

FlowScope: Non-Intrusive Distributed Tracing with Method-Level Delay Estimation for Microservices Troubleshooting

Yantao Geng
Tsinghua University
Beijing, China
gyt22@mails.tsinghua.edu.cn

Han Zhang*
Tsinghua University
Beijing, China
zhhan@tsinghua.edu.cn

Zhiheng Wu
Tsinghua University
Beijing, China
wu-zh24@mails.tsinghua.edu.cn

Yahui Li
Tsinghua University, China
Beijing, China
liyahui_ego@126.com

Jilong Wang
Tsinghua university
Beijing, China
wjl@cernet.edu.cn

Xia Yin
Tsinghua University
Beijing, China
yxia@tsinghua.edu.cn

Abstract

As the scale and complexity of microservices increase, their operation and debugging become increasingly challenging. A single user request may involve interactions among hundreds of components. In such complex systems, non-intrusive distributed tracing techniques, which require no modification of application code, significantly enhance the convenience of troubleshooting. However, existing non-intrusive tracing frameworks still exhibit critical shortcomings in achieving tracing accuracy: current approaches either rely on specific thread model assumptions, limiting their applicability, or employ incorrectly constructed causal delay distributions to infer request causality, leading to erroneous trace construction. To overcome these limitations, we propose FlowScope, a non-intrusive distributed tracing framework for microservices that optimizes trace reconstruction through method-level delay distribution estimation. This framework estimates the causal delay distribution between requests of different method types via cross-correlation analysis, then uses these delay distributions to accurately evaluate the probability of causal relationships between requests. Subsequently, it efficiently reconstructs traces by incorporating multiple constraints with an iterative algorithm. Our testbed results indicate that FlowScope improves tracing accuracy by 51% compared to state-of-the-art frameworks (e.g., DeepFlow and TraceWeaver) and has been successfully applied in multiple real-world use cases.

CCS Concepts

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

Keywords

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

*Han Zhang is the corresponding author.



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

ICSE '26, Rio de Janeiro, Brazil

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2025-3/26/04

<https://doi.org/10.1145/3744916.3764518>

ACM Reference Format:

Yantao Geng, Han Zhang, Zhiheng Wu, Yahui Li, Jilong Wang, and Xia Yin. 2026. FlowScope: Non-Intrusive Distributed Tracing with Method-Level Delay Estimation for Microservices Troubleshooting. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26)*, April 12–18, 2026, Rio de Janeiro, Brazil. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3744916.3764518>

1 Introduction

In recent years, user-interactive online services have gradually shifted from monolithic to microservices architectures [13, 25, 41, 57]. Microservices architecture breaks down the original single monolithic binary into multiple loosely coupled, single-purpose components [36, 56]. Because of decoupling, microservice systems have better scalability than monolithic programs. However, this also leads to modern microservices applications being highly distributed, posing significant challenges for troubleshooting [8, 11]. A single user request can involve hundreds or even thousands of components [34, 36, 66], making it difficult to pinpoint the specific components responsible for requests with problems.

In such complex microservices environments, distributed tracing frameworks like Dapper [55] and Canopy [30] have become crucial troubleshooting tools for microservices [29, 44, 52, 65]. These frameworks generate a **trace** for each user request to record its end-to-end execution flow across all components. Based on whether they *interfere with application running*, these frameworks can be categorized into **intrusive** and **non-intrusive**. Non-intrusive tracing frameworks, such as DeepFlow [52] and TraceWeaver [6], are achieved by intercepting external interaction interfaces of microservice components, such as system calls, thereby avoiding interference with application running. In contrast, intrusive frameworks like Jaeger [29] and Zipkin [68] implement request tracing by adding tracing logic to the application-level source code of components. Compared to intrusive tracing, non-intrusive tracing offers advantages such as convenient and rapid deployment in large-scale microservices and the ability to cover closed-source or third-party components [16, 20, 50] that cannot be modified. As a result, non-intrusive distributed tracing has recently attracted widespread attention [48, 51, 52].

However, achieving accurate, non-intrusive distributed tracing is challenging. A common format for a trace is a *span tree*, where *spans* serve as nodes and *parent-child relationships* as edges [44, 55]. 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 request spans (child) during its processing. Tracing frameworks must correctly associate spans with such parent-child relationships, yet this often depends on *invisible* application logic. When a component employs mechanisms such as non-blocking I/O or cross-thread concurrency to handle multiple incoming spans simultaneously, it is difficult to determine which incoming spans lead to which outgoing spans. In § 2.3, existing non-intrusive distributed tracing frameworks are classified according to the *span association mechanisms* into **FIFO-based** and **delay-based** approaches to explain their limitations.

FIFO-based frameworks. Existing FIFO-based frameworks, including DeepFlow [52] and vPath [59], operate on a *first-in, first-out* principle for request association. Their core assumption is that each incoming request and all its child requests are processed by the same thread, and no new requests are handled before the current incoming request is completed. These frameworks therefore infer parent-child relationships by matching incoming and outgoing requests in the order of their arrival within the same thread. However, this method heavily relies on a specific threading model and has limited applicability in real-world microservices environments. Modern microservices often employ coroutines, as seen in Python and Golang; cross-thread invocations, like those in Thrift and gRPC communication libraries; or event-driven asynchronous models, such as Node.js servers. These mechanisms break the FIFO order within threads, causing missequencing of requests and ultimately leading to incorrect associations.

Delay-based frameworks. Delay-based approaches [4, 48] estimate the delay distribution between receiving an incoming request and triggering a child outgoing request. This distribution is used to assess the probability of a parent-child relationship between requests, with those exhibiting high probability being associated. This approach is independent of threading models, offering broad applicability. However, conventional methods often rely on a single delay distribution for an entire component, ignoring variations across different types of parent-child method pairs. In practice, an incoming request of a specific method may, after variable delays, trigger outgoing requests calling different methods. Each such parent and child method forms a distinct *method pair*, and each method pair type has its own characteristic delay distribution. A unified distribution fails to capture these fine-grained differences, leading to distorted parent-child probability estimates and ultimately causing association errors.

FlowScope framework. To address these challenges, we propose FlowScope, a non-intrusive distributed tracing framework based on *method-level delay distribution estimation*, designed to achieve *efficient* and *high-accuracy* request trace reconstruction in microservices environments. FlowScope automatically constructs fine-grained method-level delay distributions by analyzing the cross-correlation between request sequences of different method pairs, thereby accurately assessing the probability of parent-child request relationships. Ultimately, it transforms the span association problem into a constrained optimization problem and identifies the optimal span association combination through a heuristic iterative algorithm. Our **novel designs** include:

Non-intrusive span construction (§ 3.1). Since microservices often comprise components written in multiple programming languages, non-intrusive tracing must support *multi-language applicability* to cover all components traversed by a request trace. Some existing non-intrusive solutions rely on language-specific instrumentation techniques [45, 53] to intercept requests and construct spans. However, these approaches are limited to specific languages, require customized development for each language, and pose challenges for unified maintenance.

In contrast, FlowScope innovatively operates at the kernel level by leveraging eBPF programs to hook entry and exit points of major network-related system calls, thereby intercepting network communication activities across all microservice components. It automatically collects data such as the five-tuple and payload of requests when they enter or exit the kernel. In user space, it correlates requests and responses to construct spans, achieving programming language-agnostic and non-intrusive span construction.

Method-level delay distribution estimation (§ 3.2). Accurately assessing the probability of parent-child relationships between incoming and outgoing spans relies on high-precision delay distributions. The primary limitation of existing delay-based approaches lies in constructing a single delay distribution for an entire component, which overlooks significant delay variations across different method pairs due to differing business logic.

FlowScope proposes a method-level delay distribution estimation mechanism based on *cross-correlation analysis*. We serialize requests of different methods and compute the cross-correlation between request sequences of any two methods at various time offsets. This process automatically identifies the high-probability peak delay that best represents their causal relationships, thereby constructing dedicated Gaussian delay distribution models for each method pair. This approach requires no prior knowledge and demonstrates strong robustness in dynamic invocation environments.

