Fig. 4. Microbenchmark with Apache Benchmark [38], 32 concurrent requests. Responses feature a small 13-byte HTTP payload.



(a) eBPF overhead. (b) Components overhead.

Fig. 5. Impact of sampling in terms of processing.

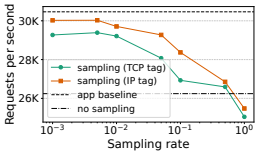
3.2.1 Architecture overview. Figure 3 depicts the Agent's architecture overview. It comprises a user-space component and a series of eBPF kernel-level probes. The user-space component is responsible for two primary operations: (1) maintain a shared `service IPs` map with the IPs of traced services, fetched by interacting with the cluster controller; (2) retrieve *events* reported by the kernel-level probes. Kernel-level probes are responsible for reporting relevant network events while propagating context information. This is implemented by two pairs of probes, hooked before (enter) and after (exit) a message is received/sent.[2] In practice, and based on the design choices, additional eBPF probes might be needed to implement network-level packet tagging/extraction (explicit inter-service context propagation); we detail them later, along with the design choices. Below, we describe the generic tracing workflow steps as depicted in Figure 3.

Receive path. When a TCP segment is received ①, the Agent checks whether the event is correlated to a monitored service through the `service IPs` map. If so, it assigns the event a Trace ID. In case a sampling policy is in place (optional steps with dashed lines in Figure 3), the Trace ID is extracted from the packet (explicit inter-service propagation); if no trace ID is found, and the segment originates from an entry point address, a new one is generated based on the sampling policy. The Agent then associates the current thread[3] with the Trace ID via the `event trace ID` map. After the kernel processing completes ②, a paired *exit* probe retrieves the Trace ID and pushes it to a `pending trace ID` map, which associates the current thread with the last pending Trace ID to be forwarded (either upstream or downstream)[4]. Finally, the event enriched with timestamps, TCP sequence numbers, payload size, endpoint addresses, and part of the TCP payload (for further processing if needed e.g., response status, etc.), is reported to user-space via the `events buffer`, a ring-buffer map that can be used to send a large amount of data from eBPF programs to userspace.

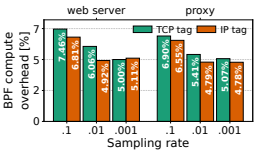
Send path. For outgoing messages ③, a similar service IP filtering procedure is adopted. The *enter* probe checks if the current thread is associated with a pending Trace ID. If so, it propagates toward the *exit* probe through the `event trace ID` map. Optionally, unless the message is a response to the user request, the Trace ID is injected in the outgoing message for inter-service propagation. This is needed in the case of sampling to keep the decision consistent along the service chain. After kernel processing ④, the event is reported to the user-space based on the sampling policy ②.

3.2.2 Design choices. We now present the Agent architecture in detail based on some key design choices validating them in a simple high-load benchmark setting consisting of an nginx web server behind a proxy, deployed in two virtualized nodes, each with eight cores, as depicted in Figure 4.

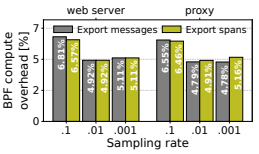
[2]ChainScope implementation only uses one hook after the receive and one before the send, at the price of some missing information (e.g., sequence numbers, timestamps); we describe the full enter/exit solution for simplicity and completeness.
[3]Retrieved with the eBPF utility `bpf_get_current_pid_tgid()`.
[4]Note that, in general, a thread might have multiple pending Trace ID at the same time; we better detail our intra-service propagation mechanism at Section 3.2.3.



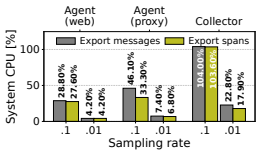
(a) E2E throughput.



(b) eBPF overhead.



(a) eBPF overhead.



(b) Components overhead.

Fig. 6. Performance for different tagging approaches.

Fig. 7. Performance for different event granularities.

To measure the compute overhead of BPF probes, we leverage `perf` and sample relevant kernel components at 1000 Hz to get the percentage of total execution time spent in the BPF probes.

Design choice 1 — Hooks placement. The location of the eBPF probes affects both tracing overhead and granularity of captured events. System call tracepoints, used by [35]—i.e., `recvmsg`/`recvfrom` for the receive path (cf. ① and ② in Figure 3), and `sendto`/`sendmsg` for the send path (cf. ③ and ④)—might trigger fewer times than TCP/IP kernel functions (a single `sendto` might lead to multiple IP packets sent), but might often be called out of the scope of network communications. We argue that kernel TCP functions, i.e., `tcp_recvmsg` and `tcp_sendmsg`, provide a good compromise: they offer more targeted instrumentation, avoid IP-level fragmentation[5], and hook into highly stable kernel points that are rarely subject to breaking changes across kernel updates.

Design choice 2 — Sampling. A core design choice concerns incorporating a head-based sampling mechanism at the kernel level (a complementary retroactive sampling strategy could be easily implemented by storing events in kernel maps and exporting them only if needed). This provides a powerful lever for reducing the number of events reported to the user-space. However, it comes with challenges as the sampling decision should be consistent across all the events of a given service chain, to avoid collecting incomplete traces. This requires explicit inter-service context propagation to inform next-hop Agents if a request is traced. Note that an agent that reports all events without sampling would not need any explicit inter-service context propagation—this could be inferred implicitly from the TCP sequence numbers after the events have been collected.

To evaluate this trade-off, we vary the sampling rate from 0.1% to 10%, and compare it to the case in which all events are reported. Figure 5a shows ChainScope probes overhead on the service processing. While the sampling mechanism adds some overhead, this is highly compensated for small enough sampling rates. Figure 5b shows that the sampling has instead a much bigger impact system-wide. When all events are reported, ChainScope components reach up to 100% of CPU usage, indicating they cannot process the information at line rate. Conversely, the CPU usage quickly drops when reporting only a fraction of events. Finally, we report the end-to-end performance of the service chain in Figure 6a, observing that it can serve a higher number of requests compared to the *no sampling* case as long as the sampling rate is below ≈50%. Remarkably, 1% sampling has less than 4% performance loss compared to the application baseline, i.e., run without tracing system. Because of the prohibitive system-wide CPU consumption, we argue that a simple non-sampling solution such as the one adopted by [35] is impractical. Therefore, for the remainder of this paper, we will always refer to a sampling-enabled agent with explicit inter-service context propagation.

