

Low-Overhead Distributed Application Observation with DeepTrace: Achieving Accurate Tracing in Production Systems

Yantao Geng¹*, Han Zhang¹*, Zhiheng Wu¹, Yahui Li¹, Jilong Wang¹, Xia Yin²

¹Institute for Network Sciences and Cyberspace, Tsinghua University

²Department of Computer Science and Technology, Tsinghua University

*zhhan@tsinghua.edu.cn °gyt22@mails.tsinghua.edu.cn

Abstract

As microservices grow in scale and complexity, their operation and debugging become increasingly challenging. Even a single user request can involve interactions across hundreds of components. In such intricate systems, distributed tracing, which tracks the end-to-end execution flow of requests, has become a critical monitoring tool. Among these, non-intrusive tracing frameworks that do not require code modification are particularly valued for their convenience. However, existing non-intrusive solutions either have limited applicability or lack sufficient accuracy under high concurrency. To address these challenges, we propose DeepTrace, a transaction-based, non-intrusive distributed tracing framework designed for microservices. DeepTrace leverages API endpoints and transaction fields embedded within request content to categorize requests into distinct transactions, thereby reducing the likelihood of incorrectly merging traces from different transactions. Compared to state-of-the-art frameworks, DeepTrace maintains an accuracy rate of over 95% even under high concurrency. It has also been adopted by dozens of companies in their production systems for tasks such as failure diagnosis and resource optimization.

CCS Concepts

• **Networks** → **Network monitoring**; **Network performance analysis**; **Network measurement**.

Keywords

Distributed Tracing; Application Performance Monitoring; Network Performance Monitoring; eBPF

ACM Reference Format:

Yantao Geng¹*, Han Zhang¹*, Zhiheng Wu¹, Yahui Li¹, Jilong Wang¹, Xia Yin². 2025. Low-Overhead Distributed Application Observation with DeepTrace: Achieving Accurate Tracing in Production Systems. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25)*, September 8–11, 2025, Coimbra, Portugal. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3718958.3750477>

1 Introduction

In recent years, online services have undergone a significant architectural evolution, transitioning from monolithic structures to

Han Zhang is the corresponding author.



This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

SIGCOMM '25, Coimbra, Portugal

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1524-2/25/09

<https://doi.org/10.1145/3718958.3750477>

distributed microservices frameworks [15, 28, 56, 76]. This architecture decoupling inherently enables elastic horizontal scaling capabilities surpassing monolithic systems's vertical limitations. However, this fragmentation also creates non-deterministic failure modes, particularly complicating root cause analysis of tail latency anomalies and performance degradation patterns [7, 12]. How to troubleshoot microservice failures is becoming an essential challenge.

Distributed tracing frameworks (e.g., DeepFlow [73], Jaeger [37]) have emerged as a critical tool for microservice failure diagnosis by providing comprehensive visibility into request execution paths. A common format for a trace is a span tree, where *spans* serve as nodes and *parent-child relationships* as edges [59, 75]. Each span corresponds to a request-response pair, capturing the execution lifecycle of a single request within a component. Parent-child relationships emerge when an incoming request's span (parent) initiates one or more outgoing requests' spans (childs) during its processing. Non-intrusive distributed tracing frameworks (e.g., DeepFlow [73], TraceWeaver [3]) demonstrate superior operational efficiency and scalability advantages compared to the intrusive distributed tracing frameworks by eliminating the need for code instrumentation across thousands of microservice components [18, 23, 68, 73, 74]. This zero-code instrumentation capability proves particularly critical in modern cloud-native environments, where heterogeneous service architectures frequently incorporate unmodifiable closed-source or third-party components [20, 22, 69] that fundamentally preclude traditional code modification approaches.

However, achieving accurate, non-intrusive distributed tracing is challenging. The current non-intrusive distributed tracing frameworks [3, 67, 70, 73] can be categorized into FIFO-based correlation frameworks and delay-based correlation frameworks according to their span correlation strategies. Although most of them have already shown high performance in request tracing, they are still inadequate in the following two key aspects:

First, most existing *FIFO-based correlation frameworks* might **exhibit incoming and outgoing request mapping ambiguities** due to concurrent processing of multi-threads. Existing FIFO-based frameworks, including vPath [78] and DeepFlow [73], are fundamentally predicated on a sequential execution model. These systems operate under the operational premise that a dedicated thread exclusively processes each incoming request along with its subsequent sub-requests through linear execution, maintaining strict temporal isolation between distinct requests. Although this mechanism has high correlation efficiency, their serial processing paradigms are systematically violated in concurrent execution environments where multiple requests may interleave within shared thread resources through asynchronous scheduling. This fundamental discrepancy

between design assumptions and runtime concurrency realities critically compromises their general applicability, resulting in erroneous request linkage during distributed tracing operations.

Second, the majority of *delay-based correlation frameworks* have **large overhead and performance fluctuation under high request concurrency**. Delay-based correlation schemes, exemplified by WAP5 [67] and TraceWeaver [3], operate under the assumption that parent-child request pairs exhibit inter-arrival delays conforming to a characteristic probability distribution. These methodologies employ statistical estimation techniques to model the delay distribution and subsequently establish causal relationships between request pairs demonstrating high-probability temporal relationships. While theoretically applicable to diverse threading architectures, their sole reliance on temporal delay metrics incurs substantial performance penalties in high-concurrency environments, as all span correlation operations must be processed through a centralized server. Moreover, increasing concurrency elevates request density on the timeline, creating multiple temporally adjacent parent candidates for individual outbound requests. The observed parent-child delay often deviates significantly from the peak of the latency distribution pattern. This mismatch amplifies the probability of misidentifying parent requests exponentially with candidate pool expansion, thereby propagating correlation inaccuracies.

DeepTrace framework. To address these challenges, we present DeepTrace, a non-intrusive distributed tracing framework (§ 3), leveraging the transaction analysis at the end host. DeepTrace achieves both *efficient* and *accurate* end-to-end request tracing as they propagate across heterogeneous service components with the following designs:

Protocol-aware span construction (§ 3.1). Existing intrusive distributed tracing frameworks exemplified by Jaeger [37] necessitate manual code instrumentation to enable cross-protocol trace context propagation and span generation, an approach that introduces substantial implementation complexity and maintenance overhead in production environments.

DeepTrace proposes a novel eBPF-driven monitoring framework that achieves non-intrusive observation of distributed system communications through kernel-level packet capture and protocol-aware analysis. DeepTrace incorporates protocol parsing templates derived from systematic examination of RFC documents [24, 25, 34–36, 60, 66] and open-source implementations (e.g., Zeek [82], Suricata [77]), currently supporting more than 20 mainstream application-layer protocols. To accelerate the span construction, DeepTrace proposes a signature-based protocol identification through fixed-position field matching combined with semantic-aware parsing that utilizes native protocol length indicators, enabling both efficient packet processing and accurate request boundary detection. This technical advancement fundamentally addresses the protocol coverage limitation inherent in existing non-intrusive tracing solutions (e.g., TraceWeaver [3], vPath [78]), which typically focus on basic RPC/HTTP analysis or employ computationally expensive full-payload inspection [73].

Transaction-based span association mechanism (§ 3.2). To accurately associate spans with parent-child relationships, the advanced mechanisms [3, 73] advocate using the stateless features (e.g., delay, arrival sequence) to infer parent-child relationships

between incoming and outgoing requests at servers but suffer from significant storage overhead and inference inaccuracies.

Our empirical analysis reveals that spans within the same transaction (i.e., logical operation group) exhibit a stronger correlation likelihood than cross-transaction spans (§ 2). Building upon this finding, DeepTrace introduces a transaction-based correlation mechanism that employs a dual-phase approach: First, we compute transaction affinity scores through semantic analysis of contextual features (e.g., user identifiers in HTTP headers, session tokens), quantifying the likelihood of spans sharing common transactional context. Second, these scores are integrated with multidimensional observational evidence - including temporal proximity metrics (e.g., execution time windows), causal dependency patterns (e.g., causal delay between parent and child spans), and behavioral signatures - through a probabilistic fusion model to establish robust span associations. This approach prioritizes transactional semantics when explicit identifiers are present in payload content while maintaining backward compatibility. When transaction context is unavailable, DeepTrace automatically rebalances feature weights by reducing reliance on transaction affinity and emphasizing causality-related metrics such as temporal sequence alignment and dependency pattern matching. The adaptive weighting mechanism ensures stable correlation performance across both transaction-explicit and transaction-agnostic scenarios.

Query-driven trace assembling (§ 3.3). Traditionally, distributed tracing frameworks typically collect all spans centrally for trace assembly, which can incur substantial transmission and storage overhead [59, 75]. While some solutions temporarily store spans locally on hosts for subsequent selective centralized collection, this approach still imposes significant memory pressure due to the accumulation of spans [10, 45, 84].

In contrast, DeepTrace advocates efficient on-host span compression and caching. To reduce caching overhead and accelerate queries, it indexes spans based on their tags and metrics, thereby avoiding redundant caching of identical tags or similar metrics. DeepTrace takes the query conditions specified by the operator as input and iteratively extracts spans of the target traces that meet these conditions from the hosts to complete trace assembly. This approach avoids the overhead of collecting and transmitting a large number of spans that are not of interest to operators and supports various query conditions, including metric queries and tag queries.

Contributions. DeepTrace has proven its value in the production environments of dozens of companies. DeepTrace is now publicly available at <https://github.com/DeepShield-AI/DeepTrace>. In summary, our main contributions are:

- We undertake an in-depth study of the existing distributed tracing framework and present a new perspective on the design of span correlation (§ 2).
- We design and implement DeepTrace, a transaction-based non-intrusive distributed tracing framework to provide efficient and accurate distributed application observation (§ 3).
- We conduct a comprehensive evaluation of DeepTrace in both production (§ 4) and testbed (§ 5). Our testbed evaluations show that DeepTrace has stable performance, and it can achieve a

tracing accuracy of over 95% under high-concurrency scenarios and successfully complete various real-world tasks with a probability of over 90%.