Iterative span association (§ 3.3). Finally, FlowScope estimates the probability of parent-child relationships between incoming and outgoing spans using method-level delay distributions. To identify the optimal set of mappings that maximizes the overall parent-child probability, constraints such as method invocation relationships and the requirement that each outgoing span has only one parent span must be considered. FlowScope formulates span association as a global optimization problem. This problem is theoretically NP-hard, leading to exponential growth in computation time as the number of spans increases, which fails to meet the real-time requirements of large-scale microservice clusters. To address this, FlowScope proposes a *top-k iterative heuristic algorithm* with a constraint conflict resolution mechanism. Instead of seeking a global optimal solution in a single step, this algorithm iteratively selects the top-k mappings with the highest parent-child probabilities from the pool of unprocessed candidate mappings. This approach significantly improves the efficiency of trace reconstruction while maintaining high accuracy.

Contributions. In summary, our main contributions are:

1. We provide a detailed analysis of the limitations of existing FIFO-based and delay-based non-intrusive distributed tracing frameworks in achieving tracing accuracy of microservices (§ 2.3).

2. We design and implement FlowScope, a non-intrusive tracing framework based on method-level delay estimation. It leverages fine-grained, method-level delay distributions to accurately evaluate the probability of parent-child relationships between spans and efficiently reconstructs request traces (§ 3).
3. We conduct a comprehensive evaluation of FlowScope in testbed environments, demonstrating that it achieves over 94% tracing accuracy and successfully validates various real-world use cases (§ 4).

2 Background and Motivation

This section begins with an introduction to non-intrusive distributed tracing (§ 2.1). Then, we summarize the requirements imposed by microservice scenarios on non-intrusive tracing (§ 2.2) and provide the motivation for FlowScope (§ 2.3).

2.1 Non-Intrusive Distributed Tracing

Non-intrusive distributed tracing frameworks [3, 5, 6, 10], such as DeepFlow [52] and TraceWeaver [6], are achieved by intercepting external interaction interfaces of microservice components, such as system calls or user-space functions. They require no modifications to application source code, recompilation, or application restart, enabling *transparent request tracing without disrupting application running*. In contrast, intrusive frameworks like Jaeger [29] and Zipkin [68] rely on manually embedding tracing-related logic into the application source code, necessitating recompilation and application restart, which interferes with the application running.

Non-intrusive distributed tracing *offers significant convenience* in microservice scenarios compared to intrusive frameworks. First, components in microservices are often maintained by different teams. Non-intrusiveness allows teams to deploy and manage the tracing framework independently, reducing cross-team coordination costs. Moreover, modifying application code of intrusive frameworks is often time-consuming and requires continuous adjustments to keep pace with rapid application updates for large-scale microservices [15, 21, 49, 52, 54], whereas non-intrusive frameworks enable rapid and cost-effective deployment. Additionally, non-intrusive frameworks can seamlessly integrate with a large number of closed-source or third-party components [16, 20, 50] in microservices that cannot be modified. Thus, non-intrusive distributed tracing for microservices has recently attracted widespread attention [3, 6, 10, 51, 52, 59, 67].

2.2 Requirements of Non-Intrusive Tracing

Non-intrusive distributed tracing for microservices must meet the following key requirements to ensure its effectiveness.

Requirement-1: Multiple programming language compatibility. Modern microservices often comprise components implemented in multiple programming languages, like Python and Java. Non-intrusive tracing frameworks must be compatible with components written in different languages to ensure the integrity of request traces as they traverse heterogeneous components.

Requirement-2: Low-overhead and efficient operation. Given the substantial daily request volume in online microservice applications [30, 36], non-intrusive tracing frameworks require lightweight

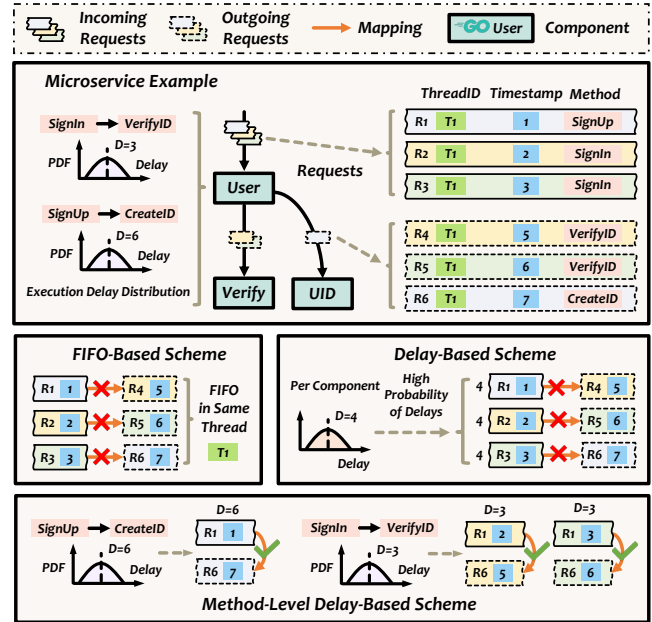


Figure 1: Comparison between FIFO-based, delay-based, and method-level delay-based schemes.

collection of per-request data and resource-efficient trace construction.

Requirement-3: Accurate end-to-end tracing. Tracing frameworks must precisely capture the end-to-end execution flow of requests as they traverse various components. This relies on *accurately associating incoming and outgoing requests with parent-child relationships* within each component. Incoming requests denote those received by a component, while outgoing requests represent those departing from the component. Parent-child relationships emerge when an incoming request (parent) initiates one or more outgoing requests (child) during its processing. Incorrect request association can lead to abnormal trace structures or data loss for faulty requests, significantly undermining the effectiveness of using traces for failure diagnosis and performance debugging. However, microservices commonly employ mechanisms such as multi-threading or asynchronous I/O for concurrent request processing. This makes it challenging to determine which incoming requests triggered which child requests from a non-intrusive perspective, which lacks visibility into the internal execution logic of components.

2.3 Motivating FlowScope

Over the years, non-intrusive distributed tracing technologies like DeepFlow [52] and TraceWeaver [6], designed for microservice troubleshooting, have attracted considerable interest [3, 9, 48, 51, 52, 61]. However, these schemes often struggle with accurately associating incoming and outgoing requests. In this paper, we categorize existing non-intrusive tracing solutions into two categories according to *request association mechanisms*: **FIFO-based** and **delay-based** approaches, detailing their principles and limitations with Figure 1. **◊ FIFO-based schemes are applicable only to specific threading models.** These schemes, such as vPath [59] and DeepFlow [52],

assume components adopt a specific thread model where *each incoming request and all its child requests are processed by a single thread, with no new requests handled until the current incoming request completes*. Consequently, these methods correlate incoming and outgoing requests within each thread following FIFO principles. However, microservice components often employ diverse threading models, invalidating these assumptions and causing request correlation errors. For instance, in the *coroutine mechanisms* of Python [46] and Golang [17], multiple coroutines can reuse a single thread, enabling concurrent request processing within one thread. Each coroutine handles one incoming request, and application code controls switching between coroutines. This flexibility disrupts the FIFO ordering of requests within a single thread, leading to request correlation errors. In the User component in Figure 1, although request R1 arrives first, its child request R5 becomes the third outgoing request, ultimately causing R5 to be incorrectly correlated with the third incoming request R3. Beyond coroutine mechanisms, microservice components may also utilize *cross-thread* operations in RPC libraries like Thrift [2] and gRPC [26] or adopt the *event-driven asynchronous threading model* (e.g., Node.js servers). These threading models further exacerbate the limitations of FIFO-based solutions.