Design choice 3 — Tagging mechanism. Explicit inter-service context propagation requires a mechanism dictating how Trace IDs propagate. We discard HTTP-level tagging as the eBPF verifier restricts arbitrary header parsing and cannot inspect encrypted HTTPS traffic. Instead, we evaluate two lower level approaches: TCP and IP-level tagging. By operating below the TLS layer, both approaches can propagate context even in the case of encrypted traffic. We implement the first

[5]TCP-level fragmentation is handled by capturing consecutive calls to kernel TCP functions, which are separately reported to the user-space and later merged at the Collector level.

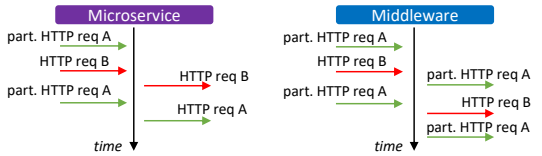


Fig. 8. Intra-service context propagation behavior in case of Microservice and Middleware.

Sampling rate	1	0.1	0.01	0.001
Additional CPU (mean)	1.73%	0.75%	0.43%	0.11%
Requests per second	-4.08%	-1.53%	-0.55%	-0.12%

Table 2. Extra overhead due to the usage of the epoll-FIFOmechanism.

through TCP options. In the receive path, an additional probe at the `tcp_v4_rcv` kernel function parses the TCP header option, accessing the socket buffer right before step ① in Figure 3. In the send path, a `sock_ops` probe triggers twice: first to reserve space for the additional TCP option, and then to write the Trace ID in the option—this happens right after step ③ in Figure 3.[6]

In contrast, IP-level tagging uses a `tc` eBPF hook to write the Trace ID as a custom IP option. On the receive side, an `ip_options_compile` hook parses the option. Specifically, the `tc` probe reads the Trace ID to be injected from a map (indexed by the socket buffer's `seq` field) where it was first written by the `tcp_sendmsg` probe. The `ip_options_compile` probe stores the Trace ID in the same map, to be later read by the `tcp_sendmsg` probe. For local communication where the NIC and `tc` probe are not involved (e.g., containers on the same node), the `tcp_recvmsg` probe directly reads the Trace ID from the map, as it shares the same socket buffer pointer with `tcp_sendmsg`.

Figure 6 compares the overhead of TCP and IP-level tagging for our service chain. Overall, IP-level tagging has a lower overhead, primarily because it simplifies local communication and requires fewer eBPF probes. We therefore use IP-level tagging in ChainScope.

Design choice 4 — Events granularity. So far, we described an Agent that reports data to user-space at the message granularity. This means that every occurrence of a `tcp_recvmsg` (or `tcp_sendmsg`) belonging to a sampled chain is reported as soon as it is observed. The Collector is then in charge of correlating requests/responses and associating them in spans. An alternative approach consists of reconstructing spans directly at the eBPF probe level, and then reporting each span as a whole instead of as two (or more) distinct events. While this halves the reports from kernel- to user-space, also reducing the Collector workload (Figure 7b), it adds complexity to the eBPF probes. This is because it requires maintaining an additional state to correlate request events with their responses. Furthermore, if a response comprises multiple messages, additional events must be reported after a span is closed, further complicating state maintenance. Figure 7 presents unstable performance in terms of eBPF overhead when reporting spans, with occasionally higher CPU load even for smaller sampling rates. The reason is that, as the sampling rate drops, the overhead of the additional state maintenance is higher than the benefit of halving the events to report. Since different use cases might lead to different priorities between service stability and system overhead, we support both modes in ChainScope. For simplicity, from now on, we refer to the message-level variant unless otherwise specified.

3.2.3 Intra-service context propagation. Finally, we detail the mechanism for propagating trace information within a service, specifically between steps ② and ③ in Figure 3. As mentioned earlier, we implement two slightly different propagation mechanisms, one for middleware and another for microservices. This distinction is necessary because middleware might initiate upstream communication before completely receiving the downstream message, whereas we assume microservices exhibit run-to-completion behavior.

[6] `sock_ops` and `tcp_v4_rcv` hooks used for writing and parsing TCP header options are run by different (kernel) threads. To propagate the Trace ID, we use an additional map indexed with the address to the socket buffer.

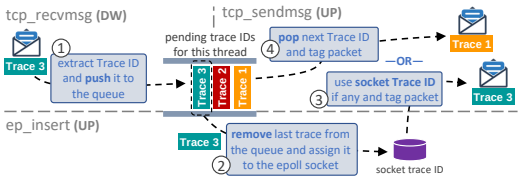


Fig. 9. Epoll-enhanced FIFO mechanism for intra-service context propagation.

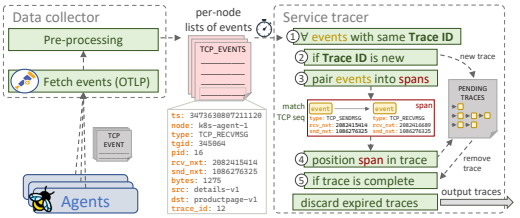


Fig. 10. Architecture of the ChainScope Collector.

As shown in Figure 8, a microservice's single thread might receive two different requests (traces A and B). It handles B first (by sending an upstream request) because trace A was not yet complete. This run-to-completion behavior simplifies our approach: for microservice threads, it is sufficient to store a single pending trace ID at a time, representing the last observed `tcp_recvmsg`. We assume that when the thread sends a TCP segment, the last received message corresponds to a complete request that is now being responded to or handled with an additional upstream request. We trace this behavior using a `pending trace ID` map that stores the last received Trace ID for each thread and immediately propagates it intra-service as soon as that thread performs a `tcp_sendmsg`.

For middlewares, we need to store multiple pending Trace IDs per thread in the `threadtrace ID` map. We assign each thread a queue of Trace IDs, and implement a FIFO mechanism, propagating the oldest Trace ID still in the queue during a `tcp_sendmsg` event. However, due to asynchronous processing, this FIFO assumption might not always hold. We extend our mechanism by tracing `epoll` calls to detect out-of-order downstream and upstream communications (Figure 9). In particular, at `ep_insert` call, we check if it refers to an upstream socket (*UP*) being used to send. If so, we bind it with the last downstream socket (*DW*) used to receive. We then remove the Trace ID received by *DW* from the queue (if present) and propagate it upstream once a `tcp_sendmsg` on *UP* occurs. This epoll-enhanced FIFO mechanism allows us to model the behavior of widely used middleware (e.g., envoy [26] and nginx [19]) and correctly propagate the trace context in highly concurrent asynchronous scenarios at the cost of little extra overhead (cf. Table 2).