2 Background and Motivation

This section begins with an introduction to distributed tracing in microservices (§ 2.1). Then, we summarize the challenges of non-intrusive distributed tracing (§ 2.2) and provide the motivation for DeepTrace (§ 2.3).

2.1 Distributed tracing in microservice

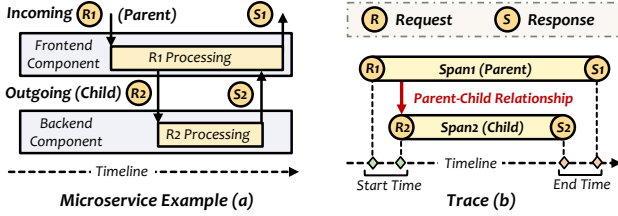


Figure 1: An example of a trace (b) obtained from a microservices application example (a).

Figure 1 illustrates a trace obtained from a simple microservice application. A trace is typically represented as a *span tree*, where nodes are *spans* and edges denote *parent-child relationships* [37, 59, 75]. Each **span** represents a request-response pair, starting with a request and ending with the corresponding response. It documents the execution lifecycle of a single request within a component. For example, within the Frontend component, request R1 and response S1 constitute Span1. Spans contain metadata, such as the caller, callee, start time, and end time. Within a component, an incoming request and its response form an *incoming span*, while an outgoing request and its response form an *outgoing span*. An incoming span and *one or multiple* outgoing spans triggered during the processing of the incoming span form a **parent-child relationship**. The parent-child relationships between spans are equivalent to the parent-child relationships between the *requests of the spans*. In the above example, to process and respond to the incoming request R1, the Frontend component invokes the Backend component through a child outgoing request R2. This makes the outgoing Span2 a child span of the incoming Span1. A parent span and one of its child spans form a **span mapping**, e.g., (Span1, Span2). The trace of a request *r* is the tree consisting of the span corresponding to *r* and all descendant spans recursively generated by it.

2.2 Non-Intrusive distributed tracing challenges

The primary challenge in non-intrusive distributed tracing lies in accurately reconstructing causal relationships between requests across components employing concurrent processing, constrained by tools like strace [43] and eBPF [40, 52] that only observe *system-level manifestations* - including network traffic characteristics (message timing, payload size, and content fragments) and thread scheduling patterns. While these external observations enable basic span construction through request-response pairing, they fundamentally fail to capture the application's internal request routing logic and

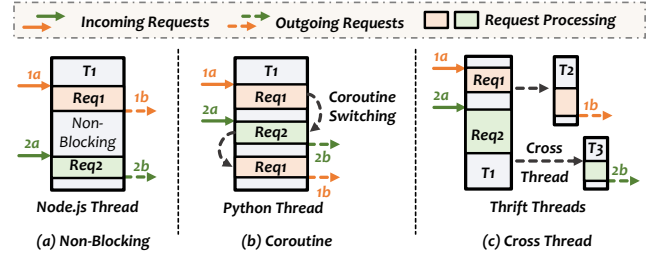


Figure 2: Different concurrent processing mechanisms.

are confounded by concurrency mechanisms (thread pools, async I/O, event loops) that create temporal overlaps between independent processing chains. This dual opacity of hidden internal logic and concurrent execution interference renders timing-based correlations ineffective, particularly in microservice architectures where multiple overlapping parent requests may trigger distinct downstream calls without explicit context propagation. The resulting ambiguity in determining which incoming requests spawn specific outgoing requests undermines the construction of accurate end-to-end transaction traces, as illustrated in Figure 2, ultimately compromising distributed system analysis capabilities that depend on reliable causal relationships.

In the non-blocking I/O scenario, thread T1 sends an outgoing request 1b and then processes a new incoming request 2a without awaiting the response from 1b. This is common in components using event-driven asynchronous communication, such as Node.js servers. Simply linking requests handled by the same thread could mistakenly merge spans from separate traces. Microservices often incorporate numerous components based on programming languages like Go and Python, which employ multiple distinct user-space coroutines to multiplex a single thread. The switching between these coroutines, as well as the mapping relationship between coroutines and threads, is governed by invisible application logic. It is unclear which internal coroutine receives or sends the incoming and outgoing requests, thereby making request association difficult. In cross-thread scenarios, the workflow involves complex handoffs among different threads using mechanisms such as task queues, channels, or semaphores. Notable examples include the handoffs between communication threads in RPC libraries such as Thrift [1] and gRPC [29]. Non-intrusive tracing tools fail to record the handoffs from T1 to T2 when a parent request 1a and its child request 1b are handled by two different threads, T1 and T2. Consequently, correctly associating requests 1a and 1b becomes challenging in a non-intrusive manner.

2.3 Motivating DeepTrace

Over the years, there has been considerable interest in non-intrusive distributed tracing for microservice troubleshooting [2, 8, 67, 70, 73, 81]. However, these schemes often struggle with accurately associating incoming and outgoing spans, which can lead to irregular trace structures or the loss of critical failure data. Such inaccuracies can significantly hinder the failure diagnosis. In this paper, we classify existing non-intrusive tracing frameworks into two categories based on *span correlation strategies*: **FIFO-based**

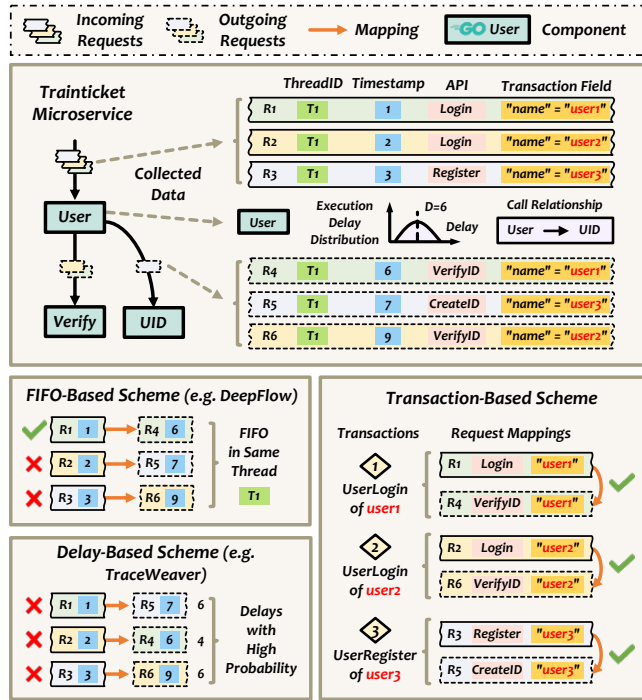


Figure 3: Comparison between FIFO-based, delay-based, and transaction-based schemes.

correlation and delay-based correlation approaches. Figure 3 shows their principles and limitations.

◊ **FIFO-based correlation schemes are only effective in specific threading models.** Modern distributed tracing paradigms (e.g., vPath [78], DeepFlow [73]) fundamentally rely on the *sequential request execution assumption*, where single-threaded processing enforces strict FIFO ordering of requests and their derived sub-requests, postponing new requests until full completion of current computational/I/O operations. While this model enables efficient causal inference through linear thread-local event streams, modern concurrency mechanisms—particularly coroutine scheduling as analyzed in § 2.2—systematically violate its core premises. For the example shown in Figure 3, in the User component, which uses coroutines to manage multiple incoming requests on a single thread, request R2 arrives second, but its child R6 is sent out third, leading to the incorrect association of R2 with the second outgoing R5.

◊ **Delay-based correlation schemes exhibit significant overhead and performance variability under high concurrency scenarios.** Delay-based tracing schemes, including TraceWeaver [3] and WAP5 [67], correlate parent-child requests using temporal delay distributions but exhibit critical limitations in high-concurrency environments. Their centralized data collection for delay estimation and trace assembly creates scalability bottlenecks, incurring prohibitive network overhead as request volume escalates [38, 51]. Concurrently, increased temporal request density amplifies statistical ambiguity: multiple candidate parents fall within typical delay intervals for each child request, while real-world delay variance allows non-parent candidates to coincidentally match peak probability ranges. The simultaneous occurrence of exponential growth

in false correlations and deviation from optimal delay patterns systematically undermines tracing accuracy. The primary weakness arises from an exclusive reliance on temporal proximity, which fails to provide the necessary discriminative power to differentiate causal relationships amid concurrent request interleaving.

◊ **Our approach: transaction-based correlation.** In this study, we advocate analyzing the content correlation of requests *locally on each host* to reduce the data transmission overhead. We define a **transaction category** (abbreviated as **transaction**) as a logically related set of inbound and outbound spans (or requests) that demonstrate semantic coherence through shared contextual attributes. The semantic coherence is determined based on the requests that either exhibit *nested API endpoint relationships* or maintain *persistent transaction fields* across multiple requests. For instance, in Figure 3, the Login endpoint, which handles user logins, must call the VerifyID endpoint to validate user IDs. The nested API endpoint relationship Login→VerifyID represents a transaction (e.g., the Userlogin transaction). In distributed microservices architectures, persistent transaction fields constitute critical transactional elements that are systematically captured from incoming requests and dynamically propagated to subsequent service invocations through component-level processing logic. These fields enable finer-grained transactions, such as the UserLogin for User1. This propagation mechanism maintains transactional integrity across service boundaries by preserving essential contextual information throughout the distributed transaction lifecycle, thereby enabling atomic operation sequences despite the decentralized nature of microservice interactions. Common examples include user ID [21, 61], order information [62, 71], and access tokens [26, 72].