◊ **Delay-based schemes struggle to accurately estimate delay distributions, resulting in association errors.** Delay-based solutions [4, 6, 48] estimate the distribution of delays between receiving an incoming request and the child outgoing requests it triggers, then correlate incoming and outgoing requests that exhibit high-probability delay patterns. Such techniques impose no specific threading model constraints on components, thus offering broader applicability. However, conventional delay-based solutions [3, 48] establish a *single delay distribution per component*, overlooking variations in parent-child request delays across different method types. This oversight results in inaccurate delay distribution estimation and request correlation errors. In microservice architectures, each component may expose one or more *methods*, with incoming requests specifying target methods through identifiers such as URLs or interfaces. Upon processing an incoming request for a specific method, a component triggers one or more outgoing requests to other component methods after varying delays determined by application logic. Consequently, distinct method combinations exhibit differentiated parent-child delay distributions. For example, within the User component, the SignIn method invokes the VerifyID method during execution, exhibiting a typical parent-child latency of 3 for SignIn-VerifyID pairs. In contrast, the SignUp-CreatelD method pair demonstrates a typical delay of 6 due to differing code logic. Traditional delay-based approaches model all method combinations within the User component using a single delay distribution (e.g., peaking at 4), preventing accurate probability estimation of parent-child relationships between incoming and outgoing requests and causing correlation errors.

Among recent delay-based solutions, only TraceWeaver [6] considers this delay variation. Nevertheless, this approach requires preliminary knowledge of method invocation relationships, potentially obtained from biased testing environments, and fails to adapt to dynamic invocation scenarios influenced by factors like component caching or method parameter variations. For instance,

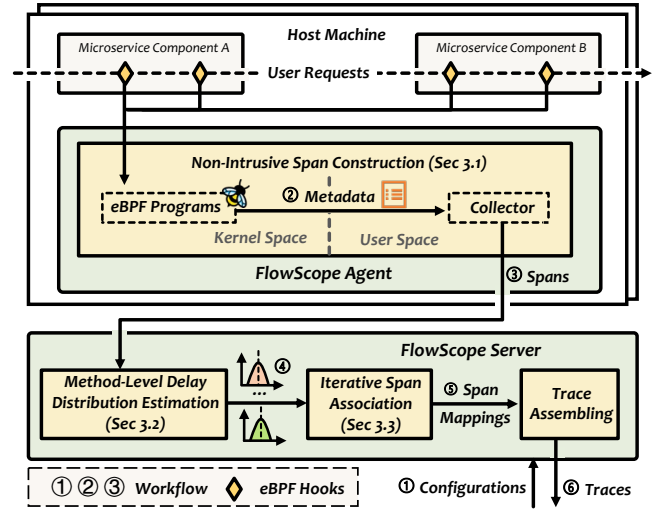


Figure 2: The overview of FlowScope framework.

an incoming request for method A may trigger an outgoing request to either method B or method C.

◊ **Our approach: method-level delay-based.** In this paper, we define *method-level delay distribution* as the *delay distribution of parent-child requests corresponding to distinct method pairs*, where each method pair comprises a parent request's method and a child request's method. For example, requests R1 and R6 form a method pair of <SignIn, CreatelD>. To accurately correlate incoming and outgoing requests, it is essential to estimate the method-level delay distribution between the methods of incoming and outgoing requests within each component, without relying on prior knowledge of method invocation relationships. For instance, within the User component, delay distributions must be constructed for method pairs such as <SignIn, VerifyID> and <SignUp, CreatelD>, while concurrently excluding invalid pairs like <SignIn, CreatelD> that exhibit no invocation relationship and consequently cannot form valid delay distributions. Subsequently, these method-specific delay distributions enable the evaluation of delay probability between incoming and outgoing requests of specific method pairs, serving as a measure of their parent-child probability.

3 The FlowScope Framework

FlowScope workflow. FlowScope is a non-invasive distributed tracing framework that employs a *span-based model* to construct traces. Each **span** represents a request-response pair, documenting the complete execution lifecycle of an individual request within a component, including metadata such as start time, end time, and invoked methods. Every trace forms a *span tree* where spans serve as nodes interconnected by parent-child relationships as edges. Figure 2 illustrates the workflow of FlowScope, which consists of two main components: an **agent** deployed on each host and a **server** operating across the microservice cluster. The agent is responsible for collecting metadata for each request (e.g., timestamps, methods, etc.) and correlating requests and responses to construct spans, while the server is responsible for estimating method-level delay distributions, associating spans, and assembling traces.

First, the FlowScope server receives *adjustable configurations* from operators as **inputs** to perform initialization (①). The main configuration is the *iteration step k* (default is 100) for performing span associations. After initialization, the agent performs **non-intrusive span construction** (§ 3.1) by collecting request metadata through eBPF programs (②) and correlating requests with their corresponding responses to build spans (③). These spans are transmitted to the server for trace reconstruction. On the server, FlowScope constructs delay distributions between arbitrary method pairs within each component (④) through the **method-level delay distribution estimation** (§ 3.2). Subsequently, during **iterative span association** (§ 3.3), FlowScope utilizes these fine-grained delay distributions to evaluate the probability of a parent-child relationship between incoming and outgoing spans within each component. FlowScope then associates incoming and outgoing spans with high parent-child probabilities to generate *span mappings*, each consisting of one parent span and one child span (⑤).

During the final **trace assembling**, FlowScope connects spans according to parent-child relationships to construct complete traces (⑥). In addition to the parent-child relationships between incoming and outgoing spans within each individual component, building the trace also requires the association of spans *across components*. For instance, when component *A* calls component *B*, we need to associate *A*'s outgoing span with *B*'s incoming span that belongs to the *same request-response pair*. DeepFlow achieves inter-component span association using connection information such as IP, port, and TCP sequence numbers, but this inference can lead to ambiguity if network address translation or tunneling modifies these packet details. FlowScope generates a random 64-bit globally unique identifier and inserts it into the transport-layer packet options of the message using eBPF, thereby enabling precise cross-component span correlation.

3.1 Non-Intrusive Span Construction

To construct request traces, FlowScope first needs to collect metadata for each request, including *protocol type, timestamp, method, request/response type, and other details*, and then matches requests and responses to generate spans. To achieve multi-language compatibility (**requirement 1**) without modifying application code, language-specific instrumentation techniques should be avoided. For instance, tools such as bytecode instrumentation in Java [45] or monkey patching in Python [53] can intercept user-space communication functions to collect request data, but they are generally limited to specific languages. For example, SkyWalking's bytecode enhancement technology [5] relies on dynamic modification capabilities in languages like Java, making it incompatible with languages compiled directly to machine code (such as Go or Rust). Moreover, these language-specific approaches require customized collection solutions for each language, including adjustments to target functions for interception and modifications to the internal logic of collection code, which complicates the achievement of multi-language compatibility.

★ **DESIGN 1. In-kernel hook-based request collection.** We observe that the execution of user requests is primarily driven and constrained by the narrow-waist communication pattern manifest in transport-layer APIs. Consequently, we identify ten network-related system calls and designate them as trigger hooks for data

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

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

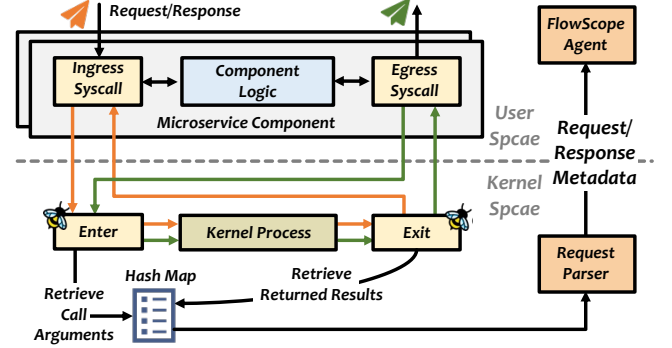


Figure 3: FlowScope collects request and response in the kernel space via pre-defined hooks.

collection, as detailed in Table 1, where **ingress** and **egress** represent the functions for receiving and sending requests, respectively. Regardless of the diversity and complexity of microservice components or their implementation languages, their execution is invariably initiated by communication behaviors. FlowScope therefore attaches multiple eBPF programs at the *entry* and *exit* points of these system call functions. These eBPF programs automatically activate whenever a microservice component sends or receives a request, thereby enabling the collection of fine-grained request data.