3.3 Collector

The ChainScope Collector operates through a two-stage pipeline: *Data Collector*, and *Service Tracer*, as illustrated in Figure 10. The first receives events reported by all Agents by means of the OpenTelemetry Protocol (OTLP), either via gRPC or REST APIs. It performs basic event preprocessing by merging TCP events associated with the same request/response. The merging process relies on thread identifiers and sequence numbers to determine event correlation. Processed events are sorted chronologically by timestamp and organized into per-node lists, which serve as the interface between the two processing stages.

The Service Tracer consumes events from the shared per-node lists and reconstructs traces through a multi-step process. Events are grouped by Trace ID ①, enabling the tracer to identify all components of a service chain. If there is no pending service trace for the Trace ID, one is created ②. The tracer then performs span reconstruction by matching complementary events: an event that refers to the receive of a request is paired with the event that refers to the send of its response, and vice versa, thus creating receive-send and send-receive span pairs ③. The tracer determines span positioning within traces using temporal ordering for intra-node spans and TCP sequence numbers for inter-node request flows ④. Finally, the service tracer validates traces by ensuring that every send-receive span has a corresponding receive-send span and that an entrypoint span exists

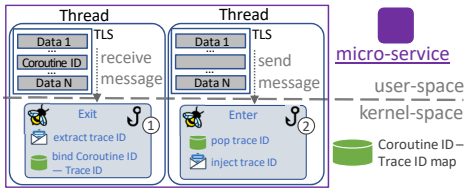


Fig. 11. Coroutine: Intra-service context propagation.

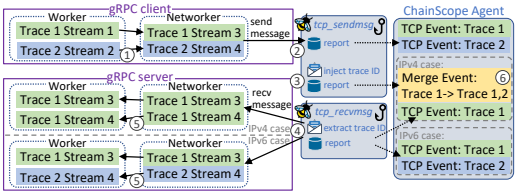


Fig. 12. gRPC: Inter-service context propagation.

⑤. Successfully reconstructed traces are output as valid traces, while incomplete or malformed traces are appropriately categorized as invalid or discarded based on predefined expiration policies.

4 ChainScope Extensions

The base version of ChainScope presented in Section 3 operates under simplified assumptions: (1) a thread-per-request model where the receiving thread handles the entire request lifecycle (at most spawning new threads only for handling upstream requests), and (2) a direct mapping of network events to application requests (no multiplexing). While these assumptions allow for efficient tracing across a wide range of microservices, modern applications increasingly adopt sophisticated execution models such as Java thread pools, Go goroutines, Rust Tokio tasks, and more complex application-level communication protocols like gRPC [15]. In this section, we extend ChainScope with two plug-ins that can be used to handle these advanced scenarios while maintaining its core in-kernel cross-protocol inter-service context propagation unaltered.

To accurately infer event causality in these complex scenarios, `uprobes` are required to retrieve important application-level metadata like Coroutine ID. However, `uprobes` are inherently expensive, as they force a thread to trap into the kernel address space for data synchronization with kernel-level hooks. Consequently, the key challenge to extend ChainScope is to carefully use `uprobes` to achieve a good balance in the accuracy-overhead trade-off. In ChainScope, we use language-specific tracing plugins with a few application hooks for higher accuracy, and do heavy operations like dumping gRPC request headers from userspace memory [48] only when a request is sampled. In the following we detail coroutines/thread-pool, and RPC plugins for ChainScope.

4.1 Handling advanced concurrency models

Sophisticated concurrency models such as thread pools and coroutines fundamentally disrupt the simple, thread-based request tracking used by the baseline ChainScope system. For example, with thread pools, network threads receive requests and dispatch them to worker threads from a shared pool, breaking the thread-request correspondence. Similarly, coroutines and user-space threads, like Go goroutines or Rust Tokio tasks, introduce lightweight execution contexts that can be scheduled over multiple OS threads, further complicating context propagation. Existing approaches [35, 42] address this challenge by monitoring coroutines scheduling and thread pool task queues by inserting uprobes.

Chainscope adopts the same uprobe-based methodology as ZeroTracer [42] for threadpools. However, for coroutines tracking, based on our observation, *it is possible to eliminate the frequently triggered uprobe for monitoring the coroutine scheduler, reducing the overhead while maintaining the same accuracy.* We achieve this with a novel unified abstraction that treats both threads and coroutines as task processors, i.e., execution contexts capable of handling requests. This enables consistent context propagation across diverse concurrency models without requiring continuous scheduler event monitoring. Our method leverages the fact that most modern coroutine runtimes

maintain the current task identifier in thread-local storage (TLS) of the thread executing the coroutine.[7] In essence, when receiving a message with a trace context, the ID of the current task processor is retrieved from the TLS and bound to the trace ID. Similarly, for outgoing messages, the Task ID is read from the TLS and bound to the trace ID. Figure 11 depicts the process mapped to the Intra-service context propagation process for the baseline ChainScope detailed in Section 3.

In practice, when coroutines are implemented via dynamic libraries, the calls to `tls_get_address` used by libc to maintain the address of TLS can be monitored using an eBPF hook. By observing the return value of this function, we can obtain a unique identifier for the current coroutine by knowing where it is saved in the TLS.[8] For example, in Golang, the unique identifier of the goroutine currently executing can be obtained from r14 register or TLS for the current thread. And the address of the TLS can be obtained from the `fsbase`[9] register according to the Golang ABI Specification [13]. The main advantage of our approach is that it eliminates the need to monitor scheduling events to track which coroutine is running on a given thread, reducing expensive uprobes overhead.

4.2 Handling RPC communication protocol

The use of modern RPC application-level communication protocols, such as gRPC [15], in microservice architectures poses a significant challenge to the accuracy of inter-service context propagation. This challenge arises from the concurrency models employed by these services constructed with a microservice architecture, which often multiplex several RPC requests (i.e., streams) within a single system call or network packet. The baseline ChainScope assumes a one-to-many mapping between a request and multiple TCP segments sent from the same socket. As a result, it would only be able to propagate the context for one of the multiplexed requests, resulting in the loss of trace contexts for the others.

Existing solutions have explored the inter-service propagation for RPC. For example, DeepFlow [35] uses implicit context propagation by recording the association between `tcp_seq` and `stream_id` for gRPC spans. However, this method is inaccurate under high loads, as the TCP segment coalesce mechanism in the Linux kernel can cause a mismatch between the TCP sequence numbers observed at the client and at the server[10]. Additionally, DeepFlow uses uprobes to retrieve the exact position of the different gRPC messages within the TCP payload, introducing significant overhead. In contrast, Beyla [14] and OpenTelemetry-auto-go [39] employ uprobes and the `bpf_probe_write_str` helper function to perform explicit context propagation by injecting metadata into gRPC headers. This approach can achieve high accuracy, but can fail to inject the header if the buffer is full or unsafe. Moreover, it requires complex logic for safe header injection and additional uprobe hooks, increasing overhead.

