Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code

Junxian Shen[1 2 ◦], Han Zhang[1 4 •], Yang Xiang[2 ⋆], Xingang Shi[1 4], Xinrui Li[3], Yunxi Shen[3], Zijian Zhang[1 2], Yongxiang Wu[3], Xia Yin[3 4], Jilong Wang[1 4], Mingwei Xu[1 4], Yahui Li[1], Jiping Yin[2], Jianchang Song[2], Zhuofeng Li[2], Runjie Nie[2]

[1]Institute for Network Sciences and Cyberspace, Tsinghua University
[2]Yunshan Networks
[3]Department of Computer Science and Technology, Tsinghua University
[4]Zhongguancun Laboratory

[◦]shenjx22@mails.tsinghua.edu.cn [•]zhhan@tsinghua.edu.cn [⋆]xiangyang@yunshan.net

ABSTRACT

Microservices are becoming more complicated, posing new challenges for traditional performance monitoring solutions. On the one hand, the rapid evolution of microservices places a significant burden on the utilization and maintenance of existing distributed tracing frameworks. On the other hand, complex infrastructure increases the probability of network performance problems and creates more blind spots on the network side. In this paper, we present DeepFlow, a network-centric distributed tracing framework for troubleshooting microservices. DeepFlow provides out-of-the-box tracing via a network-centric tracing plane and implicit context propagation. In addition, it eliminates blind spots in network infrastructure, captures network metrics in a low-cost way, and enhances correlation between different components and layers. We demonstrate analytically and empirically that DeepFlow is capable of locating microservice performance anomalies with negligible overhead. DeepFlow has already identified over 71 critical performance anomalies for more than 26 companies and has been utilized by hundreds of individual developers. Our production evaluations demonstrate that DeepFlow is able to save users hours of instrumentation efforts and reduce troubleshooting time from several hours to just a few minutes.

CCS CONCEPTS

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

KEYWORDS

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

ACM Reference Format:
Junxian Shen[1 2 ◦], Han Zhang[1 4 •], Yang Xiang[2 ⋆], Xingang Shi[1 4], Xinrui Li[3], Yunxi Shen[3], Zijian Zhang[1 2], Yongxiang Wu[3], Xia Yin[3 4], Jilong Wang[1 4], Mingwei Xu[1 4], Yahui Li[1], Jiping Yin[2], Jianchang Song[2], Zhuofeng Li[2], Runjie Nie[2]. 2023. Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10, 2023, New York, NY, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3603269.3604823

Han Zhang is the corresponding author.

ACM SIGCOMM '23, September 10, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09.
https://doi.org/10.1145/3603269.3604823

1 INTRODUCTION

Large-scale online services have already departed the stage of monolithic applications. Common distributed systems, such as distributed computing [4, 5], container orchestration [37], and cloud platforms [40, 52], are frequently used to host applications such as machine learning [2, 49], microservices [44], and network function virtualization [46]. Because of decoupling, distributed systems have better scalability, flexibility, and availability than monolithic systems.

A high degree of decoupling, however, is a double-edged sword. Due to the heterogeneity and complexity of component interactions, it imposes a strain on the operation and maintenance of distributed systems, particularly those developed and deployed under a microservice architecture. To overcome the challenge of performance debugging in distributed systems, state-of-the-art solutions, also known as *distributed tracing* [13, 62, 64, 91, 121, 141], try to obtain execution duration and causality by adding instrumentation code to different components. While traditional distributed tracing frameworks work well for general services, they are still confronted with two problems when applied to microservice scenarios.

First, existing traditional frameworks are **ineffective and inefficient for rapidly evolving microservices**. According to our survey, in many production environments, various OS kernel versions coexist, and application components are often written in multiple languages. Meanwhile, the topology of microservices is becoming increasingly intricate, with some containing as many as 1,500 components [89]. For users manually performing distributed tracing, they may not have the time to instrument every single component prior to the deployment to production. For the developers of the tracing frameworks [62, 141], maintaining software development kits (SDKs) for various languages and kernel versions is time-consuming. Our survey shows that it often takes users hours to instrument tens of lines of code for a single component, indicating that current distributed tracing frameworks still lack usability.

Second, traditional methods **disregard network information and only record application-level traces**. Communications in

microservices are substantially more complex compared to typical distributed systems such as big data analytics, which often use the same transmission protocol. AWS Lambda [8], for instance, can support up to 29 different types of interactions with other services [76]. However, existing tracing techniques neglect either physical or virtual network information, making root cause analysis more difficult in such circumstances. Even though some frameworks can include sidecar information (e.g., Envoy [110]) to infer certain timepoints in the network, users are unable to bridge the semantic gap between the application layer and the network layer due to blind spots in the network infrastructure and the absence of network metrics. For example, when a network failure causes an increase in latency, application-level tracing tools can only identify duration changes between invocations of different services, whereas network monitoring tools have trouble locating anomalies in low-level, high-volume packets. Customers of DeepFlow have reported that network infrastructure is the root cause of 47.3% of performance issues, which makes the situation worse.

DeepFlow Framework. This paper tackles the above challenges by introducing DeepFlow, a distributed tracing framework intended specifically for microservices. DeepFlow facilitates both **out-of-the-box tracing** and **rapid performance problem location** with the following designs:

First, based on the insight that microservices are triggered by network communication, DeepFlow designs a *network-centric tracing plane* (Section 3.2). In its narrow-waist network-centric instrumentation model, pre-defined kernel hooks are used to perform automatic and non-intrusive tracing. DeepFlow leverages the privileged kernel space to eliminate the blind spots caused by closed-source components and network infrastructure. Moreover, the kernel space enables DeepFlow to retrieve network metrics, such as TCP retransmissions, and attach them to traces.

Second, DeepFlow proposes an *implicit context propagation* technique to achieve out-of-the-box tracing and avoid inserting identifiers into the packets (Section 3.3). Our system automatically leverages the information gathered during the instrumentation phase to construct the life spans of components. After that, we use various network and system information to assemble spans into traces.

Finally, DeepFlow uses *tag-based correlation* to provide a connection between metrics and traces (Section 3.4). With the assistance of these tags, users may examine network/component metrics related to traces, thus accelerating the root cause analysis. To minimize the computation and storage overhead, we further design a *smart encoding* of these tags.

Contributions. DeepFlow has been in production for two years, during which time it has identified over 71 critical performance anomalies for more than 26 companies and has been utilized by hundreds of individual developers. DeepFlow is now publicly available at: https://github.com/deepflowio/deepflow under an Apache-2.0 license. It has joined the CNCF landscape [38] and the eBPF application landscape [29].

In summary, our main contributions are:

1. We conduct a wide study of microservices running in various business environments and raise a new understanding of the critical requirements for their effective distributed tracing (Section 2).
2. We design DeepFlow, a distributed tracing framework that facilitates network-centric troubleshooting of microservices. It offers automatic and non-intrusive instrumentation, eliminates network blind spots, and effectively correlates network metrics with application traces. We implement and provide it as an out-of-the-box tool and will contribute it to the community for further improvements (Section 3).
3. We perform comprehensive evaluations both in production (Section 4) and testbed environments (Section 5). Our production results indicate that DeepFlow can save users hours of instrumentation efforts and reduce troubleshooting time from several hours to just a few minutes. The testbed evaluations show that DeepFlow incurs an overhead of no more than 7%.

The remainder of this paper is organized as follows: In Section 2, we discuss the background and motivation. In Section 3, we provide an overview of DeepFlow and present our design. We show the real-world production results in Section 4 and evaluate DeepFlow with our testbed in Section 5. Section 6 is related work, and Section 7 is the conclusion.

2 BACKGROUND AND MOTIVATION

This section begins with an introduction to observability and distributed tracing (Section 2.1). Then, we provide a summary of the new requirements imposed by microservice scenarios (Section 2.2) and provide the motivation for DeepFlow (Section 2.3).

2.1 Observability and Distributed Tracing

Observability tools are currently utilized to gain crucial insights into complex distributed systems. By collecting information such as system states, they enable operators to conduct performance troubleshooting tasks such as data monitoring, root cause analysis, and system diagnostics. The data sources for observability tools consist of three primary components [13]: (i) aggregatable, fragmented *metrics*; (ii) event-driven *logging* with natural semantics; and (iii) request-oriented, workflow-centric *tracing*.

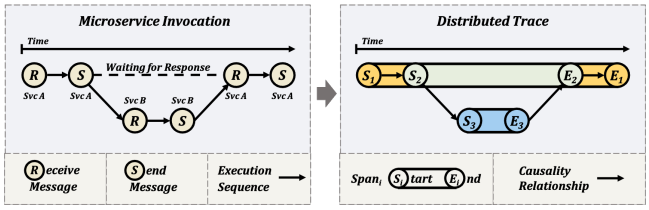


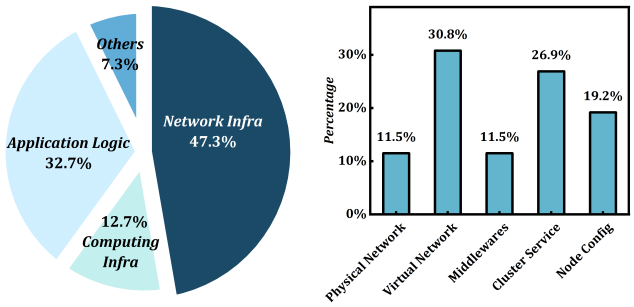
Figure 1: A distributed trace consisting of three spans.