3 The DeepTrace Framework

DeepTrace goals. We intend to achieve the following goals:

- **Goal 1: Multi-protocol compatibility.** Simultaneously support request data collection for multiple common application protocols.
- **Goal 2: Precise end-to-end tracing.** Accurately infer the parent-child relationship between requests to reconstruct the request trace with high precision.
- **Goal 3: High-efficiency operation.** Rapidly build span, reconstruct trace, and respond to users' trace queries.
- **Goal 4: Negligible performance overhead.** For continuous, long-term monitoring, reduce data transfer and storage overhead as much as possible.

DeepTrace workflow. Figure 4 illustrates the workflow of DeepTrace, which consists of two main components: an **agent** deployed on each host and a **server** operating across the microservice cluster. To ensure tracing *accuracy*, the DeepTrace agent performs non-intrusive span construction and infers parent-child relationships between spans. To maintain tracing *efficiency*, the agent compresses, indexes, and caches spans based on *tags and metrics*, while the server selectively collects spans from agents in a centralized manner according to user queries.

The DeepTrace server receives *adjustable settings* from operators as **inputs** and relays them to its agents across various hosts (①). These settings include the *quantity interval* (default is 100) and

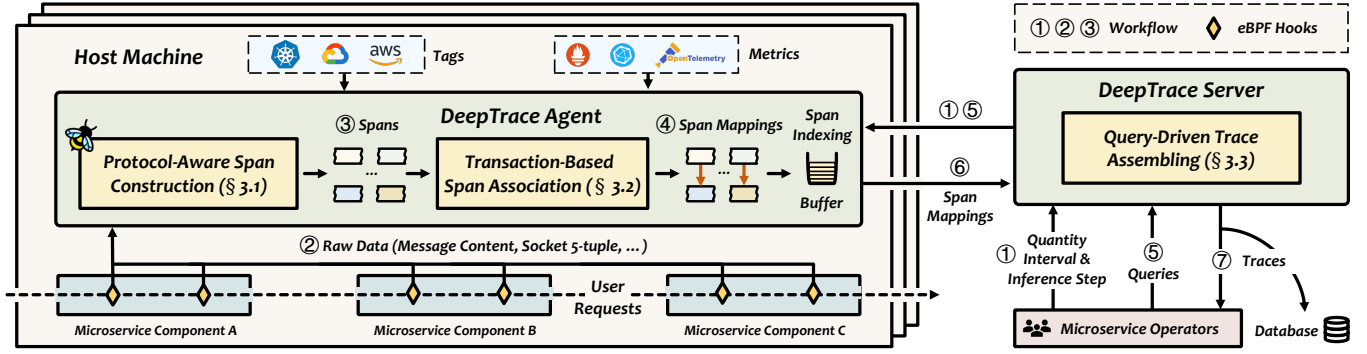


Figure 4: The overview of DeepTrace framework.

the *inference step* (default is 10). The quantity interval divides sequentially generated spans of each component into groups of equal size, within which DeepTrace performs span correlation, while the inference step represents the number of span mappings deduced by DeepTrace in each iteration. Upon initialization, DeepTrace activates **protocol-aware span construction** (§ 3.1) within agents to collect raw data about requests, such as socket 5-tuples (source IP, destination IP, source port, destination port, and transport protocol) and message content (2). Agents use this data to construct spans, enriching them with metrics and tags from third-party frameworks or cloud platforms (3). The subsequent steps involve **transaction-based span association** (§ 3.2), where DeepTrace infers the transaction types and multidimensional metric distributions of spans and uses them to infer span mappings with parent-child relationships (4). During subsequent **query-driven trace assembling** (§ 3.3), the DeepTrace agent compresses spans and builds an index in shared memory based on their tags and metrics, while the DeepTrace server continuously collects span mappings relevant to operator queries (5) from agents to construct traces (6). The assembled traces are returned to the operators and persistently saved to the database (7). Spans that remain on the agents are typically discarded or retained according to the preferences of the operators.

3.1 Protocol-Aware Span Construction

Given that microservices process numerous requests across various application protocols, DeepTrace must efficiently parse and accurately match requests and responses from different protocols to construct spans (**Goals 1 & 3**). These spans should include critical details such as API endpoints and request content, establishing a data foundation for subsequent span correlation. However, intrusive distributed tracing solutions like Jaeger [37] depend on manual code instrumentation to generate cross-protocol spans, which introduces substantial implementation complexity and maintenance burdens in production environments. Meanwhile, non-intrusive tracing methods analyze requests by intercepting communication traffic, but their capabilities are often limited to inherent protocol types, primarily focusing on basic gRPC or HTTP analysis [3, 78]. Alternatively, some approaches use computationally intensive full payload inspection, resulting in inefficient performance [73].

★ **DESIGN 1. Protocol template-based request parsing for non-intrusive span construction.** Firstly, DeepTrace leverages

Type	Selected Syscall Call ABIs					
Egress	write	sendto	sendmsg	sendmmsg	writv	
Ingress	read	recvfrom	recvmsg	recvmmsg	readv	

Table 1: DeepTrace selects ten system call types that are closely related to network communication behavior.

the eBPF mechanism [39, 40, 52] to non-intrusively gather *raw data* about requests. Inspired by DeepFlow [73], the execution of requests is primarily driven and constrained by narrow-waist communication behaviors in the transport layer API [50]. Therefore, we attach eBPF programs to the ten network-related system calls listed in Table 1 to collect raw data. DeepTrace currently does not handle *encrypted traffic*, but existing eBPF-based software solutions such as Pixie [45] and DeepFlow can intercept application-layer functions via eBPF’s uprobe [41] prior to data encryption, thereby capturing plaintext data. The raw data generated each time a system call is triggered by the application includes the PID (Process ID), system call type, socket five-tuple, timestamp, and message content. These raw data are then copied from the eBPF kernel program to user space for subsequent span construction.

Span construction procedure. Figure 5 shows that DeepTrace constructs spans in three phases: *preprocessing*, *request parsing*, and *request-response association*. In the preprocessing phase, DeepTrace sorts the message content in the raw data by timestamp and then groups the content by PID, socket five-tuple, and system call type. This grouping helps distinguish the message contents of different requests, reducing the complexity of subsequent request parsing.

Within each group, DeepTrace parses the message content to extract requests and responses. To handle the variety of application protocols in microservices, *protocol templates* are created based on publicly available RFC documents [24, 25, 34–36, 60, 66] and open-source implementations [77, 82]. DeepTrace currently supports parsing over 20 widely adopted standard public protocols, while proprietary custom protocols require operators to manually develop corresponding protocol templates. DeepTrace provides a user-friendly interface for seamless integration to new protocol templates. Each template includes *protocol type inference* and *content partitioning*. The protocol type inference template identifies the protocol type by specifying the value ranges of certain fields in the request message content for specific protocols. For example, the MongoDB protocol specifies that bytes 12 to 16 of each request

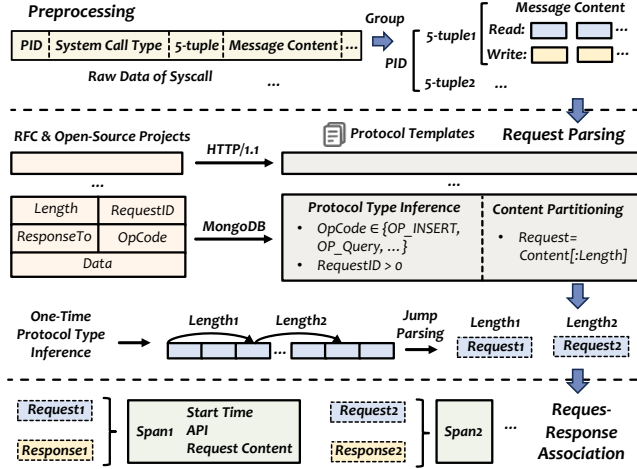


Figure 5: Three phases of span construction.

are the *OpCode* field, which should match database operations like *OP_INSERT*, *OP_DELETE*, and others as defined by the relevant protocol specifications. For each new group, DeepTrace iterates through common protocol templates to perform a *one-time* protocol type inference. Additionally, DeepTrace extracts APIs from request content according to the corresponding protocol types, such as URLs in HTTP protocols and method names in Thrift protocols.

Then DeepTrace segments the message content of distinct requests through content partitioning templates. Given the variable length of requests, large requests may be sent or received through multiple system calls. Additionally, the application layer’s request queuing mechanism can cause a single system call’s message content to include multiple small requests. Therefore, *considering the message content of a single system call as a single request is inaccurate*. To achieve accurate and efficient request segmentation, we perform *jump parsing based on request length*. We observe that in a single TCP connection of protocols without multiplexing and in a single stream of multiplexing protocols (such as HTTP2), *different requests are transmitted serially*. Therefore, we only need to determine the start and end positions of different requests in each group. We utilize the request length field in the protocol (for instance, the first four bytes in MongoDB requests) to quickly jump to the end of the current request, which is also the starting point of the next request. This approach avoids the inefficient traversal of every byte in the message content. For protocols that do not contain the request length field, such as Redis [69], we then fully parse the request according to the protocol specifications.

The final step involves constructing spans by associating requests and responses, taking into account different transmission modes. In *serial* transmission, each request within the same connection waits for the previous response to finish before being sent, as seen in HTTP/1.1 persistent connections. In *pipeline* mode, multiple requests can be sent consecutively within the same connection, with responses returned in the same order. For protocols in these two modes, DeepTrace matches the order of responses and requests within the same connection to construct spans. For protocols that achieve *parallel* transmission through multiplexed messages, DeepTrace uses embedded distinguishing attributes in the messages to

correlate requests and responses, such as the stream identifier in HTTP/2 headers [36].

Third-party framework and platform integrations. To further assist operators in troubleshooting, DeepTrace integrates tags and metrics from third-party frameworks or cloud platforms into spans. Tags help users immediately determine the locations of the problems. For example, users can query which pod has time-consuming invocations. DeepTrace allows the injection of Kubernetes resource tags (such as node, service, pod, etc.) and cloud resource tags (such as regions, availability zones, VPCs, etc.) into spans. Additionally, these tags are linked to metrics generated by frameworks like Prometheus [63], providing further details on spans.