In ChainScope, we preserve the baseline kernel-level inter-service context propagation method for gRPC, eliminating the need for additional uprobes for header injection. To prevent the inaccuracy experienced in DeepFlow, ChainScope hooks the `tcp_try_coalesce` kernel function to keep track of the coalesce mechanism and adjust trace propagation, thus avoiding dropping trace contexts due to multiplexing. In particular, we extend the baseline inter-service propagation mechanism to allow the propagation of multiple trace contexts within the same message, as shown in Figure 12. We use the `TraceID,StreamID` tuple as the trace context, where `TraceID` represents the end-to-end

[7] This assumption holds for libco (C++), greenlet (Python), tokio (Rust), goroutines (Go), and virtualthreads (Java).
[8] Note that this requires specific solutions depending on the coroutine programming language.
[9] An x86_64 CPU register indicating the start of the current thread's TLS.
[10] A TCP segment associated with `stream_id 1` and starting with `tcp_seq 1` can be coalesced with a second TCP segment starting with `seq_seq 2` (`stream_id 2`) into a larger continuous TCP segment that would start with `seq_seq 2`. In this case, while the client observes two distinct events, the server only observes one with the latter sequence number, thus preventing some of the events from being correlated.

request and `StreamID` is used to propagate the context from the server's *networker* to its *worker*. When two (or more) simultaneous gRPC requests are issued ①, the client-side networker buffers all their trace contexts (`Trace 1` and `Trace 2` in the figure), storing the association between `TraceID` and `StreamID`. These send events are reported separately to the ChainScope Agent ②. During inter-service context propagation ③, all buffered trace contexts are appended to the option header together with their stream association. The server-side networker then reports receive events ③ individually, and de-multiplexes `TraceIDs` to the worker using the associated `StreamID` ⑤. For IPv4-based inter-node traffic where trace contexts may not fit within the limited 40-byte TCP/IP option space—IPv6 options have no space limitation—we merge pending contexts and only propagate one `TraceID`,`StreamID` tuple. A special event ⑥ is produced to inform the ChainScope Agent of this merge. The Collector receiving these special events can then correctly resolve trace associations during reconstruction thanks to the `StreamID`, ensuring correct span positioning.

5 Experimental results

In this section, we analyze ChainScope performance trying to answer the following questions:

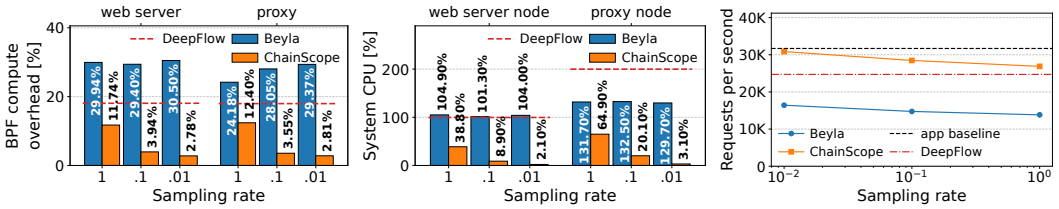
Q.1 *What is the overhead of Chainschope at different levels of the system under heavy request load?*
Q.2 *What is the impact of the sampling mechanism?*
Q.3 *How does ChainScope perform in complex microservice advanced scenarios?*
Q.4 *How does ChainScope compares against state-of-the-art distributed tracing frameworks?*
Q.5 *How does ChainScope helps in identifying performance issues in a microservices architecture?*

The experiments were carried out in two-node and six-node Kubernetes test cluster, separately for the nginx workload benchmark and the advanced gRPC benchmark. Each node is a virtual machine provisioned with 8 dedicated CPU cores and 4 GB of DRAM, deployed on a host equipped with two Intel(R) Xeon(R) Platinum 8164 CPUs (@ 2.00GHz) with 52 cores each. VMs run Ubuntu 24.04 with Linux kernel 6.8.

5.1 Baseline Chainscope tracing: Nginx Workload (Q.1 Q.2 Q.4)

To evaluate ChainScope's baseline overhead, we use the same high-load benchmark as in Section 3.2—an Nginx web server and proxy (cf. Figure 4)—and compare it against DeepFlow [35] and Beyla [14], two state-of-the-art non-intrusive tracing frameworks, representative of their respective categories from Table 1[11]. Note that a comparison to frameworks that require manual instrumentation (e.g., Jaeger, OpenTelemetry, etc.) is orthogonal to our goal of non-intrusiveness. While intrusive tracing frameworks are expected to work well when manual instrumentation is possible (see Section 5.3), ChainScope provides broad coverage and zero-effort deployment, operational advantages that are key for monitoring and rapidly debugging microservices architectures. We configured DeepFlow to use eBPF hooks for network I/O tracing only and turned off the file I/O tracing and NIC hooks. We remark that DeepFlow cannot enable any sampling mechanism. For Beyla, we used the header-based context propagation mechanism, as its alternative IP options-based method only supports HTTP1.x/HTTPS. For a fair comparison, we configure ChainScope to also dump a portion of the TCP payload to user space like Beyla and DeepFlow do—for retrieving details like API endpoints and response statuses.

[11]DeepFlow and Beyla differ in their approach to inter-service context propagation: DeepFlow relies on implicit propagation, inferring trace context from network-level observations without modifying messages, whereas Beyla explicitly injects context into outgoing request headers. We selected these two tools as representatives of their respective categories on the basis of their maturity and breadth of protocol coverage—e.g., we chose Beyla over ZeroTracer [42] as the latter does not support gRPC.



(a) Overhead of eBPF programs. (b) Tracing system. (c) End-to-end chain throughput.

Fig. 13. Compare ChainScope with other tracing systems under heavy request load (cf. Figure 4). DeepFlow is shown as a constant line as it does not support sampling.