In contrast to the more general concept of tracing, distributed tracing specifically focuses on the diagnosis and troubleshooting of distributed systems. It distinguishes itself from the other two observability pillars by providing an indispensable *end-to-end request-oriented* view of a transaction. The outputs of distributed tracing, specifically referred to as *distributed traces* [62, 64, 91, 121, 141], consist of two types of information: (i) the life cycle (i.e., spans [121]) and correlated metrics within each component; and (ii) the causal relationships and execution sequences between spans. Figure 1

illustrates an example of a distributed trace obtained in a simple microservice scenario. In this example, component A receives a request from a user, begins execution, and then sends a new request to component B while awaiting a response. The execution in component A proceeds once component B has completed its process and sent back a response. Finally, component A returns the execution results to the user through a response. Using the sending and receiving of requests and responses as beginnings and endings, the above procedure will generate a trace with three spans that denote distinct execution phases in each component. The trace provides a clear picture of how component A invokes component B and how long the execution takes in each component.

The primary objective of distributed tracing is to use spans as boundaries to subdivide end-to-end latency into finer units and produce workflow-centric outputs. Nevertheless, most of the current frameworks [6, 62, 141] mainly focus on application-level components and ignore the network infrastructure. In this paper, we refer to the components not detected by the distributed tracing framework and spans that should be but are not covered by distributed traces as *blind spots*. Blind spots in network infrastructure will reduce the number of spans, obscure the execution stage boundary, and lead to ambiguous information in traces.

2.2 New Distributed Tracing Requirements



(a) Network infrastructure is complicated and prone to failure.

(b) The virtual network is the weakest spot in network infrastructure.

Figure 2: Sources of performance anomalies.

2.2.1 Error-Prone Network Infrastructure. Figure 2 depicts a survey result of the source locations of real-world microservice failures in the production environment. The results of this survey were obtained by gathering and analysing the microservice failures reported by DeepFlow's 26 enterprise clients. As shown in Figure 2(a), it is obvious to draw the conclusion that the network infrastructure (47.3%) and the applications themselves (32.7%) are the most frequent sources of performance issues. Failures resulting from computing infrastructure, such as containers, are less frequent (12.7%). In addition, rapid surges in external traffic (7.3%), owing to attacks and other reasons, are also a failure source.

Figure 2(b) examines the composition of network-side performance breakdowns in further detail. The virtual network is most susceptible to performance issues, with 30.8% of failures occurring. Furthermore, the physical network (e.g., physical NIC), network middleware (e.g., message queues [128]), cluster services (e.g., DNS [21], gateway), and node configuration (e.g., Linux firewall settings) can all contribute to anomalies.

The above results demonstrate that the network infrastructure is susceptible to performance problems and should be a top priority for distributed tracing systems. However, current distributed tracing frameworks do not capture network information. When there is a network performance anomaly, users cannot identify the responsible device.

2.2.2 Typical In-Production Microservice Scenarios. Missing network information is merely one of the major problems that existing technologies encounter. To further illustrate the new requirements for distributed tracing, we present four typical *real-world production cases from our enterprise customers* (Table 1).

Scenarios	Convenience	Portability	Stability	Coverage	Correlation
Cross-Dept Debug				✓	✓
Online Game Ops			✓	✓	✓
Infra Services		✓	✓	✓	✓
Agile Dev	✓	✓	✓	✓	✓

Table 1: New requirements for distributed tracing.

Scenario 1 - Cross-department performance debugging is, according to our survey, one of the most common issues experienced by large companies. This time-consuming task requires the collaboration of various departments, including infrastructure teams, developers, and operators. To enable rapid problem location, traces should ensure *coverage* and reduce blind spots on the infrastructure side. Additionally, tracing frameworks should not only extract request information. They have to offer strong *correlation* capabilities to combine the semantics at various levels of the applications, pods, and physical machines.

Scenario 2 - Online game operations are a classic platform hosting scenario [42, 97] that is also highly distinctive among our real-world customer examples. Vendors deliver back-ends and update packages to the platform, which is then responsible for operation and maintenance. Game back-ends are often closed-source for commercial reasons, making it impossible for the developers to manually inject the tracing application programming interfaces (APIs) provided by the operators. Apart from *correlation*, this delivery approach necessitates that the distributed tracing system be able to detect closed-source components for *coverage*. Also, monitoring tools should be *stable* and not affect the performance of online game backends that are sensitive to latency.

Scenario 3 - Developing infrastructure services for use by other departments is one of the scenarios that DeepFlow users are most concerned with. In this situation, distributed tracing frameworks must be capable of widespread adoption. It should, first and foremost, be *stable* and *portable*, running reliably on top of various systems and language runtimes. Second, high *coverage* and strong

correlation are required in large enterprises with complicated infrastructures.

Scenario 4 - Agile microservice development has the strictest requirements for distributed tracing. First, to keep up with the frequent changes in components and service topologies, tracing frameworks must be *user-friendly*. Second, the *portability*, *stability*, and *coverage* are put to the test by the numerous third-party components introduced by microservice developers, including caching systems [94, 113], databases [22, 122], reverse proxies [32], and web frameworks [33, 39]. Lastly, microservices are typically deployed on resource orchestration platforms [37, 40], which increases the challenge of *correlation*.

2.3 Motivating DeepFlow

Unfortunately, modern distributed tracing systems are falling behind the times, especially when it comes to satisfying the aforementioned new requirements. As shown in Table 1, we divide the demands into two categories: out-of-the-box tracing and rapid problem location. Then we classify existing frameworks into two types based on their instrumentation approaches, *intrusive* and *non-intrusive*, and characterize their limitations.

Distributed Tracing		Out-of-the-Box Tracing			Rapid Problem Location	
		Convenience	Portability	Stability	Coverage	Correlation
Intrusive	Manual	○	○	●	◐	○
	Automatic	●	○	●	◐	○
Non-Intrusive	User Space	●	◐	●	◐	○
	Kernel Space	●	◐	○	◐	◐

Table 2: Existing frameworks are incapable of meeting the new demands imposed by microservices. ○, ◐, and ● correspond to unsupported, partially supported, and fully supported, respectively.

Intrusive Distributed Tracing Frameworks. Intrusive frameworks implement distributed tracing by modifying the source code of target components. Depending on whether this modification is performed automatically, intrusive frameworks can be further categorized as manual or automatic. The former [43, 55, 62, 91, 141] requires users to read the relevant component code and instrument it at specific points, whereas the latter [63, 92, 101, 115] employs techniques such as bytecode instrumentation [100] to intercept certain pre-defined functions or methods.

Although intrusive frameworks have extensive tracing capabilities, the information they provide is *insufficient to locate the performance issues*. Spans are restricted inside open-source components, and little is known about closed-source components or the underlying sophisticated network infrastructure. These blind spots waste valuable debugging time in communication among different component developers.

Additionally, intrusive frameworks *create a significant burden on users and framework developers*. On the one hand, instrumentation is often identified as time-consuming and challenging [16, 34, 65, 116, 120]. On the other hand, as illustrated in Figure 3, a substantial amount of effort is still necessary for developers to maintain the respective SDKs for each language and kernel. Moreover, to deploy or update an intrusive distributed tracing framework, users must recompile and redeploy all relevant microservice components.

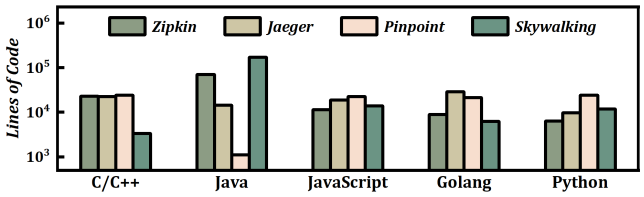


Figure 3: LOCs of distributed tracing SDK repositories.

Non-intrusive Distributed Tracing Frameworks. In order to avoid modifications, non-intrusive tracing is achieved by capturing the external interaction interfaces of microservice components. Existing frameworks [47, 56, 72, 107, 118, 121, 132, 135, 137] employ techniques like instrumenting the remote procedure call (RPC) serialization library, pre-loading libc, and developing kernel modules.

These frameworks, however, are *unable to simultaneously provide rapid problem location and out-of-the-box tracing*. First, instrumentation capabilities and isolation boundaries limit user-space implementations and create blind spots. These blind spots in closed-source software and network infrastructure make it harder to locate problems. Second, current kernel-space frameworks are difficult to deploy and use. Kernel modules are prone to crashing [27, 108] and are difficult to maintain across different kernel versions.

To summarize, existing intrusive or non-intrusive frameworks each have their own benefits, but they cannot simultaneously provide convenience, portability, stability, coverage, and correlation.

2.3.1 Opportunities by eBPF. As a kernel-supported mechanism, eBPF (extended Berkeley Packet Filter) [28, 90, 93, 99] presents unprecedented opportunities for simultaneously achieving the aforementioned objectives. It functions by providing a virtual machine in the kernel that enables the execution of small pieces of code, known as *BPF programs*, written by users and loaded by the BPF program loader [23, 82]. With the help of BTF (BPF Type Format) [83], these programs are portable across different kernel versions. With kernel privileges and visibility, these programs can be attached to different types of *hooks* (i.e., trigger functions) [66], allowing them to be executed in response to various events such as a system call or a user space function call. Attachment, triggering, execution, and detachment of BPF programs are non-intrusive and in-flight, which means that no modification, recompilation, or redeployment of the monitored application is required. In addition, these programs are validated by the eBPF verifier [67] prior to execution, allowing BPF programs to access and manipulate kernel data structures without crashing the kernel.

3 THE DEEPFLOW SYSTEM

In this section, we first provide the design goals of DeepFlow and its overall architecture (Section 3.1). Then, we elaborate on the three design aspects of DeepFlow: network-centric tracing plane (Section 3.2), implicit context propagation (Section 3.3), and tag-based correlation (Section 3.4).