3.2 Transaction-Based Span Association

After obtaining spans, DeepTrace needs to accurately associate spans with parent-child relationships to generate span mappings (Goal 2). To construct a transaction-based span association solution, DeepTrace first needs to infer the transaction type of spans, which includes nested API endpoint relationships and persistent transaction fields. The **key challenges** in inferring the transaction of spans stem from two main aspects. Firstly, the nested API endpoint relationships are determined by the invisible application logic of components. We have to infer the nested API endpoint relationships based on external information. In addition, the content of the incoming and outgoing requests typically includes many identical *schema fields*, such as parameter types, protocol version numbers, and XML elements. Simply using identical fields contained within the incoming and outgoing request contents as persistent transaction fields can erroneously merge spans from different transactions.

★ **DESIGN 2. Parent-child probability assessment based on transaction analysis.** The **core idea** of DeepTrace’s transaction inference is that requests belonging to the same transaction exhibit *stronger correlations in API traffic intensity and higher content similarity*. API traffic intensity refers to the arrival rate of requests containing specific API endpoints. Due to dependency relationships among APIs, requests with nested API endpoint relationships typically co-occur. Consequently, APIs exhibiting stronger traffic intensity correlations are more likely to possess nested relationships. Additionally, after removing the schema fields, requests containing the same transaction fields exhibit stronger *content similarity*. Figure 6 illustrates the principle of DeepTrace’s transaction analysis.

Nested API endpoint relationship inference based on API traffic intensity correlation. Within each component, DeepTrace sorts spans by their start times and groups them into quantity intervals. Given potential variations in API dependencies across microservices, DeepTrace *dynamically* infers API dependency relationships based on spans from the preceding m groups. For each API, DeepTrace calculates the arrival rate of its corresponding spans within every group among the last m intervals, thereby constructing a traffic intensity vector X , where X_i ($1 \leq i \leq m$) denotes the traffic intensity of the API within the i -th group. For any pair of incoming and outgoing spans, DeepTrace extracts the traffic intensity vectors X and Y corresponding to their APIs. DeepTrace employs a probability function, $P_H(\cdot)$, based on the Pearson correlation coefficient [19] to calculate the probability that these spans belong to the same nested API relationship:

$$P_H(\cdot) = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^n (X_i - \bar{X})^2} \sqrt{\sum_{i=1}^n (Y_i - \bar{Y})^2}} \quad (1)$$

A higher value calculated by $P_H(\cdot)$ indicates a greater likelihood that two spans share the same API endpoint relationship. This prevents the incorrect association of spans that do not possess an inherent API dependency.

Persistent transaction field inference based on the correlation of field weight vectors. DeepTrace employs the Term Frequency-Inverse Document Frequency (TF-IDF) [65] algorithm to calculate the probability that two spans contain the same persistent transaction fields. The advantage of this algorithm is that it assigns different weights to fields in the request contents based on their frequency of occurrence in the requests, thereby filtering out schema fields. This algorithm calculates the weight of a field as the product of its frequency (TF) and its inverse document frequency (IDF). To partition request content into fields, we need to consider three fundamental content formats: *text*, *binary*, and *mixed-format*. While text-based protocols like HTTP1 can delineate fields using fixed delimiters such as empty lines, binary protocols and mixed-format protocols must rely on their respective specifications to parse the entire payload, necessitating specific parsing mechanisms. Given the implementation complexity and protocol compatibility requirements, DeepTrace divides each span's request and response content into equal-sized fields using a size- s sliding window for a given interval containing M spans. It is important to note that s must be smaller than the length of the original persistent transaction field to ensure that the same transaction field in different requests is divided into identical field sets. For example, when $s = 7$, a persistent transaction field such as `username1` will always be split into `username`, `sername`, and `ername1` across all requests. Considering that transaction fields typically consist of 32-bit or 64-bit user IDs, order IDs, or user names with minimum length constraints, DeepTrace defaults the window size to 4 bytes.

For each unique field, DeepTrace counts the number of spans that include this field as m and subsequently computes the IDF for the field as $\log\left(\frac{M}{m+1}\right)$. For schema fields, due to their frequent occurrence across most spans, they have a lower IDF , resulting in low weights. Within a single span, DeepTrace calculates the TF by dividing the number of occurrences of each field within that span by the total number of fields the span contains. For any two spans, we obtain two field weight vectors by computing the $TF \times IDF$ for each field. Finally, DeepTrace calculates the cosine similarity [64] between two vectors to represent the probability that they contain the same transaction fields, with the corresponding probability function denoted as $P_D(\cdot)$.

Parent-child probability assessment. After obtaining the two transaction probability functions, it remains insufficient to evaluate all potential parent-child relationships among spans, as a single transaction may contain spans from multiple distinct traces or lack relevant transactional information in parent-child span pairs. To address these scenarios, we incorporate multiple causality-based parent-child span metrics to establish a more refined parent-child probability assessment function. DeepTrace incorporates multiple metrics for span pairs, including the delay between start times,

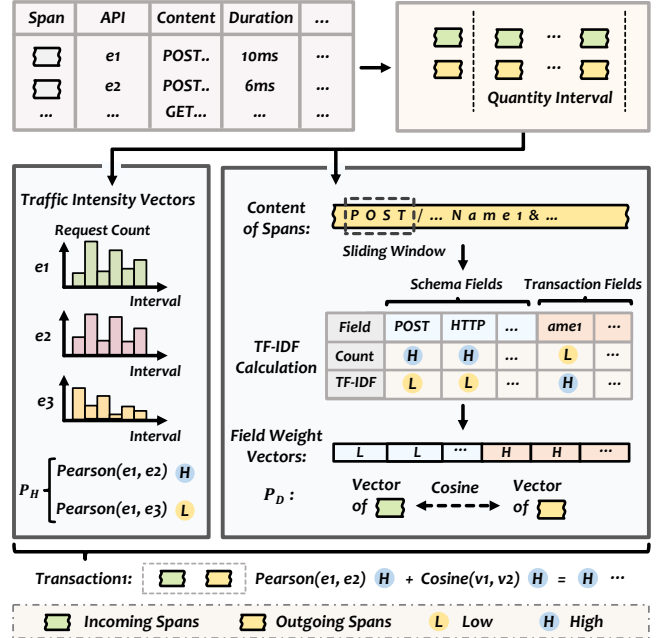


Figure 6: Transaction analysis of DeepTrace.

as well as their respective request sizes and durations. DeepTrace adopts a method similar to TraceWeaver [3] to obtain the initial parent-child span mappings and then uses a Gaussian kernel function [5] to construct a distribution $P_M^i(\cdot)$ for each metric between the incoming and outgoing spans for each API pair based on these initial mappings. For any incoming span s and outgoing span d , the probability of a parent-child relationship between them is given by:

$$P(s, d) = w_1 \times P_H(s, d) + w_2 \times P_D(s, d) + \sum_{i=1} w_{i+2} \times P_M^i(s, d) \quad (2)$$

w_i is a weight between 0 and 1. When transaction information is available, the function assigns a high probability to span mappings that belong to the same transaction and satisfy the parent-child span metric distribution. In cases where transaction information is unavailable, the parent-child relationship probability can still be assessed based on multi-dimensional metric distributions.

★ DESIGN 3. Iterative span association with automatic function weight adjustment. Within each quantity interval, DeepTrace dynamically constructs these probability functions and subsequently selects the set of span mappings with the highest probability. Algorithm 1 illustrates the iterative span association process. Since the effectiveness of each probability function varies, corresponding weights must be adjusted based on their effectiveness. For instance, the probability function $P_D(\cdot)$ becomes invalid when persistent transaction fields are absent, while the delay distribution loses its effectiveness when the delay between parent and child spans exhibits high randomness. However, due to the absence of ground truth for parent-child relationships, we cannot directly assess the validity of the probability functions.

Automatic function weight adjustment based on entropy. We observe that if the probability values calculated for each span mapping by the probability function are consistently close, it indicates

that the probability function is hard to distinguish between correct and error span mappings. Conversely, an effective probability function generates more distinct probability values for correct and incorrect span mappings. Therefore, DeepTrace evaluates the effectiveness of a probability function based on the Shannon entropy [49] of the set of probability values it calculates for all possible span mappings. Therefore, prior to span association, DeepTrace evaluates all potential parent-child span mappings formed by incoming and outgoing spans by computing their association probabilities using the probability functions (lines 3-7). DeepTrace then discretizes the probability values generated by each function and calculates their corresponding entropy values (line 10). Finally, the normalized entropy of each function serves as its respective weighting coefficient in the overall assessment (line 13).

Heuristic iterative span associations with linear complexity. To accurately perform span associations, certain constraints must be considered. Within each group, for a set of incoming spans $S = \{s_i \mid i \leq |S|\}$ and outgoing spans $D = \{d_j \mid j \leq |D|\}$, let $E = \{e_{ij}\}$ represent the inferred associations between spans, where $e_{ij} = 1$ indicates that s_i is associated with d_j , and $e_{ij} = 0$ indicates no association. We need to identify a span mapping set E that maximizes the sum of parent-child probabilities for the span mappings, expressed by the following formula:

$$\max \sum_{i=1}^{|S|} \sum_{j=1}^{|D|} P(s_i, d_j) \times e_{ij} \quad (3)$$

Simultaneously, mapping set E must satisfy two **conditions**: (1) for each span mapping, the outgoing span starts after and ends before its corresponding incoming span; and (2) every outgoing span must be associated exclusively and necessarily with a single incoming span. The **challenge** stems from the fact that the above multidimensional assignment problem is NP-hard [57], making it impractical to attain an exact solution. While some solvers [30] are capable of computing approximate solutions, the processing time increases exponentially with the number of spans, rendering them inefficient for real-time processing of large volumes of spans. Additionally, the initial metric distributions often lack accuracy, and solving for all mappings directly can lead to lots of association errors.