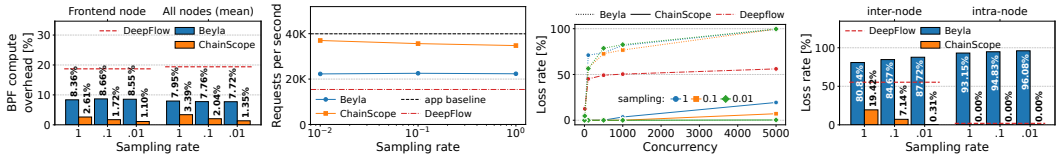
Compute overhead — eBPF and System. Figure 13a presents the overhead of eBPF hooks. At 100% sampling rate, ChainScope overhead is comparable to that of DeepFlow. The latter has slightly higher overhead because it includes extra logic for protocol inference, an operation ChainScope delegates to the userspace to keep eBPF programs simple. The high eBPF overhead of Beyla is mainly due to two factors. First, it must parse the packet payload to perform trace context injection, requiring duplicated hooks due to the eBPF instructions limit. Second, Beyla includes additional logic to match requests and responses into spans. Crucially, as the sampling rate decreases, *only ChainScope's eBPF overhead is reduced*, demonstrating the benefit of its in-kernel data filtering. Although Beyla supports sampling, it sends all potential trace data to its user-space agent before making a sampling decision. As a result, lowering the sampling rate in Beyla only reduces the overhead of exporting spans to the collector. However, Figure 13b shows that both Beyla and DeepFlow could not process events at line rate for this high-load test; their user-space CPU is 100-200%—suggesting that their threads are saturated. On the contrary, ChainScope agents exhibit lower CPU demands, even negligible (2.10%-3.10%) when the sampling rate is 1%.

End-to-end performance. Figure 13c shows the end-to-end performance in requests per second (RPS). For very high sampling rates, ChainScope performance is slightly better than DeepFlow, due to the additional overhead for protocol inference. At lower sampling rates, *ChainScope data collection overhead is minimal and its performance is nearly identical to the baseline*, the benchmark with no tracing. Beyla consistently exhibits the lowest performance due to its high overhead at both eBPF and system levels.

Accuracy. In the end-to-end performance test, we recorded the accuracy, defined as the percentage of traces completely reconstructed from a total of 100k requests. Chainscope achieves 100% trace-level accuracy (excluding intentional drops due to sampling) primarily because our epoll-enhanced FIFO assumption perfectly matches services behavior. In contrast, Beyla achieves 90% accuracy while DeepFlow only 21%. To investigate the poor performance of DeepFlow and Beyla, we conducted additional experiments sending fewer requests $[1k; 10k]$. These tests yielded much higher accuracy, suggesting Beyla and DeepFlow either misinterpret asynchronous processing events, i.e., epoll, or miss spans and hence traces due to their higher overhead.

5.2 Advanced tracing: gRPC/coroutines mesh (*Q.3 Q.4*)

For evaluating advanced tracing scenarios, we use the DeathStarBench hotel reservation mesh [10], a complex gRPC-based application with coroutine-enabled microservices, and deploy it on top of the six-node cluster with 3 instances for each service for load balancing. We configure the microservice placement such that gRPC traffic is 50% inter- and 50% intra-node. Requests towards



(a) eBPF overhead. (b) E2E throughput. (c) Loss by concurrency. (d) Inter/intra node loss.

Fig. 14. Tracing systems comparison under a complex chain (Hotel Reservation). **(d)** is for concurr.=5K.

the microservice chain are sent using the Apache Benchmark command suite ab [38] with 1000 concurrent requests, unless otherwise specified.

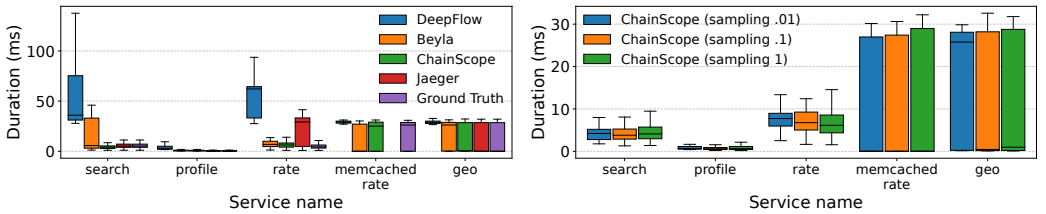
Coverage. Regarding coverage, *only DeepFlow and ChainScope can produce spans for all services*, including storage services like Memcached and MongoDB. Beyla's support is limited to pre-instrumented libraries (e.g., for Redis and SQL), resulting in incomplete traces for this application, as it does not cover MongoDB or Memcached.

End-to-End Overhead. Figures 14a and 14b show the eBPF overhead (frontend node and average) and the end-to-end chain performance, respectively. DeepFlow exhibits the highest eBPF overhead, significantly degrading application performance. Our analysis reveals this is caused by its heavy use of the `bpf_probe_read_user` helper function to traverse gRPC internal data structures to find socket descriptors and TCP sequence numbers. While Beyla's eBPF overhead is roughly 50% lower than DeepFlow's, it remains considerably higher compared to ChainScope. This is due to the additional uprobes for application-layer context propagation, and the intensive user-space memory reads and string comparisons required for header parsing. In contrast, ChainScope's method for advanced concurrency (cf. section 4.1) simply requires a single hash map lookup to look for a trace context attached to the current coroutine/thread, resulting in lower overhead and higher end-to-end throughput.

In general, *ChainScope achieves the lowest end-to-end impact on performance* (cf. Figure 14b). However, unlike in the nginx-based experiment, reducing the sampling rate provides only a marginal 5% performance improvement, and the RPS drops by only ≈6% from the baseline at 1% sampling rate. This behavior suggests that the benchmark is `non-CPU-bound` for this service mesh. By minimizing the number of expensive uprobes, ChainScope keeps its CPU demand low. In contrast, Beyla and DeepFlow require significantly more CPU resources, causing the service mesh to become `CPU-bound` and suffering a greater performance impact.

Tracing accuracy. Finally, we evaluate the accuracy under high concurrency. In our experiments, we found that DeepFlow cannot produce end-to-end traces for this benchmark. This is because DeepFlow intra-service context propagation is incompatible with the gRPC thread-pool model, making it unable to correlate the frontend HTTP span with the subsequent gRPC client span. Hence, the accuracy results for Deepflow only refer to inter-service context propagation.

Figure 14c shows the reconstruction loss, i.e., sampled chains that could not be correctly reconstructed. All tracing systems achieve perfect accuracy (i.e., 0% loss) with a single concurrent request. However, as concurrency increases, the reconstruction loss for DeepFlow and Beyla increases drastically. Beyla's loss rate approaches 100% when concurrency reaches 5000. This occurs because it injects trace context into gRPC request headers at the application layer with `bpf_probe_write_user`, which fails whenever the buffer is full or the operation is not safe. This decision depends on the buffer state, not on sampling. Hence, reducing the sampling rate does not reduce loss. DeepFlow loss rate also increases with concurrency due to the Linux TCP segment coalescing mechanism. This can cause the TCP sequence number at the server to differ from the



(a) Compare different tracing systems.