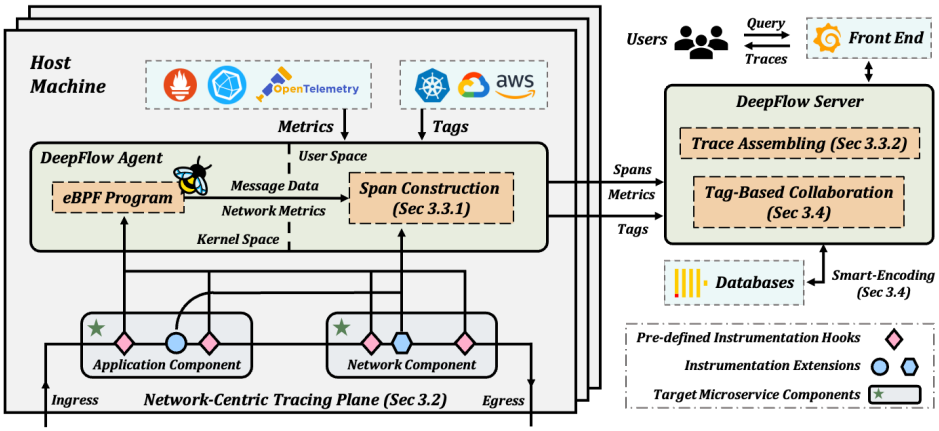


Figure 4: Architecture overview of DeepFlow.

Ingress System Calls
recvmsg
recvmmsg
readv
read
recvfrom

Egress System Call
sendmsg
sendmmsg
writev
write
sendto

Table 3: Network-centric instrumented ABIs of DeepFlow.

3.1 DeepFlow Overview

The limitations of existing tools discussed in Section 2 lead to the following high-level design goals for DeepFlow:

- **Goal 1: Instrumentation Convenience.** First, users are not required to determine where to conduct the instrumentation. Second, users can instrument target components without directly modifying their code.
- **Goal 2: Maintenance Simplicity.** First, developers do not need to maintain multiple implementations or SDKs of the same framework for different languages or applications. Second, developers are not required to offer several implementations of the same interface dependent on kernel changes. Finally, DeepFlow should be stable and secure.
- **Goal 3: High Accuracy and Coverage.** DeepFlow must ensure thorough and accurate tracing. It should *avoid blind spots* in the closed-source components, cloud infrastructure, and underlying network.
- **Goal 4: Cross-Layer and Cross-Component Correlation.** DeepFlow must have effective cross-layer information integration capabilities to build *correlations*. On the other hand, it must be able to *correlate the data properties* provided by different components for additional analysis.
- **Goal 5: High Performance.** DeepFlow must have *negligible instrumentation, transmission, and processing overhead* in order to provide real-time distributed tracing without degrading application performance.

Considering these design goals, we present the overall architecture of our system in Figure 4. DeepFlow consists of two high-level components: **Agent** and **Server**. An Agent is deployed in each container node, virtual machine, or physical machine to capture trace data using pre-defined eBPF instrumentation hooks and instrumentation extensions. In addition, the Agent is responsible for integrating metrics and tags from third-party frameworks or cloud platforms and transmitting them to the Server. The DeepFlow Server is a cluster-level process. It is responsible for storing spans in the database and assembling them into traces when users query.

3.2 Network-Centric Tracing Plane

Instrumentation is the foundation of the entire distributed tracing system. With a precise microservice execution abstraction and automatic non-intrusive instrumentation, we create a high-performance, independent, network-centric tracing plane in the kernel space.

3.2.1 Microservice Execution Abstraction. To facilitate instrumentation and maintenance, DeepFlow extends interface-level distributed tracing and selects a collection of system call application binary interfaces (ABIs) as the basic instrumentation points. Choosing system call ABIs over library functions enables DeepFlow to have the highest degree of development generality, i.e., only one single framework is required to build support for various languages and kernel versions (**Goal 2**). In addition, users can gather primary trace data through these pre-defined interfaces without having to adapt to different microservice components (**Goal 1**). Nevertheless, regardless of the complexity of microservice components' logic, DeepFlow must ensure that these pre-defined instrumentation points do not miss critical trace spans (**Goal 3**) when compared to manual instrumentation. Meanwhile, DeepFlow should achieve high performance (**Goal 5**).

★ Design 1: A narrow-waist instrumentation model with two sets of functions: ingress-egress and enter-exit. Regardless of diversity and complexity, the execution of a microservice component is triggered by its communication. Therefore, DeepFlow ignores the control flow information and bases the trace collection procedure on the ingress-egress behaviors.

As revealed in Table 3, DeepFlow instruments ten system call ABIs and classifies them as **ingress or egress**. These ABIs are capable of covering all data communication scenarios (blocking or non-blocking, synchronous or asynchronous) between microservice components while remaining independent of application logic and communication protocols. Notably, neither ingress nor egress corresponds directly to a request or a response. For a client-side component, an egress message corresponds to a request, while an ingress message relates to a response. On the other hand, a server-side component operates in the exact opposite way. The request-response inference is introduced in Section 3.3.

DeepFlow stores information about each ingress or egress call as it **enters or exits** the kernel. Four categories of information are recorded for further processing in the user space: (i) Program information, including process ID, thread ID, coroutine ID, program name, etc.; (ii) Network information, including the DeepFlow-assigned global unique socket ID, five-tuple, the TCP sequence, etc.; (iii) Tracing information, including data capture timestamp, ingress/egress direction, etc.; and (iv) System call information, such as the total length of read/write data, payload to be transferred to the DeepFlow agent, and so on. With the information above, DeepFlow can provide a universal abstraction for the tracing plane and accomplish the aforementioned objectives.

Instrumentation Extensions. DeepFlow provides three types of instrumentation extensions to respond adaptably to varied conditions. First, DeepFlow integrates network data from the classic Berkeley Packet Filter (cBPF [69]) and AF_PACKETS [84] to derive NIC-side information. Second, the use of `uprobe` [70] enables the interception of code in any location in the user space. This grants DeepFlow easy access to important information, such as the original payload prior to TLS encryption. Lastly, DeepFlow enables the incorporation of traces from third-party distributed tracing frameworks (e.g., OpenTelemetry [103]). This enables DeepFlow to maintain overall compatibility while granting users the greatest instrumentation flexibility.

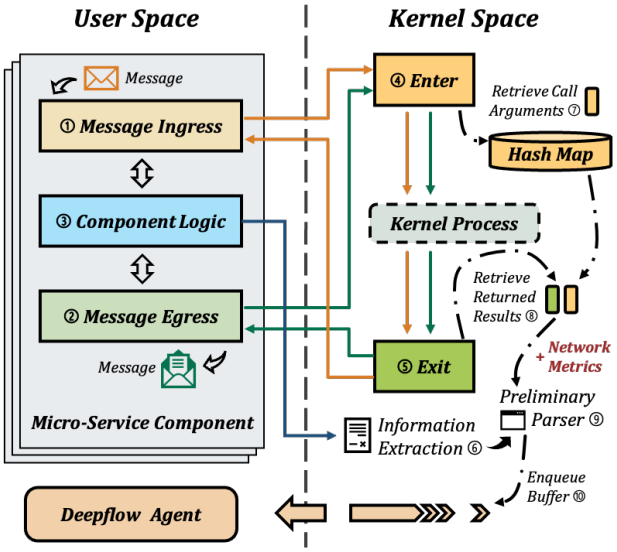


Figure 5: DeepFlow collects message data and network metrics in the kernel space via pre-defined hooks.

3.2.2 Automatic Non-Intrusive Instrumentation. The network-oriented instrumentation model is merely the first step toward achieving our goals. DeepFlow seeks to make further efforts by *automating instrumentation*, relieving users of the time-consuming task of modifying code (**Goal 1**). The framework should also be *non-intrusive* to capture trace data from closed-source applications and network infrastructure (**Goal 3**). Finally, network metrics from different layers should be captured *in zero code* (**Goal 4**).

★ **Design 2: In-kernel hook-based instrumentation.** In accordance with the pre-defined instrumentation model, DeepFlow automatically *registers hooks* to collect trace data, as shown in Figure 5. For message ingress (①) or egress (②), the corresponding system call will trigger the registered kprobe or tracepoint [68, 71] hooks when it enters (④) and exits (⑤) the kernel. The tracing process will retrieve the arguments (⑦), wait for the kernel to complete its processing, and then retrieve the returned results (⑧). The preliminary parser (⑨) will integrate and enqueue the primary data into the buffer (⑩), which will subsequently be transmitted to the user space for further processing. Additionally, DeepFlow utilizes uprobes and uretprobes [70] to extract information (⑥) from extended instrumentation points within the component's logic (③). All of the operations are executed automatically. Users can perform distributed tracing *in zero code*.

Instead of collecting traces from the GNU C library [135] or serialization libraries [137], DeepFlow runs *in the kernel space based on eBPF*. The kernel privileges permit DeepFlow to access information (e.g., message content, TCP sequence, socket flags, etc.) at multiple network layers of different components. Changing a single library, on the other hand, makes it hard to get this information because of privilege limits and isolation boundaries. Moreover, the kernel provides unified structured data across all types of components, which means that the data produced by DeepFlow has a consistent structure. Because of this, it is easier and more efficient to combine, associate, and store traces.

DeepFlow implements the trace data collection module based on eBPF [66, 69]. Since eBPF programs can directly monitor any on-the-fly programs, DeepFlow can be deployed *in zero code* at any time, and users no longer need to restart active online microservices. Thanks to the eBPF verifier [67], DeepFlow will not result in kernel crashes, which is quite a common issue in kernel modules [27, 108].