DeepTrace employs a heuristic-based efficient search method for span correlation. Initially, DeepTrace identifies all candidate mappings satisfying condition (1) based on temporal span information (line 15), then transforms this optimization problem into *selecting the optimal parent incoming span for each outgoing span*. Specifically, for a given outgoing span, DeepTrace identifies all candidate mappings M_{sub} containing that outgoing span (line 18), then selects the mapping with the highest parent-child probability using Equation 2 (line 19) while discarding the remaining mappings in M_{sub} due to condition (2) (line 21). Simultaneously, after every k span mappings are selected, DeepTrace updates the multidimensional metric distributions by incorporating the metrics from these newly identified mappings (line 23). It should be noted that dividing spans into distinct quantity intervals may result in parent and child spans being allocated to different intervals. To address this, when performing span correlation within each interval, DeepTrace incorporates incoming spans from the previous interval that

Algorithm 1 Iterative Span Association Algorithm

Input: Incoming spans S , outgoing spans D , probability function list P , step k
Output: Span mapping set M

```

1:  $E \leftarrow \{\}$  ▷ Corresponding entropy of the function
2:  $W \leftarrow \{\}$  ▷ Function weighting coefficients
3: for each function  $p \in P$  do ▷ Calculate entropy per function
4:    $\text{Scores} \leftarrow []$ 
5:   for each incoming span  $s \in S$  do
6:     for each outgoing span  $d \in D$  do
7:        $\text{Scores.append}(p((s, d)))$ 
8:   end for
9:   end for
10:   $E[p] \leftarrow \text{Entropy}(\text{Scores})$ 
11: end for
12: for each function  $p \in P$  do ▷ Normalize weights
13:   $W[p] \leftarrow (E[p] - \min(E)) / (\max(E) - \min(E))$ 
14: end for
15:  $M_{\text{cand}} \leftarrow \text{GenerateCandidates}(S, D)$ 
16:  $\text{count} \leftarrow 0$ 
17: for each outgoing span  $d \in D$  do
18:   $M_{\text{sub}} \leftarrow \{(s, d) \mid (s, d) \in M_{\text{cand}}\}$ 
19:   $\langle s^*, d \rangle \leftarrow \arg \max_{(s, d) \in M_{\text{sub}}} \sum_{p \in P} W[p] \cdot p((s, d))$ 
20:   $M \leftarrow M \cup \{(s^*, d)\}$ 
21:   $M_{\text{cand}} \leftarrow M_{\text{cand}} \setminus M_{\text{sub}}$  ▷ Prune candidate set
22:   $\text{count} \leftarrow \text{count} + 1$ 
23:  if  $\text{count} \equiv 0 \pmod{k}$  then
24:     $\text{UpdateDistribution}(P, M)$  ▷ Probability function iterative update
25:  end if
26: end for
27: return  $M$ 
```

can form valid candidate mappings with outgoing spans in the current interval according to condition (1).

3.3 Query-Driven Trace Assembling

The final goal of DeepTrace is to accurately assemble traces that *meet user query requirements with low transmission overhead* (**Goal 4**). Trace assembly typically requires centralized span collection due to the distribution of a single trace's spans across multiple agents. However, given the enormous requests processed by microservices daily [13, 14], centrally collecting all spans incurs significant transmission overhead [13, 14]. Although various trace sampling strategies have been proposed, they often fail to *strike a balance between accurately responding to queries and maintaining low transmission overhead*. For example, Dapper [75] and Jaeger [37] employ header-based uniform sampling, which overlooks a substantial number of traces that satisfy query requirements. Tail-based sampling [38, 47], while capable of filtering spans based on queries, still necessitates centralized collection of all spans by the server. Retrospective-based sampling detects edge cases at the agent and subsequently retrieves related traces [84], but it is limited to edge cases and lacks adaptability to user queries. Alternative approaches [10, 45] temporarily store monitoring data on agents for selective collection by the server, but this temporary storage of substantial raw spans could impose significant memory overhead on agents.

★ **DESIGN 4. Query-driven trace assembly with tag- and metric-based span index.** To align with user query requirements and ensure low transmission overhead, DeepTrace temporarily stores spans in agent memory while enabling the server to iteratively collect relevant trace spans based on user queries. Observing

that numerous spans share identical tags (e.g., host names, service names) and exhibit similar metrics (duration, memory usage), caching raw spans at agents proves inefficient. To decrease agent memory pressure, DeepTrace implements a *dual-indexing strategy* as illustrated in Figure 8. Specifically, DeepTrace constructs an inverted index for each tag type to efficiently reference all spans containing that specific tag. Additionally, DeepTrace maintains a histogram for each metric where each bucket groups spans with similar metric values. The stored span information is minimized to essential attributes, including parent-child relationships and timestamps, thereby avoiding duplication of identical tags and metrics across spans.

- (a) `QueryTag(Tags=["TagName":'value', ...])`
- (b) `QueryMetric(Metrics=["MetricName": Threshold, ...])`

Figure 7: Query API supported by DeepTrace.

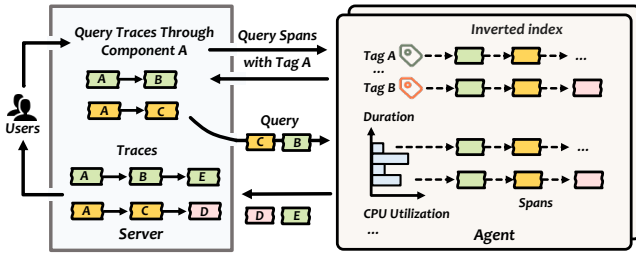


Figure 8: The procedure for trace assembly.

Operators can set queries through the flexible API shown in Figure 7. They can query traces that include spans with specific tags (a) or traces that include spans where specific metrics exceed a threshold (b). The agent filters span sets by query-specified tag indexes and spans within designated histogram buckets according to user queries.

On-demand span collection for trace assembling. DeepTrace takes user query requests as input and completes trace assembly by iteratively collecting spans from agents. In Figure 8, when a query for traces involving Component A is received, the DeepTrace server first forwards the query to all agents to collect span mappings that include Component A. During the subsequent assembly process, the server extracts new spans from each received span mapping and continues to query the agents for their parent or child spans. When no new spans are found, the trace assembly is complete, and the results are returned to the operator.

4 DeepTrace in Production

DeepTrace has been successfully deployed in large-scale production microservices across dozens of enterprises, including applications in e-commerce, cloud platforms, online gaming, telecommunications, and web services. The traces generated by DeepTrace are utilized by operators for various tasks such as failure diagnosis, resource planning, and testing, significantly improving diagnostic and debugging efficiency. We present two real-world cases from these enterprises here.

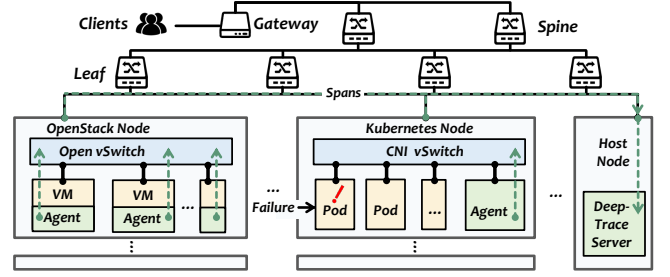


Figure 9: Production deployment of DeepTrace.

Use Case 1: Tail latency diagnosis. As illustrated in Figure 9, a financial institution’s online application utilizes the OpenStack [58] virtualization platform to host core business systems, while Kubernetes [44] orchestrates critical microservice components such as payment gateways and user authentication. These services interact intensively via HTTP, gRPC, and diverse database protocols. The system occasionally exhibits client request timeout anomalies. Despite leveraging Prometheus [63] for fault diagnosis, the tool’s coarse-grained metrics fail to isolate individual request-level granularity, impeding precise identification of latency anomalies across end-to-end request paths. This limitation prolongs troubleshooting efforts for **several days** without resolution.

To address this, the company deploys a DeepTrace server on a dedicated host and installs a DeepTrace agent across each Kubernetes node and OpenStack virtual machine without requiring one line of code modifications. DeepTrace agents autonomously collect request data to construct spans, while the server gathers and correlates spans, as well as reconstructs end-to-end request traces. By querying timeout-associated traces and analyzing latency distribution across components, operators rapidly identify the root-cause component within **15 minutes**. Subsequent analysis of this component reveals that insufficient allocation of connection resources underlies the observed failure.

Use Case 2: Application DDoS analysis. The business system of a smart device manufacturer is deployed across multiple availability zones in a public cloud environment and comprises complex components such as APISIX gateways, cloud load balancers, and Kubernetes clusters. It experiences an asymmetric application-layer DDoS attack targeting order service APIs. Attackers persistently submit order retrieval requests containing deeply nested query parameters at a low-frequency rate of 150 RPS. This induces resource exhaustion in Redis components, causing widespread failure of legitimate user requests requiring Redis access. Under conventional monitoring tools, the attack’s limited traffic scale remains below detection thresholds of traffic-based alerting systems, while performance degradation alerts from affected components are obscured by high noise levels. Given the system’s architectural complexity, operators struggle to trace request paths across the distributed infrastructure, delaying attack identification for hours.