Span construction process. Figure 3 illustrates the request collection workflow. For requesting ingress or egress, registered kprobe or tracepoint hooks trigger when the corresponding system calls enter and exit the kernel. Upon function entry, FlowScope stores information such as the payload, five-tuple, and process ID of the system call in a hash map. After kernel processing completes, FlowScope retrieves the return results (such as the total length of written data) and correlates them with the temporarily stored data from the entry point using the hash map. This merged data is subsequently transmitted to FlowScope's user-space agent for processing.

In the user-space agent, FlowScope utilizes open-source protocol parsing tools to extract method names and determine request or response types from the payload. To construct spans, it then correlates requests and responses according to different transmission modes. For *serialized transmission* protocols such as HTTP/1.1 with persistent connections, each request over the same connection must await the completion of the previous response before being sent. In *pipelining mode*, multiple requests can be sent successively within a single connection, and responses are returned in the same order. For both patterns, FlowScope correlates responses with requests based on their sequence within the connection. For protocols enabling *parallel transmission* through multiplexed messages, such as HTTP/2, FlowScope uses embedded distinguishing attributes in the messages to correlate requests and responses, such as the stream identifier in HTTP/2 headers.

3.2 Method-Level Delay Distribution Estimation

After obtaining the spans, FlowScope needs to evaluate the parent-child relationship probability between the ingoing and outgoing spans within each component. An incoming span represents a request-response pair received by the component, whereas an outgoing span denotes a request-response pair sent from the component. To ensure tracing accuracy (**requirement 3**), FlowScope employs a parent-child probability assessment strategy based on *method-level delay distributions*. For any method pair, FlowScope estimates two types of delay distributions: the delay distribution $P_s(\cdot)$ between their start times and the delay distribution $P_e(\cdot)$ between their end times. Taking the method pair $\langle M_1, M_2 \rangle$ in Figure 4 as an example, when component A processes an incoming request invoking method M_1 , it triggers an outgoing request to invoke method M_2 in component B, and continues processing after receiving the response from B until it returns a response to the incoming request. For this method pair, two distributions are constructed: the first distribution, $P_s(\cdot)$, characterizes the time interval between the request arriving at method M_1 and the subsequent issuance of the corresponding outgoing request to method M_2 (e.g., $t_2 - t_1$); the second distribution, $P_e(\cdot)$, captures the time interval between receiving the response from method M_2 and returning the corresponding response at method M_1 (e.g., $t_6 - t_3$). Once these distributions are obtained, the parent-child relationship probability between any ingoing span s_1 belonging to method M_1 and any outgoing span s_2 belonging to method M_2 can be calculated using the formula:

$$P(s_1, s_2) = \frac{P_s(\Delta t_s) + P_e(\Delta t_e)}{2} \quad (1)$$

where Δt_s denotes the start time difference between s_1 and s_2 , and Δt_e represents their end time difference. It is important to note that estimating such distributions is challenging, as in environments without a tracing system, we cannot obtain the actual parent-child relationships between requests arriving at method M_1 and those invoking method M_2 , making it impossible to acquire accurate delay samples.

★ DESIGN 2. Method-level delay distribution estimation based on cross-correlation analysis. The core idea of FlowScope's method-level delay distribution estimation is to *identify the peak probability delay T by analyzing the cross-correlation between request sequences of different methods*. For a given method, its *request sequence* refers to a sequence formed by the number of requests invoking that method within each fixed time interval on a discrete timeline. The peak probability delay T represents the most typical delay duration between a parent request and its corresponding child request. This indicates that for a specific method pair $\langle m_1, m_2 \rangle$, after method m_1 receives an incoming request, it typically initiates an outgoing child request to method m_2 after approximately delay T . Therefore, if the request sequence of m_1 is shifted backward by this peak probability delay T , the largest number of parent requests in the shifted m_1 sequence and child requests in the m_2 sequence will fall within the same time interval. At this point, the shifted request sequence of m_1 and the request sequence of m_2 exhibit the strongest similarity. Consequently, the peak probability delay T can be determined by identifying the specific time shift that maximizes the similarity between the request sequence of m_1 and the request sequence of m_2 . For instance, as illustrated

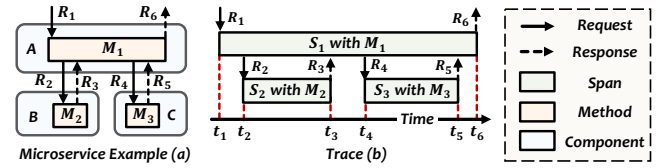


Figure 4: An example of a trace (b) obtained from a microservices application example (a).

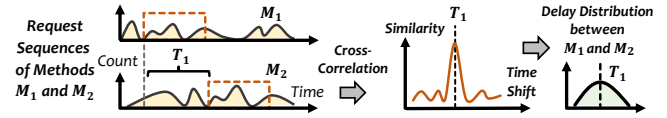


Figure 5: Estimation of delay distribution between different methods based on cross-correlation analysis.

in Figure 5, it can be observed that within the dashed box region, the similarity between the request sequence of method M_1 and that of M_2 reaches its maximum when the request sequence of M_1 is shifted by the peak probability delay T_1 . Cross-correlation [12] can measure the similarity between one sequence and another when shifted by different time intervals. Therefore, we can determine the peak probability delay T by analyzing the cross-correlation between two request sequences, thereby constructing the delay distribution. The *advantage* of this scheme is that it does not rely on the test environment and is robust to complex scenarios such as dynamic method invocations. As long as there is a subset of requests with a parent-child relationship between any two request sequences, this scheme can be effective.

For a set of requests corresponding to a method, let $N(t)$ denote the total number of requests within the time range $[0, t)$. The request set is transformed into a request sequence: $Q(n) = N((n+1)\mu) - N(n\mu)$, where μ represents the time interval and the integer n denotes the interval index. A smaller μ reduces the error in estimating the peak probability delay but increases the computational complexity of cross-correlation calculations. Therefore, to balance accuracy and efficiency, μ is set to half the average time intervals between adjacent requests in the original set of requests. Given two request sequences Q_1 and Q_2 , the similarity between Q_2 and Q_1 shifted by different time intervals can be expressed using cross-correlation as $Q_2 \otimes Q_1 = \int_0^{n_{\max}} Q_2(n)Q_1(n+\tau)d\tau$, where $n_{\max}$ represents the maximum number of intervals between the sequences Q_1 and Q_2 . Afterwards, FlowScope extracts the peaks from the cross-correlation results according to the 3-sigma rule. To avoid noise, we require that at least one point between adjacent peaks fall within the 3-sigma normal range. Based on the time interval n^* corresponding to the highest peak, the peak probability delay can be calculated as $T = n^* \times \mu$. Using T as the mean, a Gaussian distribution $P(\cdot)$ is constructed to model the delay distribution between the two request sets. FlowScope uses a fixed default value as the variance of the Gaussian distribution because the variance does not affect the relative magnitude of probabilities across different delays.

FlowScope estimates method-level delay distributions within each component. First, it groups the incoming and outgoing spans of the component by their methods, then converts the spans within

each group into request sequences and response sequences. For any method pair composed of an incoming request method m_1 and an outgoing request method m_2 , FlowScope constructs the delay distribution $P_s(\cdot)$ by performing cross-correlation analysis between the request sequences of m_1 and m_2 , and similarly builds the delay distribution $P_e(\cdot)$ using their response sequences. Note that not all method pairs yield a probability peak delay T to form such distributions. When no invocation relationship exists between two methods, their request sequences remain entirely independent and random, exhibiting no time-shifted similarity. For example, in Figure 4, if method M_4 in component A triggers calls to method M_5 in another component, the method pair $\langle m_1, m_2 \rangle$ would not exhibit a probability peak delay T . When delay construction fails, FlowScope converts the corresponding method pair $\langle m_1, m_2 \rangle$ into a *method invocation constraint* (**Constraint 1**, C_1), indicating that an incoming request of method m_1 cannot be associated with an outgoing request of method m_2 .