3.3 Implicit Context Propagation

DeepFlow's network-centric tracing plane enables the recording of extensive monitoring information and eliminates blind spots in network infrastructure. However, as stated in Section 2, achieving comprehensive and efficient troubleshooting necessitates more than independent, fragmented, and primitive information. In this section, we demonstrate how DeepFlow constructs the spans and traces specified in Section 2.1 using the data collected in Section 3.2.

Traditional distributed tracing frameworks modify the source code [64] or serialization libraries [121, 137] to explicitly insert *context information* into the headers [105, 130] or payloads [45] of messages. Per-message unique identifiers generated by frameworks [62, 91, 141] (e.g., trace IDs and span IDs) are encoded in context information and propagated with the transmission of messages. Using these unique identifiers, it is straightforward to aggregate data collected during the sending and receiving of requests and responses to produce spans and the trace of the same message. Despite its seeming simplicity, this technique, also known as *explicit context propagation*, requires that the users be familiar with the core logic of the microservice components being monitored. Users must either apply the semi-automatic APIs [125] provided by the tracing framework or manually define the start and end of spans [20]. This intrusive nature of explicit context propagation

severely restricts instrumentation convenience (**Goal 1**) and maintenance simplicity (**Goal 2**). Additionally, if the application layer protocol prohibits header modification, the framework would risk corrupting messages by altering their payloads.

★ Design 3: Implicit context propagation with hierarchical aggregation. DeepFlow suggests implicit context propagation as a solution to the non-intrusive construction of spans and traces, where context information is no longer transmitted along with messages. Our key insight is that *the information required for context propagation is already contained in network-related data*. By maximizing the utilization of data from each network layer, DeepFlow does not need to explicitly include context information within the message. In general, DeepFlow combines independent, fragmented, and primitive measurements into request-oriented traces containing precise causal correlations through the following two phases: (i) constructing spans from the instrumentation data and (ii) assembling traces from spans using implicit causal relationships.

3.3.1 Construction of a Single Span. Similar to the majority of distributed tracing frameworks [62, 121, 141], DeepFlow generates spans that always begin with a request and end with a response (Figure 1). Execution models with one-to-many, many-to-one, and many-to-many communication patterns *in a single connection*, such as streaming and message subscription, are beyond the scope of this paper. In addition, we presume that every microservice component should return a response in response to a request. This constraint is regarded reasonable due to the fact that if a microservice component fails to return a response, the component that invoked it will be unable to determine whether or not the requested action was executed successfully. DeepFlow considers any missing responses as outcomes resulting from unexpected execution terminations.

Figure 6 depicts the three phases of our span construction procedure: message data production, message type inference, and session aggregation. Firstly, DeepFlow associates information captured during the enter and exit of the same system call by using *process IDs and thread IDs* (⑦ and ⑧ of Figure 5). The association is predicated on the fact that the kernel can simultaneously handle only one selected system call (listed in Table 3) for a given (*Process_ID*, *Thread_ID*). For languages such as Golang [48], DeepFlow monitors the creation of coroutines to save the parent-child coroutine relationship in a *pseudo-thread* structure and performs similar operations. DeepFlow temporarily saves the enter parameters in a hash map, retrieves them at exit time, and combines them with the exit parameters. The combined data is referred to as *message data*, and its type is classified as ingress or egress based on the type of system call captured. To decrease the amount of data transferred, we only process the first system call for a message, not the subsequent ones that are used for further data transfers.

In the second phase, DeepFlow performs message protocol inference and parses messages with their original semantics using relevant network information. Specifically, after the message data has been transferred to the user space, the DeepFlow Agent iterates through *the common protocol specifications* [35, 36, 57, 59, 60, 106, 114] and *the optional user-supplied protocol specifications*, executing a one-time protocol inference for each newly established connection. Then, DeepFlow parses the payload to determine the request/response type of the message. Once type inference has been performed, the message data is prepared for use as either the start or end of spans (Figure 1). Although deep packet inspection is unavoidable, DeepFlow as an open source project, typically only extracts information from the packet headers and does not examine the sensitive user data primarily located in the payload.

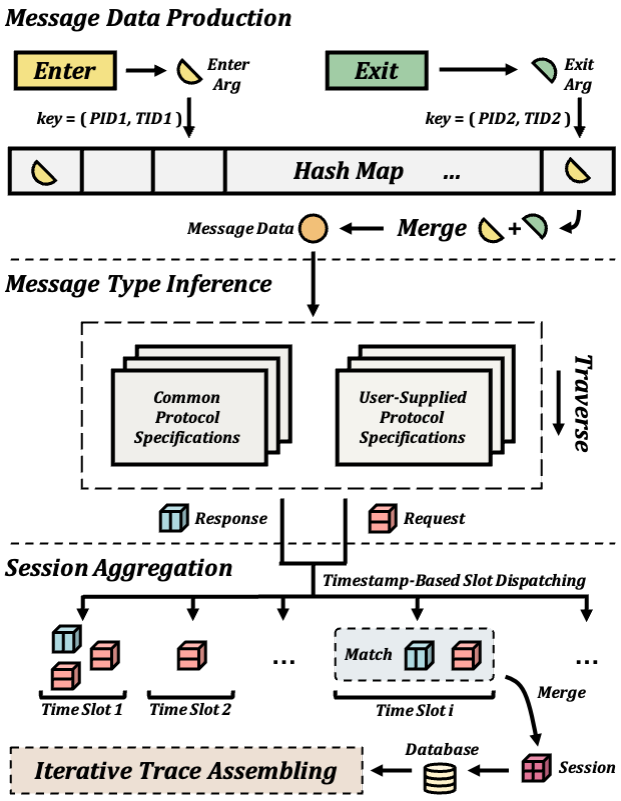
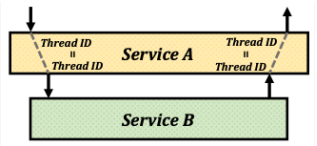


Figure 6: Three phases of span construction procedure.

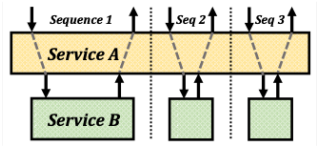
Last but not least, DeepFlow will try to aggregate one request and one response from the same flow into *sessions*. A session can be used to represent a span (Figure 1), where the request of the session is the start of the span and the response is the end. To accomplish session aggregation, DeepFlow must take into account both *pipeline* and *parallel* application layer protocols. Pipeline protocols send and receive messages in a pipeline pattern, where the order of requests and responses for the same connection remains constant. Therefore, DeepFlow just needs to match the order between responses and requests *in the same flow*. Parallel protocols, on the other hand, multiplex messages on the same connection. To ensure precise correspondence, DeepFlow makes use of the embedded distinguishing attributes in the message as originally defined by the protocol (e.g., IDs in DNS headers [58] and stream identifiers in HTTP/2 headers [60]). To enable effective merging and address the message disorder problem introduced by multiple CPU cores, DeepFlow implements a time window array and stores messages according to their timestamps. When aggregating, only messages in the same time slot or next to it will be queried. Based on our production experience, DeepFlow presently sets the duration of

each time slot to 60 seconds. Messages received outside of the time period are uploaded to the DeepFlow Server, where they can be aggregated again using the same technique.

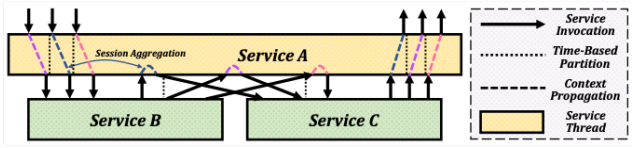
3.3.2 Trace Assembling. In order to generate complete traces, it is necessary to execute a final step on spans: trace assembling. Taking into account the need for instrumentation convenience (**Goal 1**), it is necessary to infer the causal relationship between intra-component and inter-component in a non-intrusive manner. Using Figure 1 as an illustration, the solid arrow between spans 1 and 2 represents intra-component causality, whereas the solid arrow between spans 2 and 3 represents inter-component causality. Existing research suggests using logs or system call stacks to non-intrusively assemble traces [72, 136, 139, 140]. However, fuzziness in log semantics and call stack boundaries can affect the accuracy and coverage of tracing (**Goal 3**). To address this challenge, DeepFlow determines the relationships between spans by utilizing information across different network layers saved during instrumentation. DeepFlow takes the spans that users query as starting points and merges the associated spans. Meanwhile, cross-layer correlation is supported by intra- and inter-component association as well as third-party span integration (**Goal 4**).



(a) Implicit context propagation via thread IDs.



(b) Using time sequences to partition thread-reusing spans.



(c) Association for multiple requests or responses.

Figure 7: Intra-component causal association.

Intra-Component Association. Typically, a microservice component may need to invoke other components after receiving a request, and vice versa. Therefore, DeepFlow must retain its relationships within a thread without being aware of the underlying logic. DeepFlow typically identifies intra-component causal relationships between spans using thread IDs, time information, and insight of scheduling. A same `systrace_id` is assigned to both of the spans that possess a causal association, which serves as a global unique identifier. Instead of injecting the `systrace_id` into packets, it is appended as an attribute to the message data of spans, which satisfies our non-intrusiveness requirements.