(b) ChainScope varying the sampling rate.

Fig. 15. Root cause analysis for the Hotel Reservation mesh under inflated latency at the *geo* and *memcached-rate* services. Plots show the duration of relevant services for the top 2% slowest traces.

one sent by the client, breaking DeepFlow's mechanism for correlating spans and leading to wrong inter-service context propagation.

ChainScope maintains a high accuracy at a high concurrency level. The loss derives from the limited space in network headers for inter-node context propagation. As shown in Figure 14d, the inter-node loss rate drops as the sampling rate is reduced from 100% to 1%, while the intra-node loss rate remains near zero. Therefore, by controlling the sampling rate, ChainScope can achieve high tracing accuracy even in complex, high-concurrency gRPC scenarios.

5.3 End-to-end use case: root cause analysis (*Q.4 Q.5*)

Finally, we show how the performance of ChainScope translates into practical debugging benefits for a cloud operator. To do so, we evaluate ChainScope with a use case of root cause analysis where the operator should *localize which service is causing tail latency for the slowest 2% requests* (similarly to [3]). We use the Hotel Reservation service mesh, and emulate an anomaly by inflating the processing time of the *geo* and *memcached-rate* services by 25 to 30 ms (uniformly distributed) for 3% of randomly selected requests. For these tests, we generate ≈1.5K requests per second on the `/hotels` endpoint[12] using `ab` with 32 concurrent requests.

Analysis methodology. We collect spans with different non-intrusive (i.e., DeepFlow, Beyla[13], and ChainScope) and intrusive (i.e., Jaeger) tracing systems, and analyze them to identify the culprit service. Ideally, this requires reconstructing complete traces out of the collected spans, determining the real duration of each operation by subtracting those of direct children spans, and isolating services that exhibit an anomalous heavy-tailed distribution. However, this requires full coverage and correct context propagation across the services. Figure 15a shows the duration of the relevant involved services estimated using each tracing system. For each service, we only plot samples belonging to the slowest 2% requests. Since Jaeger uses code-level instrumentation, we also use it as the ground truth where possible; however, due to the lack of an SDK for plain C language, it could not be used to instrument the *memcached* service. Therefore, no data are reported for this span in the plot. For this specific case, it was still possible to manually reconstruct the ground truth with prior knowledge about the service chain and the randomly injected latency.

Results. As already observed in Section 5.2, DeepFlow fails to determine the parent-child relationships of spans in this high concurrency setting. In Figure 15a, DeepFlow reports multiple high latency services among the top 2%, as they were wrongly regarded as separate requests and the operator could not derive the real service duration when child spans are missing. Beyla reports

[12]The service chain behind this endpoint is described in [3]; *memcached-rate* is located just behind the *rate* service.

[13]For a fair comparison, in contrast with Section 5.1, we extend Beyla with TCP/IP context propagation for non-multiplex protocols, and keep the application-layer injection methodology for gRPC, for full coverage.

many incomplete traces due to limited inter-service context propagation over gRPC. This leads to wrongly considering many spans as leaves and attributing them the latency of their lost children (i.e., *search* service in this case). On the contrary, by reconstructing full traces, ChainScope reports duration distributions that are close to the ground truth and can effectively reveal *geo* and *memcached-rate* as the culprits; furthermore, Figure 15b shows that sampling does not affect much the reported distributions, therefore maintaining a similar ability to identify the sources of the long tail requests. Finally, note that even an intrusive tracing framework like Jaeger may lead to wrong conclusions due to incomplete coverage: since the *memcached* service falls out of the observability range, Jaeger wrongly reports anomalous durations for the parent; this highlights the value of ChainScope as a zero-effort observability baseline that avoids blind spots through broad-coverage tracing, while reserving intrusive tools for smaller-scoped monitoring of specific services.

6 Related work

Distributed tracing has evolved significantly, with a focus on improving accuracy, reducing overhead, and enhancing automation. Our work primarily distinguishes itself by its novel non-intrusive, protocol-independent approach, which leverages specific low-level kernel mechanisms to achieve high-accuracy traces without requiring application modification while minimizing overhead.

eBPF-Based Observability Tools. The rise of eBPF has enabled a new class of powerful observability tools. For example [5, 44, 49], focus on collecting performance metrics, eBPF's low-level access to the kernel makes it ideal for non-intrusive tracing. Like other distributed tracing systems, Chainscope leverages eBPF's capabilities to hook into system calls and kernel events precisely, providing a robust, language-agnostic foundation for distributed tracing.

Intrusive Distributed Tracing Systems. Traditional distributed tracing systems like Dapper [36], Zipkin [31], and Jaeger [20] are foundational to performance diagnosis in distributed environments. These systems rely on manual instrumentation in the application source code to insert trace IDs and span IDs, which are then passed between services to correlate requests. More recently, alternative tracing systems have extended instrumentation beyond the application layer. For instance, BufferScope [11] and Fathom [32] extend instrumentation to the network layer, aiming to generate higher-quality trace data for faster performance diagnosis. Other systems, like Astraea [40], Mint [16], Hindsight [45], and Canopy [21], focus on reducing the overhead of trace generation, collection, and storage while maintaining the quality of the trace for performance diagnosis. However, a common drawback across all these approaches is that their effectiveness and the correctness of the generated trace ultimately rely on the manual effort of developers. ChainScope limits the scope of manual instrumentation by providing a zero-effort observability baseline that correctly identifies the problematic microservice.

Non-Intrusive Distributed Tracing. A significant body of work focuses on non-intrusive distributed tracing. Systems such as PreciseTracer [34], RepTracer [43], and Magpie [4], infer intra-service context propagation by monitoring low-level kernel events such as thread spawn / synchronization, and message passing activities. While highly accurate for applications that adhere to their underlying assumptions, these systems often fall short in asynchronous or event-driven architectures, e.g., `epoll`, coroutines. Other non-intrusive systems, such as DeepTrace [12], TraceWeaver [3], RAKE [47], DeepFlow [35], and 5GCTracer [46], rely on implicit context propagation by leveraging application-specific semantics, payload inspection such as DeepTrace [12], or statistical analysis to infer request causality. This requires complex trace reconstruction, which is impossible with encrypted HTTPS/TLS traffic if payload inspection is needed, as well as complete event collection, thus preventing the use of head sampling to control processing overhead. Furthermore, without additional application-level mechanisms, e.g., `uprobes`, RequestID propagation, direct knowledge

of application semantics, these methods cannot sustain high accuracy, especially in complex scenarios. Close to ChainScope, systems like ZeroTracer [42], Beyla [14], PiCoP [29], X-Trace [9], Causeway [7], and BufferScope [11] utilize in-kernel instrumentation or sidecar agent for explicit context propagation, offering a language-independent approach without requiring application modification. Contrarily to ChainScope, these tools sacrifice performance or accuracy when dealing with complex microservice architectures, e.g., asynchronous processing, RPC, etc.