As illustrated in Figure 10, following DeepTrace deployment, the system continuously captures latency metrics across all components in real time. Through trace analysis of timed-out requests, operators identified a significant degradation in Redis query response times, escalating from an average of 15 ms to 210 ms, along with clustered

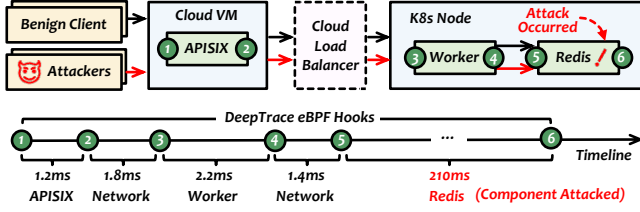


Figure 10: Using DeepTrace to locate the attacked component.

occurrences of slow queries exhibiting distinctive patterns. Security teams pinpoint the targeted Redis component within minutes and consequently neutralize the attack by dynamically implementing firewall rules to block malicious query patterns.

5 Testbed Evaluation

In this section, we examine the performance of DeepTrace with testbed evaluation. Our main results are as follows:

- DeepTrace maintains an end-to-end tracing accuracy of over 90% under scenarios of high concurrency, latency fluctuations, and caching (§ 5.1).
- DeepTrace’s novel designs enhance the tracing accuracy by approximately 15% (§ 5.2).
- DeepTrace achieves a trace reconstruction rate that is ten times faster than TraceWeaver (§ 5.3).
- DeepTrace reduces transmission overhead of trace assembling by 94% (§ 5.4).
- DeepTrace results in only a 5% degradation in application end-to-end performance (§ 5.5).

Testbed setup. Our testbed comprises a three-node Kubernetes cluster [44] (version 1.24.17), with each node equipped with identical Intel Xeon E5-2620 v3 CPUs (2.40 GHz, 12 physical cores) and 128 GB of RAM. The servers operate on Ubuntu 22.04 with kernel version 5.15.0.

Workload setup. We evaluate the social networking and movie reviewing applications from DeathStarBench [27], which consist of 36 and 38 components, respectively. These applications integrate diverse concurrency mechanisms, including thread pools from the Thrift library, asynchronous I/O in the C++ standard library, and hybrid communication protocols such as Redis, Thrift, and HTTP. The wrk2 [80] tool is employed as a load generator to regulate request arrival intervals using an exponential distribution at each request-per-second (RPS) level, with sustained 10-second test durations and randomized unique fields (e.g., user IDs and post content) embedded in request payloads. Each experiment mitigates systematic errors by averaging results from multiple test iterations across both applications. To quantify the tracing accuracy of non-intrusive tracing, the intrusive tracing framework Jaeger [37] serves as a ground-truth reference by injecting unique contextual identifiers (e.g., trace IDs) into parent-child request payloads. This allows us to establish the ground truth of request parent-child relationships based on the Jaeger context extracted from the captured request payloads. When evaluating different non-intrusive tracing algorithms, we first remove the Jaeger context from request payloads to ensure unbiased comparison, then assess their tracing accuracy by comparing the tracing results against the established ground truth.

Baselines. For the *accuracy of non-intrusive end-to-end tracing*, we compare DeepTrace with *FIFO-based* DeepFlow [73] and vPath [70], as well as *delay-based* TraceWeaver [3] and WAP5 [67]. DeepFlow matches each outgoing span with the nearest incoming span in the same thread. vPath assumes that all outgoing spans triggered by a single incoming span are processed by the same thread, T1, and correlates them with the incoming span whose response is sent by T1. WAP5 assigns each outgoing span to the nearest unassigned incoming span to generate a delay distribution. TraceWeaver models the delay distribution using a Gaussian distribution and incorporates the call graph to establish constraints. Additionally, we also compare the *performance overhead* of DeepTrace with common frameworks such as Jaeger and DeepFlow.

5.1 End-to-end Tracing Accuracy

Tracing accuracy under different loads. We first evaluate the impact of varying loads, measured in requests per second (RPS), on end-to-end tracing accuracy. Figure 11 shows that as the load increases, higher parallelism leads to more frequent request re-ordering, significantly reducing the accuracy of baseline methods. FIFO-based vPath and DeepFlow achieve only about 40% accuracy at 1500 RPS, as they fail to account for operations such as cross-thread interactions and thread reuse. Under high concurrency, the density of spans on the timeline significantly increases. A large number of incorrect span mappings obtain high probabilities in the delay distribution, leading to erroneous associations in delay-based solutions. For example, TraceWeaver achieves only 63% accuracy at 1500 RPS. In contrast, DeepTrace *maintains an accuracy rate of over 95% across different RPS levels*. It fully leverages various transaction-related information contained in the request content, such as post information and user IDs, to significantly reduce the number of erroneous candidate mappings. DeepTrace’s errors primarily occur in a small subset of requests lacking transactional fields.

Tracing accuracy across requests binned by end-to-end latency. Analyzing traces corresponding to tail latency requests is crucial for failure diagnosis. We measure the accuracy of traces grouped by end-to-end latency percentiles at 500 RPS. As shown in Figure 12, *incorrect traces of all baselines are concentrated in the tail requests*. This is because tail requests exhibit higher execution duration variability, resulting in more severe request disordering and causing delay between parent-child requests to depart from normal distributions, consequently leading to errors in both FIFO-based and delay-based approaches. In contrast, DeepTrace *relies on transaction information and other metrics to avoid the failure of delay distribution, maintaining 92% accuracy in the tail 10%ile of requests*.

Tracing accuracy under delay fluctuations. In real-world environments, delay patterns between parent-child requests may be influenced by external factors such as resource contention, failures, and workload variations. To simulate such conditions, we select the composepost component in the social networking application and inject randomized additional delays (mean: 50 ms, variance: 20 ms) into 0-50% of incoming requests at 500 RPS. Figure 13 reveals that TraceWeaver achieves only 76% accuracy with 50% delayed requests. This performance degradation occurs because delay fluctuations disrupt the normal delay patterns between

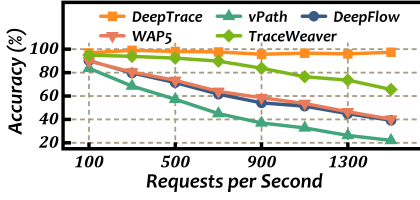


Figure 11: End-to-end tracing accuracy under different loads.

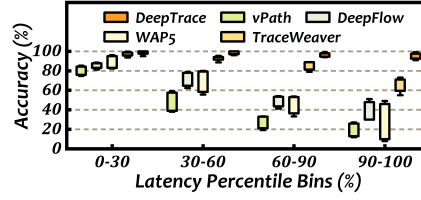


Figure 12: End-to-end tracing accuracy binned by end-to-end latency.

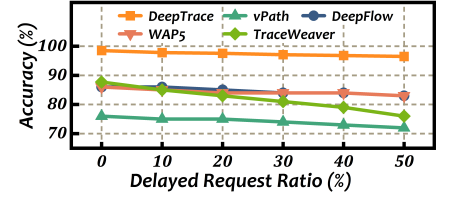


Figure 13: End-to-end tracing accuracy under delay fluctuations.

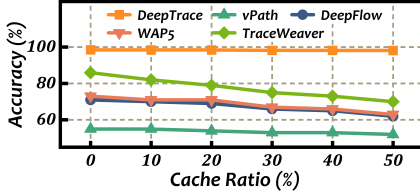


Figure 14: End-to-end tracing accuracy under dynamic execution paths.

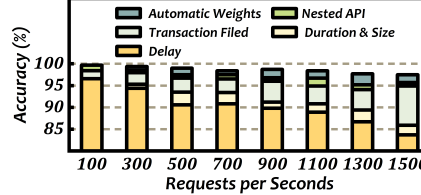


Figure 15: The impact of different modules of DeepTrace on the accuracy.

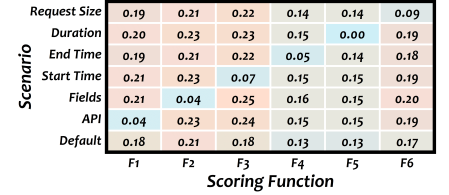


Figure 16: Weight distribution of the scoring functions in different scenarios.

parent-child requests, rendering delay-dependent methods ineffective. In contrast, DeepTrace maintains 95% tracing accuracy even under 50% delayed requests by automatically adjusting weights to disregard delay-based scoring functions and instead leveraging request content correlation unaffected by external factors for request association.

Tracing accuracy under dynamic execution path. This experiment evaluates tracing accuracy under dynamic execution path conditions where behaviors like caching and error handling may alter request processing paths. To simulate dynamic execution paths, we artificially introduce caching mechanisms in the social networking application's composepost component, randomly discarding outgoing spans (and their descendant spans) triggered by this component, with cache ratios ranging from 0% to 50%. As shown in Figure 14, while TraceWeaver accounts for dynamic execution paths, its accuracy remains severely constrained (merely 70%) due to assumptions about call graphs of outgoing requests. In contrast, DeepTrace independently selects the most probable incoming span as the parent for each outgoing span. Consequently, untriggered outgoing spans resulting from cache misses do not affect parent inference for triggered outgoing spans, thereby maintaining accuracy of 97%.

5.2 DeepTrace Analysis

DeepTrace ablation study. To quantify the benefits of different modules for span association, we conduct an ablation study on DeepTrace by sequentially removing automatic weight adjustment, nested API information, transaction fields, and metrics (duration and request size). When eliminating automatic weight adjustment, uniform weights are applied to all scoring functions. Figure 15 demonstrates that removing these four modules reduced DeepTrace's accuracy at 1500 RPS by 2%, 1%, 9%, and 2%, respectively. This underscores that DeepTrace's content analysis contributes most significantly to tracing accuracy, while the auxiliary mechanisms of automatic weight adjustment and multi-dimensional metrics further enhance system stability.