First, DeepFlow associates spans within the same thread using thread IDs (Figure 7(a)). Due to the *1:1 relationship between the kernel thread and user thread* in the thread-based microservice components, this association can be performed in the kernel. For coroutines in Golang, DeepFlow can also conduct association by tracking the invocation relationships between coroutines during execution. Second, when threads are reused, the trace will be partitioned based on the time sequence (Figure 7(b)). Finally, DeepFlow needs to handle multiple requests or responses (Figure 7(c)). The key takeaway is that, for a single thread, *computing does not (and should not) yield to scheduling, whereas network communication does.* Consequently, we label two consecutive messages of different types and from different sockets with the same `systrace_id`.

Cross-thread intra-component association is a notorious problem due to the fact that users can pass packets between threads in ways that are difficult to capture, such as memory copies and parameter transfers. DeepFlow employs the fact that the microservice component itself must maintain the intra-component association to circumvent this issue. For microservice components such as HAProxy [51], Envoy [110], and Nginx [32], DeepFlow utilizes its original capabilities to generate X-Request-IDs [50, 102, 111] for messages, preserving the association of spans across threads.

Inter-Component Association. DeepFlow does not insert context information into messages, making it difficult to correlate across network infrastructures, processes, and machines. Additionally, device-level spans gathered by cBPF [69] and AF_PACKET [84] must be associated with relevant spans. Fortunately, since network transmissions (Layer 2/3/4 forwarding) do not change the TCP sequence, DeepFlow leverages this for the inter-component association. During the instrumentation phase (Section 3.2), we calculate and record the TCP sequence for each message in the kernel. It is then used to differentiate and maintain the inter-component association of spans within the same flow.

Third-Party Span Integration. DeepFlow can incorporate spans generated from user-defined distributed tracing frameworks. For instance, by parsing the reserved header fields [104, 105, 129] used by OpenTelemetry in the message collected by DeepFlow, the context information of OpenTelemetry can be extracted.

Bottom-Up Trace Assembling. In the preceding phase, DeepFlow does not directly generate traces, but rather injects associations as tags into the message data and sessions. At present, DeepFlow does not consider the scenario where multiple messages are cached within a single data structure, thereby significantly simplifying the mapping of system calls to message data. Based on this assumption, a thread can process only one message simultaneously. Hence, we regard consecutive ingresses or egresses as system calls that are activated by the same message. This assumption indeed makes DeepFlow incapable of managing scenarios such as message queues. We plan to tackle this problem in future work.

Algorithm 1 demonstrates the final step of trace assembling: iteratively aggregating spans using previously injected single-threaded intra-component information (`systrace_ids` and pseudo-thread IDs), cross-threaded intra-component information (X-Request-IDs), inter-component information (TCP sequences), and third-party information (trace IDs) in order to generate traces. Users can select spans that they are interested in, such as time-consuming invocations, as the starting points of the assembly procedure (Line 2). In the first part of the algorithm, DeepFlow searches the database using the user-specified iteration times (the default is 30). In each iteration, we add to the span set any new spans that share the same `systrace_id` (Line 6), pseudo-thread ID (Line 7), X-Request-ID

(Line 8), TCP sequence (Line 9), and trace ID (Line 10) as the current spans. The search is terminated if the number of related spans does not increase between two consecutive searches (Lines 13-14).

In the second phase of the algorithm, we iterate over the span set and set the parent spans. The determination of parent spans is also based on the aforementioned intra- and inter-component associations, but with stricter conditions. We set 16 rules based on the collection location (server or client), start time and finish time, span type, and message type (Line 20). For instance, if an eBPF span collected on the client side has the same TCP sequence as an eBPF span collected on the server side, the parent of the client side span is set to the server side span. Finally, we sort the span set by time and parent relationship (Line 25) to generate a display-friendly trace and transmit it to the front end.

Algorithm 1 Iterative Trace Assembling Algorithm

Input: *start_span* - User-Chosen Span, I - Iteration Times
Output: T - Assembled Trace

```
1:  // Iterative Span Search
2:  span_set ← Set(start_span)
3:  filter ← {id == start_span.id}
4:  for iter ← 1 to I do
5:    for s ← span_set do
6:      filter ← filter ∪ {systrace_id == s.systrace_id}
7:      filter ← filter ∪ {pseudo_th_id == s.pseudo_th_id}
8:      filter ← filter ∪ {x_req_id == s.x_req_id}
9:      filter ← filter ∪ {tcp_seq == s.tcp_seq}
10:     filter ← filter ∪ {trace_id == s.trace_id}
11:   end for
12:   span_set = search_database(filter)
13:   if span_set.not_update then
14:     Break
15:   end if
16: end for
17: // Set Parent for Each Span
18: for s ← span_set do
19:   for r_s ← s.related_spans() do
20:     if related_s.is_parent(s) then
21:       s.set_parent(related_s)
22:     end if
23:   end for
24: end for
25: T ← span_set.sort()
26: Return T
```

3.4 Tag-Based Correlation

To achieve cross-component correlation *in zero code*, DeepFlow injects uniform tags into the spans (**Goal 4**). We enable the injection of Kubernetes resource tags [73] (e.g., node, service, pod, etc.), self-defined labels [73] (e.g., version, commit-ID, etc.), and cloud resource tags [9, 30] (e.g., region, availability zone, VPC, etc.). Users can use these tags to immediately determine the locations of the problems, such as in which pod the invocations are time-consuming. These tags also connect tracing and metrics, allowing DeepFlow to integrate them. When querying traces, users can simultaneously view the related metrics data generated from frameworks like Prometheus [112]. Note that during tag-based correlation, we do not modify the original packets, and tag injection is performed off-path on the data collected by DeepFlow.

However, in a typical production environment, up to 100 tags might be related to a single trace. To minimize the overhead, we introduce a technique called smart-encoding.

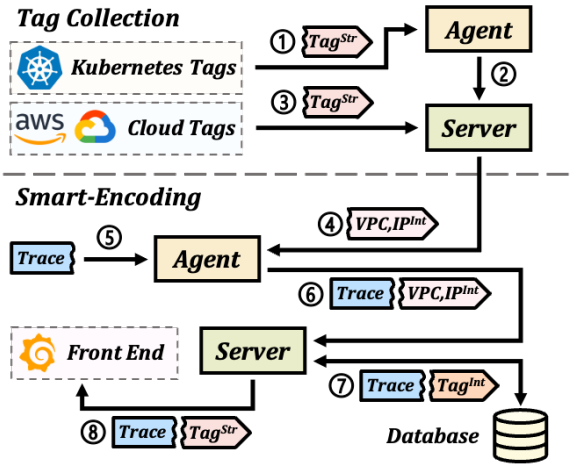


Figure 8: Integrate resource tags with smart-encoding.

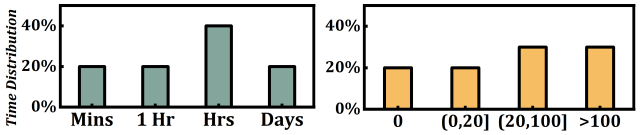
★ **Design 4: Smart-encoding based on phased tag injection.** As demonstrated in Figure 8, DeepFlow divides the tag injection process into two phases: *tag collection* and *smart-encoding*. During the tag collection phase, DeepFlow Agents inside the cluster will collect Kubernetes tags (①) and send them to the Server (②), while the cloud resource tags are gathered directly by the Server (③). In the smart-encoding phase, DeepFlow only injects virtual private cloud (VPC) tags and IP tags in `Int` format into traces (④-⑥). The Server then injects the resource tags in `Int` format into the traces based on the VPC/IP tags and stores them in the database (⑦). At query time, DeepFlow Server determines the relationship between self-defined tags and resource tags, injects self-defined tags into traces, and then uploads the traces with all the tags to the front end (⑧). By partitioning the tag injection phases, DeepFlow reduces the calculation, transmission, and storage overhead.

4 DEEPFLOW IN PRODUCTION

DeepFlow has served dozens of companies in the years since it went into production. Simultaneously, we will make DeepFlow publicly available and aim to develop a substantial open-source community. In this section, we show the performance of DeepFlow in the production environment.

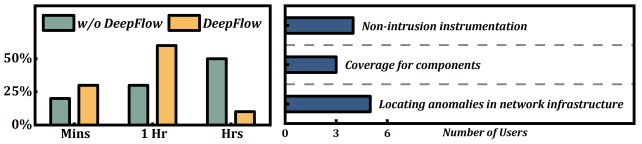
Figure 9 demonstrates the results of the target interviews with ten commercial customers[1]. *All of the participants are Fortune Global 500 companies and have tens of thousands of employees.* Without using DeepFlow, 60% of the users must spend hours or days instrumenting a single component. For 30% of the customers, the burden of modifying hundreds of lines of code per component is

[1]The original data from the questionnaire is included in Appendix C.



(a) Time spent on instrumentation. (b) Lines of code that need to be modified for instrumentation.

Figure 9: Instrumentation efforts without DeepFlow.



(a) Time spent locating the performance problems. (b) Primary advantages of utilizing DeepFlow as reported by users.

Figure 10: DeepFlow's contribution in production cases.

overwhelming. DeepFlow, on the other hand, eliminates instrumentation efforts through automatic, non-intrusive instrumentation. Without manual instrumentation, developers can retrieve traces by deploying DeepFlow in the cluster.

Figure 10(a) emphasizes that DeepFlow can indeed shorten the time required for locating performance issues. As a result of the coverage and correlation abilities, users do not have to manually capture network metrics or check the status of network infrastructure. With DeepFlow, operators can pinpoint and identify the root cause in a fraction of the time it would take otherwise. Our user questionnaire confirms this (Figure 10(b)). Five out of ten consumers acknowledge that network coverage is one of the reasons that motivate them to employ DeepFlow. Four users find the non-intrusive instrumentation helpful. Three users believe the tracing of closed-source components to be one of DeepFlow's benefits.