7 Discussion and Conclusion

In this paper, we introduced ChainScope, a non-intrusive distributed tracing system designed to deliver high-accuracy, full-scope observability for complex microservice architectures. By leveraging eBPF, ChainScope establishes an in-kernel universal tracing plane, effectively balancing the trade-offs between accuracy and performance overhead, particularly in highly concurrent environments using gRPC and coroutine-based microservices.

While ChainScope significantly advances non-intrusive tracing, its design relies on certain assumptions that define its optimal deployment scope. First, because ChainScope propagates trace context explicitly via network packet headers, it is best suited for controlled data-center or on-premises deployments where network policies are configurable. In case the microservice chain extends to external environments, middleboxes such as firewalls or WAN optimizers might strip custom IP/TCP options, which would cause trace context loss and incomplete traces. Second, our current implementation focuses on TCP-level event hooks. Because the underlying IP-level tagging mechanism is transport-agnostic, extending ChainScope to support UDP-based protocols (e.g., QUIC, DNS) simply requires adapting the relevant event hooks. However, doing so introduces interesting challenges for trace reconstruction, as UDP lacks the native segment sequence numbers ChainScope currently relies on. Finally, ChainScope's baseline cf. section 3.2 intra-service propagation relies on a run-to-completion execution model. Although our epoll-enhanced FIFO mechanism successfully mitigates asynchronous out-of-order execution for most standard middlewares, highly unconventional applications that fundamentally break this FIFO assumption by reordering responses could still lead to incorrect trace reconstruction and require the use of ChainScope extensions cf. section 4.

Our evaluation cf. section 5 shows that ChainScope is unique in its ability to provide configurable overhead at the cost of a higher sampling rate thanks to its eBPF-based data collection. With a low sampling rate, in simple scenarios, ChainScope achieves higher accuracy with a much lower performance impact compared to competitors like DeepFlow and Beyla. For the complex gRPC-based DeathStarBench Hotel Reservation service mesh under high load, ChainScope maintains ≈80%-100% accuracy, while DeepFlow and Beyla reached only 1% and 45% accuracy, respectively, with a much greater performance impact. Furthermore, using a realistic use case, we show that ChainScope effectively serves as a zero-effort baseline to identify rare bottleneck events. Ultimately, ChainScope demonstrates that by moving context propagation and intelligent sampling directly into the kernel, cloud operators can achieve high-fidelity, zero-effort observability without the costly burden of manual application instrumentation.

This work does not raise any ethical issues.

Acknowledgments

This work was supported in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2023B1515020054 and the National Natural Science Foundation of China under Grant 62272495.

References

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 74–89. doi:10.1145/945445.945454

[2] Apache. [n. d.]. Apache SkyWalking Application performance monitor tool for distributed systems. https://skywalking.apache.org/.

[3] Sachin Ashok, Vipul Harsh, Brighten Godfrey, Radhika Mittal, Srinivasan Parthasarathy, and Larisa Shwartz. 2024. TraceWeaver: Distributed Request Tracing for Microservices Without Application Modification. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 828–842. doi:10.1145/3651890.3672254

[4] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for request extraction and workload modelling.. In *OSDI*, Vol. 4. 18–18.

[5] Soeren Becker, Robin Goegge, and Odej Kao. 2024. Measuring Application Interference With System-Level Instrumentation. In *2024 IEEE International Conference on Big Data (BigData)*. IEEE, 3648–3653.

[6] Brendan Gregg. [n. d.]. Linux uprobe: User-Level Dynamic Tracing. https://www.brendangregg.com/blog/2015-06-28/linux-ftrace-uprobe.html.

[7] Anupam Chanda, Khaled Elmeleegy, Alan L Cox, and Willy Zwaenepoel. 2005. Causeway: Support for controlling and analyzing the execution of multi-tier applications. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 42–59.

[8] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering* (2017), 195–216.

[9] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. 2007. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. USENIX Association, Cambridge, MA. https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework

[10] 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* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. doi:10.1145/3297858.3304013

[11] Kaihui Gao, Chen Sun, Shuai Wang, Dan Li, Yu Zhou, Hongqiang Harry Liu, Lingjun Zhu, and Ming Zhang. 2022. Buffer-based End-to-end Request Event Monitoring in the Cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 829–843. https://www.usenix.org/conference/nsdi22/presentation/gao-kaihui

[12] Yantao Geng, Han Zhang, Zhiheng Wu, Yahui Li, Jilong Wang, and Xia Yin. 2025. Low-Overhead Distributed Application Observation with DeepTrace: Achieving Accurate Tracing in Production Systems. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 1056–1069.

[13] Golang Community. [n. d.]. Golang ABI Specification. https://go.googlesource.com/go/+/refs/heads/dev.regabi/src/cmd/compile/internal-abi.md.

[14] Grafana Labs. [n. d.]. Beyla – eBPF application auto-instrumentation. https://grafana.com/docs/beyla/latest/.

[15] grpc.io. [n. d.]. opensource google RPC framework. https://grpc.io/.

[16] 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* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 683–697. doi:10.1145/3669940.3707287

[17] 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, Article 22 (July 2024), 21 pages. doi:10.1145/3643748

[18] 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)*. 436–446. doi:10.1109/ICWS53863.2021.00063

[19] Igor Sysoev. [n. d.]. nginx ("engine x") is an HTTP web server, reverse proxy, content cache, load balancer, TCP/UDP proxy server, and mail proxy server. https://nginx.org/en/.

[20] Jaegertracing. [n. d.]. Jaeger: open source, distributed tracing platform. https://www.jaegertracing.io/.