Effectiveness of adjustable function weights. This experiment evaluates automatic weight adjustment effectiveness by creating

distinct scoring function impairment scenarios. For each scenario, we systematically invalidate corresponding scoring functions F_i by randomizing (1) APIs, (2) request content fields, (3) start-time intervals, (4) end-time intervals, (5) request durations, and (6) request sizes of parent-child spans. As Figure 16 demonstrates, transactional fields provide the most reliable request association under default conditions, resulting in scoring function F_2 receiving the highest weight allocation. During each impairment scenario, DeepTrace reduces weights of compromised scoring functions, thereby eliminating their detrimental effects on request association.

5.3 Trace Reconstruction Efficiency

The efficiency of trace reconstruction directly impacts failure diagnosis efficiency. Thus, we first compare the time required by different algorithms to construct varying numbers of traces. Figure 17 shows that DeepTrace *has an association speed comparable to the FIFO-based scheme and is over 10 times faster than the delay-based TraceWeaver*. The FIFO-based schemes, which do not involve complex delay distribution estimation, have the fastest association speed. In contrast, TraceWeaver, using a solver to compute the span set with the highest delay probability, is the slowest, taking about 440 seconds to process 10,000 requests. Due to DeepTrace's greedy search strategy, it can quickly infer the span set with a higher probability in the metric distribution, processing 10,000 requests in about 40 seconds.

5.4 Transmission Overhead of Assembling

To verify the advantages of *query-driven trace assembling* of DeepTrace, we test the transmission overhead incurred by different tracing frameworks when responding to *user queries for traces of requests with greater than 95th percentile latency*. We deploy Jaeger with sampling rates of 1% and 100% (denoted as Jaeger, 1% and Jaeger, 100%), as well as DeepFlow and DeepTrace without sampling. We then configure the workload generator to produce varying numbers of user requests. Figure 18 shows that DeepTrace's overhead for obtaining target traces ranges from 30MB to 60MB. Although DeepTrace requires multiple interactions between the

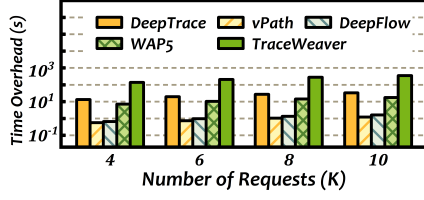


Figure 17: Efficiency of trace reconstruction in different algorithms.

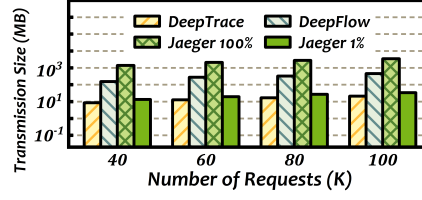


Figure 18: Transmission overhead generated by different frameworks.

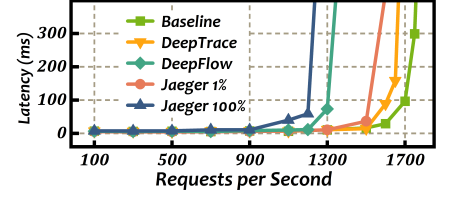


Figure 19: The end-to-end performance evaluation of DeepTrace.

server and agent, the server only needs to provide the agent with the IDs of the spans to be queried, which incurs negligible overhead. In contrast, tracing frameworks without sampling need to centrally collect and store all traces, resulting in overheads as high as 1000 MB. While Jaeger 1% has overhead similar to DeepTrace, its random sampling strategy leads to the loss of many traces for requests with tail latency.

5.5 End-to-end Performance

Finally, we evaluate the end-to-end impact of DeepTrace on microservice performance. To establish the baseline, we manually remove all tracing-related codes from the social networking application. Then, we separately run each distributed tracing framework and plot the results in Figure 19. The baseline achieves a maximum throughput of approximately 1,749 RPS. When deploying Jaeger at 100% sampling, the throughput decreases by approximately 24%, reaching 1,326 RPS. Lowering the sampling rate to 1% significantly reduces Jaeger's overhead, increasing throughput to about 1,630 RPS. DeepFlow, on the other hand, results in a throughput of approximately 1,300 RPS, representing a similar decline to Jaeger 100%. By contrast, DeepTrace reduces the baseline throughput by about 4.6%, achieving 1,667 RPS.

6 Experience & Lessons

We share some experiences in deploying non-intrusive tracing in enterprise microservices and discuss potential directions for future improvements.

Global Inference. Existing non-intrusive methods independently correlate incoming and outgoing spans within each component and then connect these spans to form traces. However, this approach suffers from declining accuracy as the number of components increases. For example, if each component achieves 99% correlation accuracy, the end-to-end tracing accuracy for a trace spanning 50 components drops to just 61% ($0.99^{50} = 0.605$). In future work, we plan to adopt a global inference approach. For example, we will combine the correlation results from components that a span has passed through to infer its child spans, thereby alleviating the accuracy degradation in large-scale microservices.

Valuable thread information. We find that the characteristics of the threading models in some microservice components can aid in span association. For example, in a dispatcher-worker cross-thread model, the thread T1 that handles the incoming span delegates all child spans triggered by this incoming span to another thread T2 for processing. We can leverage these patterns to constrain the candidate child combinations associated with an incoming span.

Additionally, in some thread interactions, system calls related to section locks are triggered, and we can further utilize this information to reduce the number of candidate mappings.

Load fluctuations. Applications exhibit peaks and troughs in request concurrency. Under low load, there are fewer candidate mappings, making it easier to obtain accurate traces. We can leverage the mappings inferred during trough periods to construct more accurate joint metric distributions.

7 Related Work

Distributed tracing. Intrusive distributed tracing involves modifying code to insert global identifiers into requests to record traces. Dapper [75] establishes the foundation for intrusive tracing. Pivot Tracing [53] introduces a novel "happened-before join" operator for instrumentation. OpenTelemetry [59] standardizes instrumentation libraries and provides SDKs for multiple programming languages. Non-intrusive tracing methods include automatic code insertion [4, 32, 48, 79], such as adding parameters to Golang functions [79] or inserting callbacks in JavaScript [48], but these methods are limited to specific languages. More general non-intrusive schemes infer causality by analyzing system logs and request metadata, like DeepFlow and TraceWeaver.

Causality inference. Existing work advances non-intrusive inference of causal relationships between requests through two primary strategies. The first involves analyzing system logs to infer request execution flows within components [9, 11, 42, 54, 55, 70, 73, 78, 81]. For instance, vPath [78] examines thread and network activity logs to identify request processing trajectories, PreciseTracer [70] transforms network communication events into causal paths and organizes them into structural schemas, and DeepFlow [73] correlates temporally adjacent network sends and receives within the same thread to trace interactions. The second strategy focuses on analyzing the distribution of delays between incoming and outgoing requests [2, 3, 6, 16, 67]. WAP5 [67] predicts service dependencies by assuming an exponential delay distribution, while TraceWeaver [3] employs a Gaussian distribution and iterative inference methodologies to improve the precision of delay estimations.

Trace sampling. Various strategies apply different sampling methods to reduce the number of traces stored [17, 31, 33, 46, 47, 83]. For example, Las-Casas et al. [46] propose a hierarchical clustering-based trace sampling approach. Sifter [47] constructs an unbiased, low-dimensional model to capture traces that stray from normal system behavior. Sieve [33] uses attention mechanisms to increase the sampling rate of uncommon traces. However, these sampling methods ignore the querying requirements of operators.

8 Conclusion

In this paper, we introduce DeepTrace, a transaction-based, non-intrusive distributed tracing framework designed for microservices. DeepTrace infers the transaction type of requests by utilizing the API endpoint and transaction fields from the request's message content. It then employs transaction types and request metrics to deduce the end-to-end execution paths of requests. Evaluations show that DeepTrace achieves over 95% accuracy in end-to-end tracing across various scenarios and has been successfully employed in multiple production environments.

Acknowledgments

We are grateful to our shepherd, Aurojit Panda, and the anonymous SIGCOMM reviewers for their insightful feedback that helped improve our work. The research is supported by the National Natural Science Foundation of China under Grant No. 62394322.