4.1 Representative Real-World Examples

Next, we demonstrate the *out-of-the-box tracing* and *rapid problem location* capabilities of our system. We provide three representative real-world examples that highlight the practical use of DeepFlow's **usability**, **network-side coverage**, and **correlation capabilities**, respectively. In Appendix A, we present DeepFlow's comprehensive traces from end-hosts to gateways as requests traverse the data center. DeepFlow currently supports rapid deployment in a single or across multiple Kubernetes clusters [37] via Helm [53], without requiring administrators to install any kernel modules or libraries. DeepFlow can be operated continuously to monitor a microservice over an extended period of time, or on demand in the event of a performance anomaly, as shown in the examples below.

4.1.1 Performance Debugging During Execution. Nginx [32] is a typical reverse proxy server that plays a critical role in data forwarding. In this case, the client encountered a timeout on a specific endpoint, which turned out to be an error in Nginx. Although the source of the timeout seems obvious in retrospect, the client spent **an entire day** unable to discover the issue using existing tools. In a rush, the developers did not instrument every component, and the invocation path was full of blind spots.

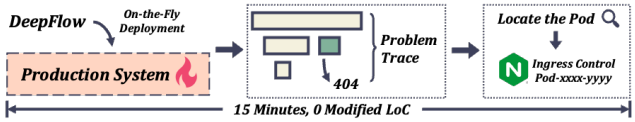


Figure 11: The operators employed DeepFlow in an on-the-fly production environment to pinpoint the failure source: a pod hosting Nginx Ingress Control.

Therefore, the user leverages DeepFlow to *locate the anomalies during the execution of the online system* (Figure 11). **Without modifying a single line of code**, operators deploy DeepFlow while the service is active and obtain hop-by-hop network data in the traces collected. Within **15 minutes**, the root cause is identified: one of the pods hosting Nginx Ingress Control in the cluster has an error, thus returning a 404 status code.

4.1.2 Accurate Diagnosis of Network Infrastructure Anomalies. In this example, newly installed Pods by an e-commerce service have a high risk of network inaccessibility to the gateway or other business services. Users have to wait between 20 and 120 minutes for communication to resume. This has a significant influence on the robustness of the service. Operators used various tools for **several months** to determine that an extra ARP request had been generated during the connection. Unfortunately, they were unable to determine where the extra ARP request originated.

DeepFlow resolves this issue by providing *network coverage*, which is lacking in other distributed tracing frameworks. Using DeepFlow, operators traverse the traces and inspect the number and status of ARP requests at each network infrastructure. After ruling out problems in the containers, virtual machines, and virtual switches, we discover that the redundant ARP requests are generated by a malfunctioning physical NIC.

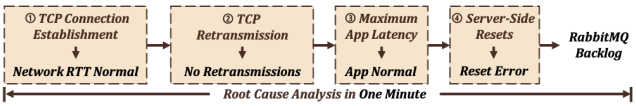
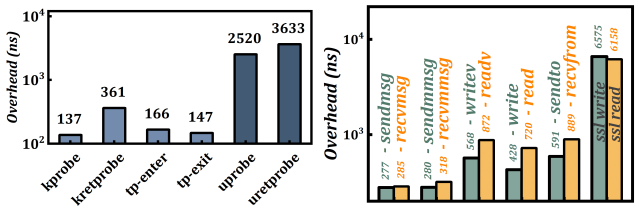


Figure 12: Locating the performance anomaly within one minute using DeepFlow.

4.1.3 Cooperative Debugging Based on Network Metrics and Distributed Traces. A customer reported that their online service system has experienced frequent service latency increases and connection terminations. Operators spent **six hours** debugging using existing observability tools. But they only identified the application-level spans that were affected.

Cross-layer correlation distinguishes DeepFlow from other tracing frameworks. Figure 12 shows the fault-locating procedure. With metric-by-metric analysis of some specific traces, users found in **one minute** that the queue backlog of RabbitMQ [128] was causing the TCP connection resets, which led to the aforementioned network problems. Without the capacity to correlate, network analyzers cannot extract the related information from the vast volume of data, and distributed tracing can only obtain the affected application-level spans.



(a) Per-event eBPF instrumentation overhead. **tp** stands for tracepoint.

(b) Overhead of each instrumentation points in DeepFlow.

Figure 13: Instrumentation overhead of DeepFlow.

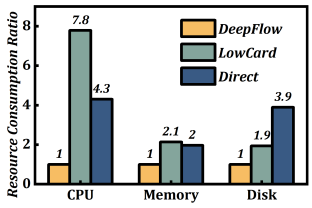


Figure 14: Trace storage resource consumption.

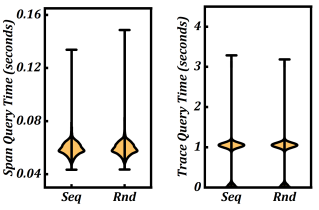


Figure 15: User query delay of spans and traces.

5 TESTBED EVALUATION

In this section, we thoroughly evaluate the performance of DeepFlow and answer the following questions:

- Is the trace collection overhead of DeepFlow negligible? (Section 5.1)
- How many resources does DeepFlow's smart-encoding save when storing traces? (Section 5.2)
- How long does the trace assembling procedure take for a user to perform a high-level query? (Section 5.3)
- What is the end-to-end performance of DeepFlow? (Section 5.4)

We conduct the experiments using a three-node Kubernetes [37] cluster testbed (version 1.24) with standard configurations [109]. The cluster consists of three identical servers with Intel Xeon E5-2620 v3 CPUs (2.40 GHz, 12 physical cores) and 128 GB of total RAM. Ubuntu 20.04 is installed on the server, along with kernel version 5.4.0. To avoid testing errors, we run each test several times and select the median.

5.1 Trace Collection Overhead

In this experiment, we measure the trace collection overhead of DeepFlow Agent. We begin by deploying an empty eBPF program to get the theoretical minimum system overhead. Then, we repeatedly invoke 100,000 system calls and calculate the average execution time before and after deploying the DeepFlow Agent[2].

Figure 13 shows that extra latency ranging from 277 *ns* to 889 *ns* is introduced to pre-defined ABIs. Notably, an ABI will trigger both enter and exit hooks, and DeepFlow only adds a latency of no more than 588 *ns* to each system call in addition to the inherent overhead. This overhead is negligible for I/O ABIs that are particularly time-consuming. Extension hooks such as `ssl_read`, `uprobe`, and `uretprobe` themselves incur a latency of 6153 *ns*. In comparison, DeepFlow's additional latency is maintained below 423 *ns*.

5.2 Effectiveness of Smart-Encoding

Next, we quantify the resources that the smart-encoding method saves during trace storage. It is compared to both direct storing and low-cardinality [19] encoding techniques. For each test, 10^7 synthetic traces are inserted into the database [18] at a rate of $2*10^5$ rows per second. We use `pidstat` [85] to determine the resources utilized during the storage of traces and establish DeepFlow as the baseline. The results are shown in Figure 14.

Smart-encoding enables DeepFlow to utilize only 0.11 CPU cores, 1.39% of the host RAM (about 800 MB), and approximately 1.4 GB of disk space during the storage procedure. Direct insertion requires 4.31 times more CPU, 1.97 times more memory, and 3.9 times more disk space than DeepFlow. This is due to the fact that storing a tag as a string requires more bytes (one char per digit) and thus more calculation and hardware resources than converting it to an integer beforehand. Compared to direct insertion, low-cardinality provides a more efficient encoding method. However, it still requires 1.94 times more storage resources than DeepFlow. In the meantime, low-cardinality uses 2.14 times as much memory and 7.79 times as much CPU as DeepFlow. By smart-encoding, DeepFlow considerably decreases the overhead associated with trace storage, ensuring that the back end does not consume an excessive amount of cluster resources.

5.3 Query Delay

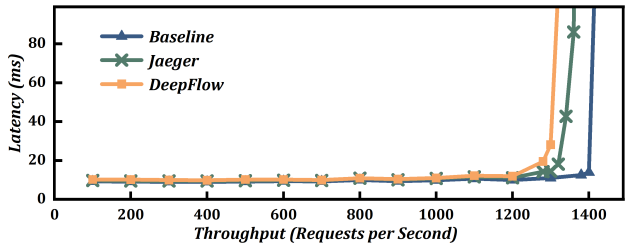
We simulate the user's query behavior to evaluate DeepFlow's back-end capabilities. First, we use load generators to create sufficient spans and traces. Both the span list query test and the trace query test are then conducted twice: once sequentially and once randomly. As the purpose of this experiment is to evaluate the performance of the procedure for trace assembly, user queries are generated via serial calls during testing. The time range for span list queries is set to 15 minutes. Figure 15 illustrates the results. DeepFlow can query a single trace in about 1 second and search a 15-minute span list in approximately 0.06 seconds.

5.4 End-to-End Performance

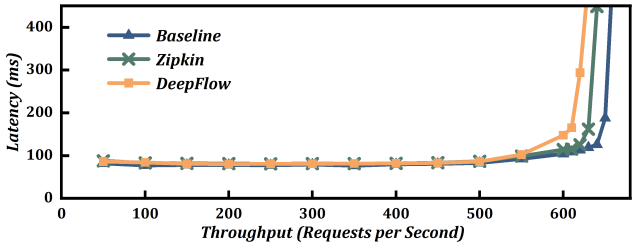
The objective of our end-to-end test is to evaluate the performance impact of DeepFlow on real-world microservices. We choose the Spring Boot demo [12] and the Istio Bookinfo application [61] as the target services. First, we deploy these microservices without tracing tools and record the relationship between throughput and latency as the baseline. Then, we reevaluate their performance after installing Zipkin [141], Jaeger [62], or DeepFlow. The results are shown in Figure 16.