In addition to estimating the delay distributions between incoming and outgoing methods, we can also identify *method order constraints* (**Constraint 2**, C_2) by evaluating the delay distributions between any pair of outgoing methods. This constraint reflects the order relationship among outgoing spans triggered by the same incoming span, influenced by factors such as data dependency and state synchronization. For instance, in Figure 4, the incoming span of method M_1 always triggers the outgoing span of M_2 first and subsequently initiates the outgoing span of M_3 after the completion of M_2 's span. Thus, a method order constraint exists between M_2 and M_3 : when spans of both M_2 and M_3 are associated with the same incoming span, the end time of M_2 's span must precede the start time of M_3 's span. For any pair of outgoing methods, FlowScope assesses whether a probability peak delay exists between the response sequence of one method and the request sequence of the other. If such a delay is detected, a method order relationship is confirmed for the method pair.

3.3 Iterative Span Association

After obtaining the method-level delay distribution, FlowScope must correlate incoming and outgoing spans within each component that are likely to form high probabilities of delays. FlowScope assumes that *the response of each parent span can only be sent out after all its child spans have completed*. Based on this assumption, FlowScope first constructs *candidate span mappings*. Each candidate span mapping consists of one incoming span and one outgoing span, where the start time of the outgoing span must be greater than that of the incoming span, and its end time must be smaller than that of the incoming span. Within a single component, given a set of incoming spans $S = \{s_i \mid i \leq |S|\}$ and a set of outgoing spans $D = \{d_j \mid j \leq |D|\}$, FlowScope iterates through all spans to generate candidate span mappings, denoted as $E_{cand} = \{e_{ij}\}$ where $e_{ij} = 1$. For each candidate span mapping $\langle s_i, d_j \rangle$, FlowScope extracts its corresponding method-level delay distribution $P(\cdot)$ and uses Equation 1 to estimate its parent-child probability: $P(s_i, d_j)$. FlowScope then selects an optimal set of span mappings E^* from the candidate mappings E_{cand} such that Constraints 1, 2, and each outgoing span has at most one parent (**Constraint 3**, C_3) are satisfied. The objective is to maximize the sum of the parent-child

probabilities over all selected span mappings in E^* , which can be formally expressed as:

$$\max_{E^* \subseteq E_{cand}} \sum_{e \in E^*} P(s_i, d_j) \times e_{ij} \quad (2)$$

This optimization problem is NP-Hard [42], making it impossible to derive an exact solution. Although some solvers [27] can perform approximate computations, the processing time increases exponentially with the number of spans.

★ **DESIGN 3. A top-k iterative span association mechanism with constraint conflict resolution.** To enhance the efficiency of span association (**requirement 2**), FlowScope employs a *heuristic iterative span association* algorithm, as outlined in Algorithm 1. Rather than directly deriving the complete set of span mappings E^* , FlowScope adopts an iterative approach wherein, during each iteration, it selects the span mapping from the top-k candidate mappings that satisfies all three constraints and exhibits the highest parent-child probability sum, thereby reducing computational complexity. Since the constraint C_1 applies independently to each candidate span mapping, FlowScope first pre-processes the candidate set by eliminating any mappings that violate the method invocation relationship constraints before initiating the iterative procedure (line 1). Then, FlowScope calculates the parent-child relationship probability for each candidate mapping according to the corresponding method-level and ranks them in descending order of probability (line 2).

During the iterative span associations (lines 3-20), the top k mappings with the highest probabilities are iteratively removed from the candidate mapping set E_{cand} , and a subset L_{best} is selected from these k mappings that satisfies constraints C_2 and C_3 while maximizing the sum of probabilities. This process repeats until E_{cand} is empty. Within the loop, FlowScope first initializes a list $\mathcal{L}$ to store all possible candidate mapping subsets that meet constraints C_2 and C_3 (line 4). Since constraints C_2 and C_3 operate between different mappings, each time a new mapping is extracted from E_{cand} , we traverse the current candidate mapping subsets to determine whether it conflicts with existing mappings. If the new mapping does not conflict with any mappings in a subset, it is directly added to that subset (line 10). If a conflict exists, a new candidate mapping subset is created by removing the conflicting mappings from the original subset (lines 11-13). In the final step within the iteration, FlowScope identifies the subset L_{best} with the maximum probability sum, adds it to the final association results, and removes mappings from E_{cand} that conflict with L_{best} (lines 17-19).

4 Evaluation

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

- The eBPF program for data collection in FlowScope introduces an overhead of approximately 3 microseconds (§ 4.1).
- FlowScope maintains an end-to-end tracing accuracy of over 91% even in high-concurrency scenarios (§ 4.2).
- FlowScope demonstrates the best performance across multiple real-world use cases (§ 4.3).
- FlowScope results in only a 5% degradation in application end-to-end performance (§ 4.4).

Algorithm 1 Iterative Span Association Algorithm

Input: E_{cand} : candidate span mappings, k : iteration step, C : constraints

Output: E : final span mappings

```

1:  $E_{cand} \leftarrow \text{REMOVECONSTRAINT}(E_{cand}, C_1)$ 
2:  $\text{SORTBYPROBABILITY}(E_{cand})$   $\triangleright$  Prioritize high-probability mappings
3: while  $E_{cand} \neq \emptyset$  do
4:    $\mathcal{L} \leftarrow \{\emptyset\}$   $\triangleright$  Initialize candidate solution list
5:   for  $iter \leftarrow 1$  to  $k$  do
6:      $e_{best} \leftarrow \text{POPBEST}(E_{cand})$ 
7:      $\mathcal{L}_{new} \leftarrow \emptyset$ 
8:     for all  $L \in \mathcal{L}$  do
9:       if  $\text{COMPLYCONSTRAINTS}(L, e_{best}, C_2, C_3)$  then
10:         $L \leftarrow L \cup \{e_{best}\}$ 
11:       else
12:         $L' \leftarrow \text{REMOVECONFLICT}(L, e_{best}, C_2, C_3)$ 
13:         $L' \leftarrow L' \cup \{e_{best}\}$ 
14:         $\mathcal{L}_{new} \leftarrow \mathcal{L}_{new} \cup \{L'\}$ 
15:       if  $\mathcal{L}_{new} \neq \emptyset$  then
16:         $\mathcal{L} \leftarrow \mathcal{L} \cup \mathcal{L}_{new}$ 
17:    $L_{best} \leftarrow \arg \max_{L \in \mathcal{L}} \text{SUMPROBABILITY}(L)$ 
18:    $E \leftarrow E \cup L_{best}$ 
19:    $\text{REMOVECONFLICT}(E_{cand}, L_{best}, C_2, C_3)$ 
20: return  $E$ 

```

Testbed and workload setup. Our testbed comprises a three-node Kubernetes cluster [32] (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. We deploy two microservices, the social networking application from DeathStarBench [23] and the Train-Ticket application [22]. The former is a social network application consisting of 36 components. The latter provides standard train ticket booking functionalities such as ticket inquiry, payments, and user notifications, encompassing 41 unique components developed in multiple programming languages. These applications integrate a variety of concurrency mechanisms, including the thread pools of the Thrift library and asynchronous I/O from the C++ standard library. Employing wrk2 [60] as the load generator, the request arrival intervals are regulated using an exponential distribution at various levels of requests per second (RPS), with each test maintained for a duration of 30 seconds. To mitigate systematic errors, each experiment averages the results from multiple test iterations across both applications. For quantifying the accuracy of non-intrusive tracing, the intrusive tracing framework Jaeger [29] serves as the ground truth reference. By injecting unique contextual identifiers (such as traceIDs) into parent and request payloads, a benchmark for request parent-child relationships is established based on the Jaeger context extracted from the captured request payloads. When evaluating different non-intrusive tracing algorithms, their tracing accuracy is assessed by comparing the tracing results against this established reference standard.