[21] Jonathan Kaldor, Jonathan Mace, Michał 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* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 34–50. doi:10.1145/3132747.3132749

[22] Ashwin Kumar, Abhik Bose, Khushboo Tiwari, Arnav Mishra, Abhishek Dixit, Abuhujair Khan, and Mythili Vutukuru. 2024. Feasibility of Application Layer Header Parsing in eBPF and P4. In *2024 IFIP Networking Conference (IFIP Networking)*. 475–481. doi:10.23919/IFIPNetworking62109.2024.10619855

[23] Pedro Las-Casas, Jonathan Mace, Dorgival 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* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 326–332. doi:10.1145/3267809.3267841

[24] James Lewis and Martin Fowler. 2014. Microservices: a definition of this new architectural term. *MartinFowler. com* 25, 14-26 (2014), 12.

[25] Linux. [n. d.]. Linux eBPF document. https://docs.ebpf.io/linux/.

[26] Lyft. [n. d.]. Envoy proxy. https://www.envoyproxy.io/.

[27] Anders Nõu, Sacheendra Talluri, Alexandru Iosup, and Daniele Bonetta. 2025. Investigating Performance Overhead of Distributed Tracing in Microservices and Serverless Systems. In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering* (Toronto ON, Canada) *(ICPE '25)*. Association for Computing Machinery, New York, NY, USA, 162–166. doi:10.1145/3680256.3721316

[28] Odigos. [n. d.]. an eBPF-based solution providing zero-code, zero-performance overhead for deeper tracing. https://odigos.io/.

[29] Hiroya Onoe, Daisuke Kotani, and Yasuo Okabe. 2025. PiCoP: Service Mesh for Sharing Microservices in Multiple Environments using Protocol-Independent Context Propagation. *IEEE Transactions on Cloud Computing* (2025).

[30] The opentelemetry community. [n. d.]. OpenTelemetry zero-code instrumentation. https://opentelemetry.io/docs/zero-code.

[31] OpenZipkin. [n. d.]. Zipkin – a distributed tracing system. https://zipkin.io/.

[32] Mubashir Adnan Qureshi, Junhua Yan, Yuchung Cheng, Soheil Hassas Yeganeh, Yousuk Seung, Neal Cardwell, Willem De Bruijn, Van Jacobson, Jasleen Kaur, David Wetherall, and Amin Vahdat. 2023. Fathom: Understanding Datacenter Application Network Performance. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 394–405. doi:10.1145/3603269.3604815

[33] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. 2006. WAP5: black-box performance debugging for wide-area systems. In *Proceedings of the 15th International Conference on World Wide Web* (Edinburgh, Sczotland) *(WWW '06)*. Association for Computing Machinery, New York, NY, USA, 347–356. doi:10.1145/1135777.1135830

[34] Bo Sang, Jianfeng Zhan, Gang Lu, Haining Wang, Dongyan Xu, Lei Wang, Zhihong Zhang, and Zhen Jia. 2012. Precise, Scalable, and Online Request Tracing for Multitier Services of Black Boxes. *IEEE Transactions on Parallel and Distributed Systems* 23, 6 (2012), 1159–1167. doi:10.1109/TPDS.2011.257

[35] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, Mingwei Xu, Yahui Li, Jiping Yin, Jianchang Song, Zhuofeng Li, and Runjie Nie. 2023. Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 420–437. doi:10.1145/3603269.3604823

[36] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc. http://research.google.com/archive/papers/dapper-2010-1.pdf

[37] Byung Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N. Chang. 2009. vPath: precise discovery of request processing paths from black-box observations of thread and network activities. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (San Diego, California) *(USENIX'09)*. USENIX Association, USA, 19.

[38] The Apache software foundation. [n. d.]. Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/current/programs/ab.html.

[39] The opentelemetry community. [n. d.]. OpenTelemetry zero-code instrumentation for golang. https://github.com/open-telemetry/opentelemetry-go-instrumentation.

[40] M. Toslali, S. Qasim, S. Parthasarathy, F. A. Oliveira, H. Huang, G. Stringhini, Z. Liu, and A. K. Coskun. 2024. Unleashing Performance Insights with Online Probabilistic Tracing. In *2024 IEEE International Conference on Cloud Engineering (IC2E)*. 72–82. doi:10.1109/IC2E61754.2024.00015

[41] Shuaiyu Xie, Jian Wang, Maodong Li, Peiran Chen, Jifeng Xuan, and Bing Li. 2025. TracePicker: Optimization-Based Trace Sampling for Microservice-Based Systems. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE081 (June 2025), 22 pages. doi:10.1145/3729351

[42] Wanqi Yang, Pengfei Chen, Kai Liu, and Huxing Zhang. 2025. ZeroTracer: In-Band eBPF-Based Trace Generator With Zero Instrumentation for Microservice Systems. *IEEE Transactions on Parallel and Distributed Systems* 36, 7 (2025), 1478–1494. doi:10.1109/TPDS.2025.3571934

[43] Yong Yang, Long Wang, Jing Gu, and Ying Li. 2023. Capturing Request Execution Path for Understanding Service Behavior and Detecting Anomalies Without Code Instrumentation. *IEEE Transactions on Services Computing* 16, 2 (2023), 996–1010. doi:10.1109/TSC.2022.3149949

[44] Yi Zhai, Junzhou Luo, and Jianrui Liu. 2025. NRCAC: Non-Intrusive Microservice Root Cause Analysis Framework for Cloud Providers. In *IEEE INFOCOM 2025 - IEEE Conference on Computer Communications*. 1–10. doi:10.1109/INFOCOM55648.2025.11044716

[45] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. 2023. The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 321–339. https://www.usenix.org/conference/nsdi23/presentation/zhang-lei

[46] Anping Zhao, Frederic Desnoes, Ilhem Fajjari, and Nadjib Aitsaadi. 2025. 5GC-Tracer: A Novel Non-Intrusive Distributed Tracing for Enhanced 5G Core Network Observability. In *NOMS 2025-2025 IEEE Network Operations and Management Symposium*. 1–10. doi:10.1109/NOMS57970.2025.11073675

[47] Yao Zhao, Yinzhi Cao, Yan Chen, Ming Zhang, and Anup Goyal. 2012. Rake: Semantics assisted network-based tracing framework. *IEEE Transactions on Network and Service Management* 10, 1 (2012), 3–14.

[48] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, Xiaozheng Lai, Dan Williams, and Andi Quinn. 2025. Extending Applications Safely and Efficiently. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 557–574.

[49] Yusheng Zheng, Tong Yu, Yiwei Yang, and Andrew Quinn. 2025. wBPF: Efficient Edge-Case Observability for CXL Pooling systems via eBPF. In *Proceedings of the 4th Workshop on Heterogeneous Composable and Disaggregated Systems (HCDS '25)*. Association for Computing Machinery, New York, NY, USA, 67–72. doi:10.1145/3723851.3726985

Received December 2025; accepted April 2026