The throughput of the Spring Boot demo without deploying tracing tools is approximately 1,420 requests per second (RPS). After deploying Jaeger and DeepFlow, the throughput decreases to 1,360 RPS and 1,320 RPS, with 4% and 7% overhead. For the Istio Bookinfo demo, Zipkin and DeepFlow reduce the throughput from 670 RPS to 650 RPS (3%) and 640 RPS (4.5%), respectively. Note that, Jaeger only constructs 4 spans for a single trace, while Zipkin

[2]We also measure the performance impact of DeepFlow Agent on the monitored components (see Appendix B).



(a) End-to-end performance evaluation of Spring Boot demo.



(b) End-to-end performance evaluation of Istio bookinfo application.

Figure 16: End-to-end performance evaluation.

produces 6. In comparison, DeepFlow creates 18 and 38 spans per trace for these two applications. The performance of DeepFlow is just marginally inferior to the other tracing tools, with an overhead of no more than 7% but significantly more spans per trace.

6 RELATED WORK

Observability Tools. Classic works, such as Magpie [10], Whodunit [14], X-trace [34], Dapper [121], and Pinpoint [101], laid the foundation for observability. Recent studies take steps further as distributed systems evolve. Pivot Tracing [92] and Panorama [56] emphasize breaking application boundaries by piggybacking metrics along the request flow and exchanging logs from caller to callee, respectively. [91] decouples intrusive context propagation and cross-cutting tool logic. OpenTelemetry [103] standardizes data-collection APIs, instrumentation libraries, and semantic conventions. Canopy [64], addresses the granularity mismatch between operator analysis and raw traces, presents a general event-based tracing model, and supports deep customization for users. As the volume of monitored data explodes, a large body of work focuses on data filtering [7, 26, 77, 78, 88, 126] and overhead reduction [75, 119, 134]. Some other research performs data analysis in areas including performance profiling [54, 55, 95], performance diagnosis [96, 117], failure diagnosis [79, 107], and root cause analysis [132, 138].

Automatic Trace Collection. Existing work contributes in three ways to the development of a user-friendly automatic tracing framework. The first category utilizes automatic code insertion [56, 63, 81, 92, 115, 131]. For example, [131] adds a parameter in each function to propagate global context in Golang. Domino [81] interposes callback registration in JavaScript for event chain construction. The second category leverages existing logs to track causal relationships. Lprof [140] statically analyzes the application bytecode to aid runtime log-based flow construction. CloudSeer [136] builds an automaton for each task by extracting sequential features from logs. Stitch [139] generates a system stack structure (S^3) graph using the identifiers of objects in logs. TS [15] focuses on online real-time log sessionization with high throughput and low overhead. The third category employs kernel-level tools. PreciseTracer [118] builds a `LOG_TRACE` kernel module; Sieve [126] employs sysdig [124]; [47] uses LTTng [87]; and IntroPerf [72] utilizes ETW [98]. However, none of them can simultaneously achieve out-of-the-box, stable, and precise trace collection as provided by DeepFlow.

eBPF. eBPF has been applied in areas including network optimization [1, 127], network virtualization [3, 11], and network security [24, 25]. For instance, InKeV [1] enables the in-kernel programmability of virtualized network functions. ExtFUSE [11] boosts user file system performance by registering request handlers into the kernel. FineLame [24] tracks resource utilization via eBPF for asymmetric Denial-of-Service attack detection.

Recent research has made some attempts to improve observability with eBPF [31, 41, 80, 86, 123]. None of them, however, handles the intricate communication interactions in microservice scenarios. VNetTracer [123] aggregates metrics collected via eBPF and profiles network performance in complex virtualized systems. CaT [31] employs eBPF to analyze content similarity and focuses on contexts such as big data analytics with basic communication models. Pixie [74] and Cilium Hubble [17] are two important works that try to solve problems with microservice observability. However, up until the point of writing this paper, they have not been able to achieve end-to-end distributed tracing. In these frameworks, L4 and L7 metrics, requests, and responses are extracted individually.

7 CONCLUSION

In this paper, we introduce DeepFlow, a network-centric distributed tracing framework for microservice scenarios. First, to overcome the usability issues and the lack of network information, DeepFlow establishes a network-centric tracing plane with eBPF in the kernel. With kernel privileges, it can collect network metrics across multiple layers and eliminate blind spots in network infrastructure and closed-source components. Second, we utilize the collected network metrics for implicit context propagation. This technique, in conjunction with the hook-based instrumentation method, enables users to perform distributed tracing in zero code. Last but not least, DeepFlow develops the smart-encoding technique to reduce tag storage overhead. DeepFlow, based on our production experience, can save users hours of instrumentation efforts. Moreover, it reduces troubleshooting time from several hours to only a few minutes with the aid of enhanced network knowledge. Testbed evaluations show that DeepFlow has a negligible overhead of no more than 7%.

All real-world use cases and questionnaires were anonymized and gathered with the consent of the participants. *This work does not raise any ethical issues.*

ACKNOWLEDGMENTS

We thank anonymous SIGCOMM reviewers for their valuable comments. We thank engineers from Yunshan Networks for their genuine help. The research is supported by the National Natural Science Foundation of China under Grant No. 62002009 and No. 62102020. This work is part of Future Internet Technology Infrastructure (FITI).

REFERENCES

[1] Zaafar Ahmed, Muhammad Hamad Alizai, and Affan A. Syed. 2018. InKeV: In-Kernel Distributed Network Virtualization for DCN. *SIGCOMM Comput. Commun. Rev.* 46, 3, Article 4 (jul 2018), 6 pages. https://doi.org/10.1145/3243157.3243161

[2] Meta AI. 2022. Pytorch - Tensors and Dynamic neural networks in Python with strong GPU acceleration. (Nov. 2022). https://pytorch.org/

[3] Nadav Amit and Michael Wei. 2018. The Design and Implementation of Hyperupcalls. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. USENIX Association, USA, 97–111.

[4] Apache. 2022. Apache Hadoop project. (Nov. 2022). https://hadoop.apache.org/

[5] Apache. 2022. Apache Spark - Unified engine for large-scale data analytics. (Nov. 2022). https://spark.apache.org/

[6] Apache. 2023. Apache SkyWalking. (July 2023). Retrieved Jul, 2023 from https://skywalking.apache.org/

[7] Emre Ates, Lily Sturmann, 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 (SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 165–170. https://doi.org/10.1145/3357223.3362704

[8] AWS. 2023. AWS Lambda, Run code without thinking about servers or clusters. (Jan. 2023). https://aws.amazon.com/lambda/

[9] AWS. 2023. Tagging AWS resources. (Jan. 2023). https://docs.aws.amazon.com/general/latest/gr/aws_tagging.html

[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] Ashish Bijlani and Umakishore Ramachandran. 2019. Extension Framework for File Systems in User Space. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*. USENIX Association, USA, 121–134.

[12] Spring Boot. 2023. Jaeger Demo. (Jan. 2023). https://github.com/chanjarster/spring-boot-istio-jaeger-demo

[13] Peter Bourgon. 2017. Metrics, tracing, and logging. (Feb. 2017). https://peter.bourgon.org/blog/2017/02/21/metrics-tracing-and-logging.html

[14] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. 2007. Whodunit: Transactional Profiling for Multi-Tier Applications. *SIGOPS Oper. Syst. Rev.* 41, 3 (mar 2007), 17–30. https://doi.org/10.1145/1272998.1273001

[15] Zaheer Chothia, John Liagouris, Desislava Dimitrova, and Timothy Roscoe. 2017. Online Reconstruction of Structural Information from Datacenter Logs. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3064176.3064195

[16] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, USA, 217–231.

[17] Cilium. 2022. Hubble - Network, Service & Security Observability for Kubernetes using eBPF. (July 2022). Retrieved Feb, 2023 from https://github.com/cilium/hubble

[18] ClickHouse. 2023. Database. (Jan. 2023). https://clickhouse.com/

[19] ClickHouse. 2023. LowCardinality. (Jan. 2023). https://clickhouse.com/docs/en/sql-reference/data-types/lowcardinality/

[20] Open Zipkin Community. 2023. Zipkin tracing library for Python and C++. (Jan. 2023). https://github.com/dulikvor/cppKin

[21] CoreDNS. 2022. CoreDNS: DNS and Service Discovery. (Nov. 2022). https://coredns.io/

[22] Oracle Corporation. 2022. MySQL. (Nov. 2022). https://www.mysql.com

[23] DataDog. 2023. eBPF manager. This manager helps handle the life cycle of your eBPF programs. https://github.com/DataDog/ebpf-manager. (July 2023).

[24] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. 2019. Detecting Asymmetric Application-Layer Denial-of-Service Attacks in-Flight with Finelame. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*. USENIX Association, USA, 693–707.

[25] Luca Deri, Samuele Sabella, and Simone Mainardi. 2019. Combining System Visibility and Security Using eBPF. In *Proceedings of the Third Italian Conference on Cyber Security*. CEUR-WS.org, Italy, 1–12.

[26] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, USA, 139–150.