Baselines. For the *end-to-end tracing accuracy*, we compare FlowScope with *FIFO-based* DeepFlow [52] and vPath [51], as well as

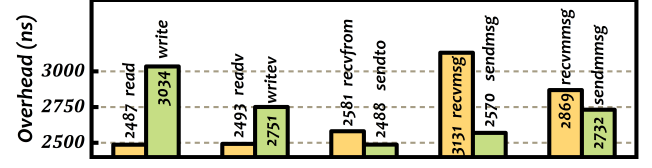


Figure 6: Time overhead of different eBPF programs.

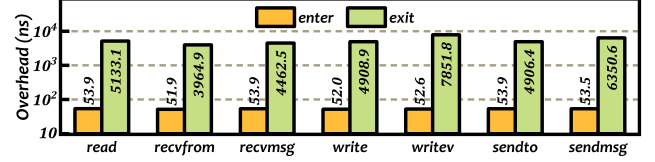


Figure 7: Collection overhead in microservice applications.

delay-based TraceWeaver [6] and WAP5 [48]. DeepFlow matches each outgoing request with the nearest incoming request in the same thread. vPath assumes that all outgoing requests triggered by a single incoming request are processed by the same thread D and correlates these outgoing requests with the incoming request whose response is sent by D . WAP5 builds the delay distribution by statistically analyzing the delays between each outgoing request and its most recent preceding incoming request. TraceWeaver, on the other hand, organizes requests according to method invocation relationships in the test environment. For each group, it computes the difference between the average timings of outgoing and incoming requests to determine the mean of the delay distribution, subsequently modeling it as a Gaussian distribution. We randomly select a subset of traces from the ground truth to provide the test environment.

4.1 Overhead of Data Collection

In this experiment, we measure the collection overhead introduced by the eBPF programs of FlowScope in both a *microbenchmark* and the *social network microservices application*. First, we construct a microbenchmark written in C, which consists of multiple code segments, each repeatedly invoking one type of system call listed in Table 1. We measure the average execution time per system call by repeating each invocation 100,000 times and compare the results before and after deploying the eBPF programs to quantify the additional latency introduced by FlowScope. As shown in Figure 6, the extra delay caused by these eBPF programs ranges between 190 and 240 nanoseconds. Compared to the inherently longer duration of network communication operations, this overhead is negligible.

Subsequently, we further evaluate the collection overhead of FlowScope's eBPF programs in the real-world social network microservices application. Since each system call involves two eBPF hooks: *enter* and *exit*, we record timestamps at the beginning and end of each eBPF program to measure its execution time. Figure 7 indicates that the enter and exit eBPF programs introduce approximately 50 nanoseconds and 6 microseconds of additional latency, respectively. It should be noted that the components in the social network microservices only trigger seven types of system calls. The longer execution time of the exit eBPF programs is due to their responsibilities for data aggregation and copying data to user space, whereas the overhead of the enter eBPF programs is negligible.

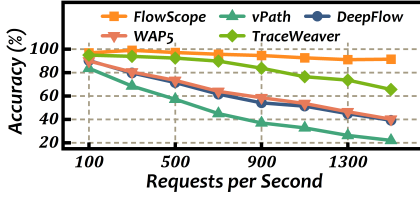


Figure 8: End-to-end tracing accuracy under different loads.

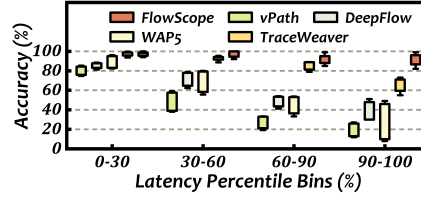


Figure 9: End-to-end tracing accuracy binned by end-to-end latency.

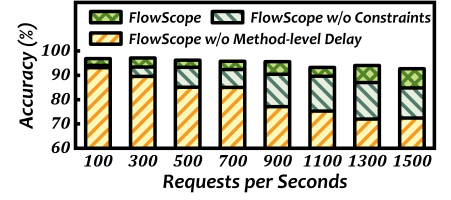


Figure 10: The impact of different modules of FlowScope on the accuracy.

4.2 Tracing Accuracy

Tracing accuracy under different loads. We test the impact of increasing load (requests per second) on end-to-end tracing accuracy. Increasing the load raises the concurrency level at which the application processes requests simultaneously. Figure 8 shows the end-to-end accuracy of FlowScope and baseline methods under different loads. As the load increases, higher parallelism leads to more frequent request reordering, significantly reducing the accuracy of baseline methods. FIFO-based vPath and DeepFlow fail to accurately handle cross-thread and asynchronous I/O in microservice applications, leading to tracing errors. Meanwhile, delay-based WAP5 does not account for differences in delay distributions across methods, also resulting in tracing errors. These solutions achieve less than 40% tracing accuracy at 1500 RPS. TraceWeaver, which constructs method-level delay distributions, demonstrates better accuracy but only reaches 62% at 1500 RPS. This is due to its inability to accurately construct method-level delay distributions in a dynamic method invocation environment due to component caching, method parameter changes, etc. In contrast, FlowScope consistently maintains more than 91% tracing accuracy through accurately constructed method-level delay distributions, method constraint relations, and other optimizations.

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 9, *incorrect traces of all baselines are concentrated in the tail requests*. This is because, in tail requests, the disorder between incoming and outgoing requests is more severe, further reducing the association accuracy of the FIFO-based approach, which drops to around 40% for the tail 10% of requests. In traditional delay-based approaches, even the most accurate solution, TraceWeaver, achieves only 60% accuracy for the tail 10% of requests. In contrast, FlowScope leverages method-level delay distributions and various constraints to *first associate incoming and outgoing requests with higher probabilities of normal delays*, thereby *reducing interference in tail request associations*. As a result, FlowScope maintains an accuracy of 84% even for the tail 10% of requests.

FlowScope ablation study. To quantify the benefits of different modules for span association, we conduct ablation experiments on FlowScope by progressively removing constraints for span mapping and method-level delay distributions. After removing the method-level delay distributions, FlowScope constructs a single delay distribution between incoming and outgoing requests for each component. As shown in Figure 10, removing constraints reduces FlowScope's tracing accuracy at 1500 RPS from 95% to 85%, demonstrating the significant role of constraints in eliminating

incorrect candidate mappings. It is important to note that the impact of method constraints varies across components. For instance, method invocation relationship constraints only take effect when a component has multiple methods and the sets of downstream methods triggered by these methods exhibit differences. After removing method-level delay distributions, FlowScope's accuracy at 1500 RPS decreases to 72%. The absence of method-level delay distributions results in the mixing of delays from parent-child request pairs across different methods, making it challenging to accurately evaluate the probability of candidate request mappings having a parent-child relationship.

4.3 Case Studies

We present two use cases in which various non-intrusive tracing frameworks are applied for microservice operation and debugging. The results demonstrate that incorrect traces generated by existing solutions can cause operational failures and severely mislead operators, whereas the accurate traces constructed by FlowScope facilitate the successful execution of operational tasks.