References

- [1] 2025. Apache thrift. <https://thrift.apache.org/>
- [2] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 74–89.
- [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*. Association for Computing Machinery, New York, NY, USA, 828–842.
- [4] Emre Ates, Lily Sturmman, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K Coskun, and Raja R Sambasivan. 2019. An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In *Proceedings of the ACM Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, USA, 165–170.
- [5] Jean Babaud, Andrew P Witkin, Michel Baudin, and Richard O Duda. 1986. Uniqueness of the Gaussian kernel for scale-space filtering. *IEEE transactions on pattern analysis and machine intelligence PAMI-8*, 1 (1986), 26–33.
- [6] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A Maltz, and Ming Zhang. 2007. Towards highly reliable enterprise network services via inference of multi-level dependencies. *ACM SIGCOMM Computer Communication Review* 37, 4 (2007), 13–24.
- [7] Peter Bailis, Peter Alvaro, and Sumit Gulwani. 2017. Research for practice: Tracing and debugging distributed systems; programming by examples. *Commun. ACM* 60, 7 (2017), 46–49.
- [8] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for request extraction and workload modelling. In *OSDI*, Vol. 4. 18–18.
- [9] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: Online Modelling and Performance-Aware Systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9* (Lihue, Hawaii) (HOTOS'03). USENIX Association, USA, 15.
- [10] Jessica Berg, Fabian Ruffy, Khanh Nguyen, Nicholas Yang, Taeyun Kim, Anirudh Sivaraman, Ravi Netravali, and Srinivas Narayana. 2021. Snicket: Query-driven distributed tracing. In *Proceedings of the 20th ACM Workshop on Hot Topics in Networks*. 206–212.
- [11] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering*. Association for Computing Machinery, New York, NY, USA, 468–479.
- [12] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D Ernst. 2016. Debugging distributed systems: Challenges and options for validation and debugging. *Queue* 14, 2 (2016), 91–110.
- [13] The Netflix Technology Blog. 2025. Making the Netflix API More Resilient. "https://netflixtechblog.com/making-the-netflix-api-more-resilient-a8ec62159c2d".
- [14] BMW. 2025. BMW Group at AWS re:Invent 2023. "https://aws.amazon.com/solutions/case-studies/bmw-keynote-aws-reinvent-2023/".
- [15] Phil Calçado. 2022. Building Products at SoundCloud – Part I: Dealing with the Monolith. <https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith>
- [16] Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. 2008. Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions. In *OSDI*, Vol. 8. USENIX Association, USA, 117–130.
- [17] Zhuangbin Chen, Zhihan Jiang, Yuxin Su, Michael R Lyu, and Zibin Zheng. 2024. TraceMesh: Scalable and Streaming Sampling for Distributed Traces. *arXiv preprint arXiv:2406.06975* (2024), 54–65.
- [18] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 2014. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, USA, 217–231.
- [19] Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. *Noise reduction in speech processing* (2009), 1–4.
- [20] Oracle Corporation. 2022. MySQL. <https://www.mysql.com>
- [21] Chinthaka Dinadasa. 2025. Microservices - User Service Implementation. <https://www.javatodev.com/microservices-user-service-implementation/>
- [22] Inc. F5. 2022. NGINX: Advanced Load Balancer, Web Server, & Reverse Proxy. <https://www.nginx.com>
- [23] 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 USENIX Conference on Networked Systems Design and Implementation (Cambridge, MA) (NSDI'07)*. USENIX Association, USA, 20.
- [24] Apache Software Foundation. 2023. KAFKA PROTOCOL GUIDE. <https://kafka.apache.org/protocol.html>
- [25] Apache Software Foundation. 2023. Remote communication details of Dubbo. <https://dubbo.apache.org/en/docs/v2.7/dev/implementation/#remote-communication-details>
- [26] Frontegg. 2025. Authentication in Microservices: Approaches and Techniques. <https://frontegg.com/blog/authentication-in-microservices>
- [27] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 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
- [28] Yoni Goldberg. 2022. Scaling Gilt: from Monolithic Ruby Application to Distributed Scala Micro-Services Architecture. <https://www.infoq.com/presentations/scale-gilt/>
- [29] Google. 2024. gRPC. <https://grpc.io/>
- [30] LLC Gurobi Optimization. 2025. Gurobi Optimizer. <https://github.com/Gurobi>
- [31] 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*. Association for Computing Machinery, New York, NY, USA, 1750–1761.
- [32] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and Enhancing In Situ System Observability for Failure Detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 1–16. <https://www.usenix.org/conference/osdi18/presentation/huang>
- [33] 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)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 436–446.
- [34] International Business Machines Corporation (IBM) and Eurotech. 2023. MQTT V3.1 Protocol Specification. <https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html>
- [35] Internet Engineering Task Force (IETF). 2023. RFC 7231. <https://www.rfc-editor.org/rfc/rfc7231>
- [36] Internet Engineering Task Force (IETF). 2023. RFC 7540. <https://www.rfc-editor.org/rfc/rfc7540>
- [37] Jaeger. 2023. Jaeger: open source, end-to-end distributed tracing. <https://www.jaegertracing.io/>
- [38] 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* (Shanghai, China) (SOSP '17). Association for Computing Machinery, New York, NY, USA, 34–50. doi: 10.1145/3132747.3132749
- [39] Linux Kernel. 2022. eBPF header file at Kernel.org git repositories. Retrieved Feb, 2023 from <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/linux/bpf.h>
- [40] The Linux Kernel. 2023. bpf(2) – Linux manual page. <https://man7.org/linux/man-pages/man2/bpf.2.html>
- [41] The Linux Kernel. 2025. Uprobe-tracer: Uprobe-based Event Tracing. <https://www.kernel.org/doc/html/v4.19/trace/uprobetracer.html>
- [42] Eric Koskinen and John Jannotti. 2008. Borderpatrol: isolating events for black-box tracing. *ACM SIGOPS Operating Systems Review* 42, 4 (2008), 191–203.

- [43] Paul Kranenburg, Branko Lankester, Rick Sladkey, et al. 2025. linux syscall tracer. <https://strace.io/>
- [44] Kubernetes. 2024. Production-Grade Container Scheduling and Management. "https://kubernetes.io/".
- [45] PIXIE labs. 2022. Instantly troubleshoot your applications on Kubernetes. Retrieved Feb, 2023 from <https://pixielabs.ai/>
- [46] 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
- [47] Pedro 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* (Santa Cruz, CA, USA) (SoCC '19). Association for Computing Machinery, New York, NY, USA, 312–324. doi: 10.1145/3357223.3362736
- [48] Ding Li, James Mickens, Suman Nath, and Lenin Ravindranath. 2015. Domino: Understanding wide-area, asynchronous event causality in web applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, USA, 182–188.
- [49] Jianhua Lin. 1991. Divergence measures based on the Shannon entropy. *IEEE Transactions on Information theory* 37, 1 (1991), 145–151.
- [50] Linux. 2023. socket(2) — Linux manual page. <https://man7.org/linux/man-pages/man2/socket.2.html>.
- [51] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Chengzhong Xu. 2022. An In-Depth Study of Microservice Call Graph and Runtime Performance. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3901–3914. doi: 10.1109/TPDS.2022.3174631
- [52] LWN. 2023. A thorough introduction to eBPF. <https://lwn.net/Articles/740157/>.
- [53] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) (SOSP '15). Association for Computing Machinery, New York, NY, USA, 378–393. doi: 10.1145/2815400.2815415
- [54] Francisco Neves, Nuno Machado, et al. 2018. Falcon: A practical log-based analysis tool for distributed systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 534–541.
- [55] Francisco Neves, Nuno Machado, Ricardo Vilça, and José Pereira. 2021. Horus: Non-intrusive causal analysis of distributed systems logs. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 212–223.
- [56] Nginx. 2022. Adopting Microservices at Netflix: Lessons for Architectural Design. <https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/>
- [57] Duc Manh Nguyen, Hoai An Le Thi, and Tao Pham Dinh. 2014. Solving the multidimensional assignment problem by a cross-entropy method. *Journal of Combinatorial Optimization* 27 (2014), 808–823.
- [58] openstack. 2025. OpenStack: Open Source Cloud Computing Infrastructure. <https://www.openstack.org/>
- [59] OpenTelemetry. 2023. High-quality, ubiquitous, and portable telemetry to enable effective observability. <https://opentelemetry.io/>
- [60] Oracle. 2023. MySQL Client/Server Protocol. https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_PROTOCOL.html
- [61] Jenish Patel. 2025. Authentication and Authorization in a microservices architecture. <https://www.thirdrocktechkno.com/blog/authentication-authorization-in-a-microservices-architecture/>
- [62] Piotr. 2025. Distributed Transactions in Microservices with Spring Boot. <https://piotrminkowski.com/2020/06/19/distributed-transactions-in-microservices-with-spring-boot/>
- [63] Prometheus. 2025. From metrics to insight. <https://prometheus.io/>
- [64] Faisal Rahutomo, Teruaki Kitasuka, Masayoshi Aritsugi, et al. 2012. Semantic cosine similarity. In *The 7th international student conference on advanced science and technology ICAST*, Vol. 4. University of Seoul South Korea, 1.
- [65] Juan Ramos et al. 2003. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, Vol. 242. Citeseer, 29–48.
- [66] Redis. 2023. Redis serialization protocol (RESP) specification. <https://redis.io/docs/reference/protocol-spec/>
- [67] 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*. Association for Computing Machinery, New York, NY, USA, 347–356.
- [68] Raja R Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R Ganger. 2016. Principled workflow-centric tracing of distributed systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, USA, 401–414.
- [69] Salvatore Sanfilippo. 2024. Redis. <https://redis.io/>
- [70] Bo Sang, Jianfeng Zhan, Gang Lu, Haining Wang, Dongyan Xu, Lei Wang, Zhihong Zhang, and Zhen Jia. 2011. Precise, scalable, and online request tracing for multiter services of black boxes. *IEEE Transactions on Parallel and Distributed Systems* 23, 6 (2011), 1159–1167.
- [71] Santiago. 2025. Efficient Transaction Management in Microservices with the Saga Pattern. <https://medium.com/@barbieri.santiago/efficient-transaction-management-in-microservices-with-the-saga-pattern-6bab70fd7f1d>
- [72] Amazon Web Services. 2025. Access token security for microservice APIs on Amazon EKS. <https://aws.amazon.com/cn/blogs/security/access-token-security-for-microservice-apis-on-amazon-eks/>
- [73] 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
- [74] Ben Sigelman. 2024. Towards Turnkey Distributed Tracing (June 2016). <https://medium.com/opentracing/towards-turnkey-distributed-tracing-5f4297d1736>
- [75] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaskan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. <https://research.google.com/archive/papers/dapper-2010-1.pdf>
- [76] Karan Parikh Steven Ihde. 2022. From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture. <https://www.infoq.com/presentations/linkedin-microservices-urn/>
- [77] suricata. 2025. A high performance, open source network analysis and threat detection software. <https://suricata.io/>
- [78] Byung-Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urganekar, and Rong N Chang. 2009. vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities.. In *USENIX Annual technical conference*. USENIX Association, San Diego, CA, 259–272.
- [79] Adam Welc. 2021. Automated code transformation for context propagation in Go. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 1242–1252.
- [80] wrk2. 2023. An HTTP benchmarking tool based mostly on wrk. <https://github.com/giltene/wrk2>
- [81] Hongteng Xu, Xia Ning, Hui Zhang, Junghwan Rhee, and Guofei Jiang. 2016. Pinfer: Learning to infer concurrent request paths from system kernel events. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 199–208.
- [82] zeek. 2025. An Open Source Network Security Monitoring Tool. <https://zeek.org/>
- [83] 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 *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*. Association for Computing Machinery, New York, NY, USA, 1221–1232.
- [84] 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>