Use Case 1: Troubleshooting tail latency requests. Microservices impose stricter tail latency requirements compared to typical cloud-based services [18, 23, 24, 35, 43]. For example, Amazon states that a 100 ms increase in page load time leads to a 1% decline in sales [31]. As a result, operators place a high priority on *quickly and accurately identifying faulty components that cause end-to-end tail latency in requests*. In this case, we randomly select components in the social networking application and inject a 40 ms delay with a 10% probability to simulate tail latency anomalies. We then employ tprof [28], a representative work for tail latency diagnosis, to perform fault component localization using traces constructed by different frameworks. Figure 11 shows that at 1500 RPS, the accuracy of fault component localization using traces built by existing frameworks is only about 50%. This is because faulty spans are incorrectly associated with some normal traces, leading to the failure of fault component localization. This forces operators to manually sift through numerous incorrect results, severely impeding fault resolution efficiency. Conversely, FlowScope's accurate traces achieve an 89% localization accuracy, enabling swift identification and repair of faulty components.

Use Case 2: Critical path analysis. During the execution of a microservice user request, the *critical path* is often referred to as the longest chain of visited components [63, 66]. Operators often analyze the critical paths extracted from traces to carry out tasks such as locating performance bottlenecks, resource allocation planning, and response time optimization [30, 37, 47, 66]. *The successful completion of these tasks depends on the accuracy of the trace structure*. In this case, ground truth trace data serves as the input to

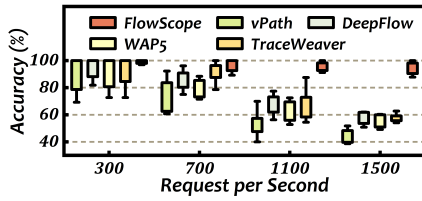


Figure 11: The accuracy of failure localization using traces constructed by different schemes.

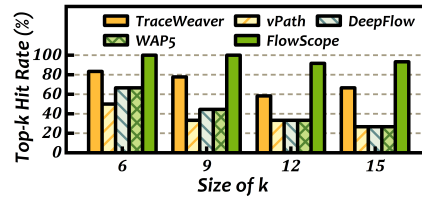


Figure 12: The hit rate of the top-k length critical paths constructed by traces from different frameworks.

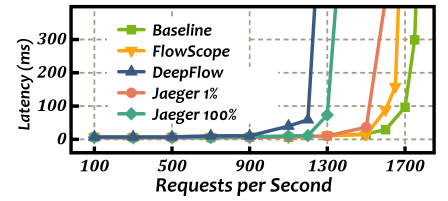


Figure 13: Impact on application end-to-end performance of different frameworks.

extract the actual longest k critical paths. Subsequently, the top- k critical paths are derived from traces constructed by various non-intrusive tracing frameworks, and their top- k hit rate is computed against the ground truth. Figure 12 shows that when k equals 15, TraceWeaver achieves a top- k critical path hit rate of only 60%, while other methods reach around 20%. In contrast, FlowScope maintains a hit rate consistently above 90%. Incorrect critical paths can lead to the failure of various operational tasks. For example, in resource allocation tasks, assigning more resources to components on incorrect critical paths may result in resource waste, as these components lie outside the actual critical paths.

4.4 End-to-end Performance

Finally, we evaluate the end-to-end performance of FlowScope.

Efficiency of trace construction. Table 2 shows the time required by these frameworks to construct traces for varying numbers of requests. FlowScope achieves trace construction speeds comparable to FIFO-based solutions and is more than 10 times faster than the delay-based TraceWeaver. This is because FIFO-based solutions only require sorting requests and matching them in order, while TraceWeaver employs a complex solver to compute the span set with the highest delay probability, resulting in the lowest efficiency. By adopting a heuristic iterative association strategy, FlowScope significantly improves trace construction speed.

Number of Requests	FlowScope	vPath	DeepFlow	WAP5	TraceWeaver
8 K	26.95 s	1.06 s	1.36 s	14.36 s	281.56 s
10 K	33.30 s	1.24 s	1.64 s	17.71 s	347.83 s

Table 2: Comparison of time cost for trace construction.

End-to-end performance overhead. Finally, we evaluate the impact of FlowScope on the end-to-end performance of microservice applications. We use the application performance without any tracing framework deployed as the *baseline*. Then, we measure the application performance after deploying different tracing frameworks. For Jaeger, we use sampling rates of 1% and 100%, respectively. Figure 13 shows that FlowScope introduces the smallest end-to-end performance overhead. The baseline achieves a maximum throughput of approximately 1,749 RPS, while deploying FlowScope results in a maximum throughput of 1,667 RPS, causing only a 5% performance degradation.

5 Related Work

Distributed tracing. Traditional intrusive distributed tracing systems, such as Dapper [55] and Pinpoint [40], established the foundation of observability. With the evolution of distributed systems,

recent research has further advanced the field: Pivot Tracing [39] breaks application boundaries by carrying metrics along request flows, while [38] decouples intrusive context propagation from cross-cutting instrumentation logic. OpenTelemetry [44] standardizes data collection APIs, instrumentation libraries, and semantic conventions. The latest distributed tracing frameworks focus on non-intrusive techniques [6, 52]. Distributed tracing is also widely applied in performance diagnosis, fault localization, and root cause analysis. For instance, TraceLingo [62] and DeepTraLog [64] employ Tree-LSTM and GNNs, respectively, to capture the tree-structured patterns in traces for anomaly detection.

Causality inference. Existing work advances non-intrusive inference of request causality through multiple approaches. One method assumes that components adopt specific thread models and infers intra-component request execution flows by analyzing thread activities. For instance, vPath [59] identifies request processing paths by examining thread and network activity logs, while DeepFlow [52] tracks interactions by correlating temporally adjacent network send and receive operations within the same thread. A second approach focuses on analyzing the characteristics of causal delay distributions between requests [3, 6, 7, 14, 48] and selects request mappings that align with these distributions. For example, WAP5 [48] employs an exponential latency distribution assumption to predict service dependencies, whereas TraceWeaver [6] constructs a Gaussian latency distribution by estimating the mean and variance of causal delays. A third category utilizes language-specific instrumentation techniques to intercept application functions processing requests, thereby capturing causal relationships. Examples include SKYWalking's bytecode injection [5] and Python's monkey patching [53], though these are typically limited to specific programming languages.

eBPF. In recent years, the utilization of eBPF in microservice observability has gradually expanded, primarily demonstrated in the following aspects: DeepFlow [52] provides out-of-the-box tracing through a network-centric tracing plane and implicit context propagation. vNetTracer [58] relies on eBPF to dynamically inject user-defined tracing programs into live virtualized networks. ViperProbe [33] offers dynamic sampling and in-depth, diverse, and precise collection of system metrics. CaT [19] captures both the context and content of network and storage events. Pyroscope [1] integrates eBPF for continuous profiling, correlating CPU/Memory/GPU flame graphs with distributed traces.

6 Conclusion

In this paper, we propose FlowScope, a non-intrusive distributed tracing framework based on method-level delay distribution estimation, which effectively addresses the limitations of existing FIFO-based and delay-based non-intrusive tracing approaches in microservice tracing accuracy. The framework leverages eBPF to achieve language-agnostic request capture, constructs fine-grained method-level delay distributions through cross-correlation analysis, and significantly improves tracing precision and efficiency by integrating constraint optimization with an iterative association algorithm. Experimental results demonstrate that FlowScope maintains over 91% tracing accuracy even under high-concurrency scenarios, introduces less than 5% performance overhead, and has been successfully deployed in multiple real-world use cases.

Acknowledgments

We are grateful to the anonymous ICSE 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. 62572269 and No. 62394322.

References

- [1] 2022. pyroscope. (May 2022). <https://github.com/grafana/pyroscope.git>
- [2] 2025. Apache thrift. (Jan. 2025). <https://thrift.apache.org/>
- [3] 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.
- [4] Animashree Anandkumar, Chatschik Bisdikian, and Dakshi Agrawal. 2008. Tracking in a spaghetti bowl: monitoring transactions using footprints. In *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 133–144.
- [5] Apache. 2023. Apache SkyWalking. (July 2023). Retrieved Jul, 2023 from <https://skywalking.apache.org/>
- [6] Sachin Ashok, Vipul Harsh, Brighton 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.
- [7] 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.
- [8] 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.
- [9] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for request extraction and workload modelling.. In *OSDI*, Vol. 4. 18–18.
- [10] 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 (HOTOS'03)*. USENIX Association, USA, 15.
- [11] 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.
- [12] Paul Bourke. 1996. Cross correlation. *Cross Correlation*, *Auto Correlation—2D Pattern Identification* 596 (1996).
- [13] Phil Calçado. 2022. Building Products at SoundCloud —Part I: Dealing with the Monolith. (Aug. 2022). <https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith>
- [14] 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.
- [15] 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.
- [16] Oracle Corporation. 2022. MySQL. (Nov. 2022). <https://www.mysql.com>
- [17] Russ Cox. 2025. Coroutines for Go. (Aug. 2025). <https://research.swtch.com/coro>
- [18] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 127–144. <https://doi.org/10.1145/2541940.2541941>
- [19] Tânia Esteves, Francisco Neves, Rui Oliveira, and João Paulo. 2021. CAT: Content-Aware Tracing and Analysis for Distributed Systems. In *Proceedings of the 22nd International Middleware Conference (Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 223–235. <https://doi.org/10.1145/3464298.3493396>
- [20] Inc. F5. 2022. NGINX: Advanced Load Balancer, Web Server, & Reverse Proxy. (Nov. 2022). <https://www.nginx.com>
- [21] 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 (NSDI'07)*. USENIX Association, USA, 20.
- [22] FudanSELab. 2024. Train Ticket - A Benchmark Microservice System. "https://github.com/FudanSELab/train-ticket". (Aug. 2024).
- [23] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyali Rath, 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 (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. <https://doi.org/10.1145/3297858.3304013>
- [24] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 19–33. <https://doi.org/10.1145/3297858.3304004>
- [25] Yoni Goldberg. 2022. Scaling Gilt: from Monolithic Ruby Application to Distributed Scala Micro-Services Architecture. (Aug. 2022). <https://www.infoq.com/presentations/scale-gilt/>
- [26] Google. 2024. gRPC. (Jan. 2024). <https://grpc.io/>
- [27] LLC Gurobi Optimization. 2025. Gurobi Optimizer. (Jan. 2025). <https://github.com/Gurobi>
- [28] Lexiang Huang and Timothy Zhu. 2021. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *Proceedings of the ACM Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, USA, 76–91.
- [29] Jaeger. 2023. Jaeger: open source, end-to-end distributed tracing. (Jan. 2023). <https://www.jaegertracing.io/>
- [30] 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 Jun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 34–50. <https://doi.org/10.1145/3132747.3132749>
- [31] Ron Kohavi and Roger Longbotham. 2007. Online Experiments: Lessons Learned. *Computer* 40, 9 (2007), 103–105. <https://doi.org/10.1109/MC.2007.328>
- [32] Kubernetes. 2024. Production-Grade Container Scheduling and Management. "https://kubernetes.io/". (Aug. 2024).
- [33] Joshua Levin and Theophilus A Benson. 2020. Viperprobe: Rethinking microservice observability with ebpf. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEEE, 1–8.
- [34] Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. 2021. MicroHECL: High-Efficient Root Cause Localization in Large-Scale Microservice Systems. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '21)*. IEEE Press, Madrid, ES, 338–347. <https://doi.org/10.1109/ICSE-SEIP52600.2021.00043>
- [35] David Lo, Lihong Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads. In *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA '14)*. IEEE Press, Minneapolis, MN, USA, 301–312.
- [36] 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. <https://doi.org/10.1109/TPDS.2022.3174631>
- [37] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. 2022. Erms: Efficient resource management for shared microservices with SLA guarantees. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. Association for Computing Machinery, New York, NY, USA, 62–77.
- [38] Jonathan Mace and Rodrigo Fonseca. 2018. Universal Context Propagation for Distributed System Instrumentation. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 8, 18 pages. <https://doi.org/10.1145/3190508.3190526>
- [39] 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 (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 378–393. <https://doi.org/10.1145/2815400.2815415>
- [40] Naver. 2023. Pinpoint | Leading Open-Source APM. (Jan. 2023). <https://pinpoint-apm.github.io/pinpoint/>
- [41] Nginx. 2022. Adopting Microservices at Netflix: Lessons for Architectural Design. (Aug. 2022). <https://www.nginx.com/blog/microservices-at-netflix-architecture-l-best-practices/>
- [42] 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.
- [43] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. 2013. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *10th International Conference on Autonomic Computing (ICAC 13)*. USENIX Association, San Jose, CA, 69–82. <https://www.usenix.org/conference/icac13/technical-sessions/presentation/nguyen>
- [44] OpenTelemetry. 2023. High-quality, ubiquitous, and portable telemetry to enable effective observability. (Jan. 2023). <https://opentelemetry.io/>
- [45] Adrian Precub. 2025. Guide to Java Instrumentation. (Aug. 2025). <https://www.baeldung.com/java-instrumentation>
- [46] python. 2025. Coroutines and tasks of python. (Aug. 2025). <https://docs.python.org/3/library/asyncio-task.html>
- [47] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishanker K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, -, 805–825. <https://www.usenix.org/conference/osdi20/presentation/qiu>

- [48] 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.
- [49] 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.
- [50] Salvatore Sanfilippo. 2024. Redis. (Dec. 2024). <https://redis.io/>
- [51] 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 multitier services of black boxes. *IEEE Transactions on Parallel and Distributed Systems* 23, 6 (2011), 1159–1167.
- [52] 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 (ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 420–437. <https://doi.org/10.1145/3603269.3604823>
- [53] Karishma Shukla. 2025. Monkeying Around with Python: A Guide to Monkey Patching. (Aug. 2025). <https://dev.to/karishmashukla/monkeying-around-with-python-a-guide-to-monkey-patching-obc>
- [54] Ben Sigelman. 2024. Towards Turnkey Distributed Tracing (June 2016). (Dec. 2024). <https://medium.com/opentracing/towards-turnkey-distributed-tracing-5f4297d1736>
- [55] 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. <https://research.google.com/archive/papers/dapper-2010-1.pdf>
- [56] Akshitha Sriraman and Thomas F. Wenisch. 2018. μ Tune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 177–194. <http://www.usenix.org/conference/osdi18/presentation/sriraman>
- [57] Karan Parikh Steven Ihde. 2022. From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture. (Aug. 2022). <https://www.infoq.com/presentations/linkedin-microservices-urn/>
- [58] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. 2018. vNetTracer: Efficient and Programmable Packet Tracing in Virtualized Networks. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Vienna, Austria, 165–175. <https://doi.org/10.1109/ICDCS.2018.00026>
- [59] Byung-Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Ugaonkar, 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.
- [60] wrk2. 2023. An HTTP benchmarking tool based mostly on wrk. (Sept. 2023). <https://github.com/giltene/wrk2>
- [61] 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.
- [62] Yong Xu, Yaokang Zhu, Bo Qiao, Hongshu Che, Pu Zhao, Xu Zhang, Ze Li, Yingnong Dang, and Qingwei Lin. 2021. TraceLingo: Trace representation and learning for performance issue diagnosis in cloud services. In *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*. IEEE, 37–40.
- [63] C.-Q. Yang and B.P. Miller. 1988. Critical path analysis for the execution of parallel and distributed programs. In *[1988] Proceedings. The 8th International Conference on Distributed*. IEEE Computer Society, Los Alamitos, CA, USA, 366–373. <https://doi.org/10.1109/DCS.1988.12538>
- [64] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *Proceedings of the 44th international conference on software engineering*. 623–634.
- [65] 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>
- [66] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. {CRISP}: Critical Path Analysis of {Large-Scale} Microservice Architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 655–672. <https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou>
- [67] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 629–644.
- [68] Zipkin. 2023. Zipkin. (Jan. 2023). <https://zipkin.io/>