[27] Ryan Eberhardt. 2023. My First Kernel Module: A Debugging Nightmare. (Jan. 2023). https://reberhardt.com/blog/2020/11/18/my-first-kernel-module.html

[28] eBPF. 2023. eBPF - extended Berkeley Packet Filter. (Jan. 2023). https://ebpf.io/

[29] eBPF. 2023. eBPF Applications Landscape. (Jan. 2023). https://ebpf.io/applications

[30] Google Kubernetes Engine(GKE). 2023. Create and manage Tags in GKE. (Jan. 2023). https://cloud.google.com/kubernetes-engine/docs/how-to/tags

[31] 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

[32] Inc. F5. 2022. NGINX: Advanced Load Balancer, Web Server, & Reverse Proxy. (Nov. 2022). https://www.nginx.com

[33] Flask. 2022. Flask - The Python micro framework for building web applications. (Nov. 2022). https://github.com/pallets/flask

[34] 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.

[35] Apache Software Foundation. 2023. KAFKA PROTOCOL GUIDE. (Jan. 2023). https://kafka.apache.org/protocol.html

[36] Apache Software Foundation. 2023. Remote communication details of Dubbo. (Jan. 2023). https://dubbo.apache.org/en/docs/v2.7/dev/implementation/#remote-communication-details

[37] Cloud Native Computing Foundation. 2022. Production-Grade Container Orchestration. (Nov. 2022). https://kubernetes.io/

[38] Cloud Native Computing Foundation. 2023. Cloud Native Landscape. (Jan. 2023). https://landscape.cncf.io/

[39] Django Software Foundation. 2022. Django: The web framework for perfectionists with deadlines. (Nov. 2022). https://www.djangoproject.com

[40] Open Infrastructure Foundation. 2022. OpenStack: Open Source Cloud Computing Infrastructure. (Nov. 2022). https://www.openstack.org/

[41] Neves Francisco, Machado Nuno, and Pereira José. 2023. fntneves/falcon: Falcon: A practical log-based analysis tool for distributed systems. (Jan. 2023). https://github.com/fntneves/falcon

[42] Amazon GameLift. 2023. Dedicated server management for session-based multiplayer games. (Jan. 2023). https://aws.amazon.com/gamelift/

[43] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 135–151. https://doi.org/10.1145/3445814.3446700

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

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

[46] Xiongzi Ge, Yi Liu, David H.C. Du, Liang Zhang, Hongguang Guan, Jian Chen, Yuping Zhao, and Xinyu Hu. 2014. OpenANFV: Accelerating Network Function Virtualization with a Consolidated Framework in Openstack. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 353–354. https://doi.org/10.1145/2619239.2631426

[47] Francis Giraldeau and Michel Dagenais. 2016. Wait Analysis of Distributed Systems Using Kernel Tracing. *IEEE Transactions on Parallel and Distributed Systems* 27, 8 (2016), 2450–2461. https://doi.org/10.1109/TPDS.2015.2488629

[48] go.dev. 2023. The Go Programming Language. https://go.dev/. (July 2023).

[49] Google. 2022. TensorFlow - An end-to-end machine learning platform. (Nov. 2022). https://www.tensorflow.org/

[50] HAProxy. 2023. HAProxy Documentation. (Jan. 2023). http://docs.haproxy.org/2.7/configuration.html#7.3.6-unique-id

[51] HAProxy. 2023. The Reliable, High Performance TCP/HTTP Load Balancer. (Jan. 2023). http://www.haproxy.org/

[52] Red Hat. 2022. Red Hat OpenShift makes container orchestration easier. (Nov. 2022). https://www.redhat.com/en/technologies/cloud-computing/openshift

[53] HelmVMware. 2023. The package manager for Kubernetes. (July 2023). Retrieved Jul, 2023 from https://helm.sh/

[54] Jiamin Huang, Barzan Mozafari, and Thomas F. Wenisch. 2017. Statistical Analysis of Latency Through Semantic Profiling. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 64–79. https://doi.org/10.1145/3064176.3064179

[55] 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 (SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 76–91. https://doi.org/10.1145/3472883.3486994

[56] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and Enhancing in Situ System Observability for Failure Detection. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, USA, 1–16.

[57] International Business Machines Corporation (IBM) and Eurotech. 2023. MQTT V3.1 Protocol Specification. (Jan. 2023). https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html

[58] Internet Engineering Task Force (IETF). 2023. RFC 1035. (Jan. 2023). https://www.rfc-editor.org/rfc/rfc1035

[59] Internet Engineering Task Force (IETF). 2023. RFC 7231. (Jan. 2023). https://www.rfc-editor.org/rfc/rfc7231

[60] Internet Engineering Task Force (IETF). 2023. RFC 7540. (Jan. 2023). https://www.rfc-editor.org/rfc/rfc7540

[61] Istio. 2023. Bookinfo Application. (Jan. 2023). https://istio.io/latest/docs/examples/bookinfo/

[62] Jaeger. 2023. Jaeger: open source, end-to-end distributed tracing. (Jan. 2023). https://www.jaegertracing.io/

[63] Yurong Jiang, Lenin Ravindranath Sivalingam, Suman Nath, and Ramesh Govindan. 2016. WebPerf: Evaluating What-If Scenarios for Cloud-Hosted Web Applications. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 258–271. https://doi.org/10.1145/2934872.2934882

[64] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 34–50. https://doi.org/10.1145/3132747.3132749

[65] Suman Karumuri. 2023. PinTrace: Distributed Tracing at Pinterest. (August 2016). (Jan. 2023). https://www.slideshare.net/mansu/pintrace-advanced-aws-meetup

[66] The Linux Kernel. 2023. bpf(2) — Linux manual page. (Jan. 2023). https://man7.org/linux/man-pages/man2/bpf.2.html

[67] The Linux Kernel. 2023. eBPF verifier. (Jan. 2023). https://www.kernel.org/doc/html/latest/bpf/verifier.html

[68] The Linux Kernel. 2023. Kernel Probes (Kprobes). (Jan. 2023). https://www.kernel.org/doc/html/latest/trace/kprobes.html

[69] The Linux Kernel. 2023. Linux Socket Filtering aka Berkeley Packet Filter (BPF). (Jan. 2023). https://www.kernel.org/doc/html/latest/networking/filter.html

[70] The Linux Kernel. 2023. Uprobe-tracer: Uprobe-based Event Tracing. (Jan. 2023). https://www.kernel.org/doc/html/latest/trace/uprobetracer.html

[71] The Linux Kernel. 2023. Using the Linux Kernel Tracepoints. (Jan. 2023). https://www.kernel.org/doc/html/latest/trace/tracepoints.html

[72] Chung Hwan Kim, Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, Xiangyu Zhang, and Dongyan Xu. 2014. IntroPerf: Transparent Context-Sensitive Multi-Layer Performance Inference Using System Stack Traces. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '14)*. Association for Computing Machinery, New York, NY, USA, 235–247. https://doi.org/10.1145/2591971.2592008

[73] Kubernetes. 2023. Labels and Selectors. (Jan. 2023). https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

[74] PIXIE labs. 2022. Instantly troubleshoot your applications on Kubernetes. (July 2022). Retrieved Feb, 2023 from https://pixielabs.ai/

[75] Chien-An Lai, Josh Kimball, Tao Zhu, Qingyang Wang, and Calton Pu. 2017. milliScope: A Fine-Grained Monitoring Framework for Performance Debugging of n-Tier Web Services. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, USA, 92–102. https://doi.org/10.1109/ICDCS.2017.228

[76] AWS Lambda. 2023. Using AWS Lambda with other services. (Jan. 2023). https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html

[77] 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 (SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 326–332. https://doi.org/10.1145/3267809.3267841

[78] 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 (SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 312–324. https://doi.org/10.1145/3357223.3362736

[79] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. 2013. Improving Availability in Distributed Systems with Failure Informers. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (nsdi'13)*. USENIX Association, USA, 427–442.

[80] Joshua Levin and Theophilus A. Benson. 2020. ViperProbe: Rethinking Microservice Observability with eBPF. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEEE, USA, 1–8. https://doi.org/10.1109/CloudNet51028.2020.9335808

[81] 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 (SoCC '15)*. Association for Computing Machinery, New York, NY, USA, 182–188. https://doi.org/10.1145/2806777.2806940

[82] libbpf. 2023. Automated upstream mirror for libbpf stand-alone build. https://github.com/libbpf/libbpf. (July 2023).

[83] Linux. 2023. BPF Type Format (BTF). https://www.kernel.org/doc/html/next/bpf/btf.html. (July 2023).

[84] Linux. 2023. packet(7) — Linux manual page. (Jan. 2023). https://man7.org/linux/man-pages/man7/packet.7.html

[85] Linux. 2023. pidstat(1) — Linux manual page. (Jan. 2023). https://man7.org/linux/man-pages/man1/pidstat.1.html

[86] Chang Liu, Zhengong Cai, Bingshen Wang, Zhimin Tang, and Jiaxu Liu. 2020. A protocol-independent container network observability analysis system based on eBPF. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, Hong Kong, 697–702. https://doi.org/10.1109/ICPADS51040.2020.00099

[87] LTTng. 2023. LTTng: an open source tracing framework for Linux. (Jan. 2023). https://lttng.org/

[88] Liang Luo, Suman Nath, Lenin Ravindranath Sivalingam, Madan Musuvathi, and Luis Ceze. 2018. Troubleshooting Transiently-Recurring Problems in Production Systems with Blame-Proportional Logging. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. USENIX Association, USA, 321–334.

[89] 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

[90] LWN. 2023. A thorough introduction to eBPF. https://lwn.net/Articles/740157/. (July 2023).

[91] 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

[92] 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

[93] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (USENIX'93)*. USENIX Association, USA, 2.

[94] memcached. 2022. memcached - a distributed memory object caching system. (Nov. 2022). https://memcached.org/

[95] Haibo Mi, Huaimin Wang, Hua Cai, Yangfan Zhou, Michael R. Lyu, and Zhenbang Chen. 2012. P-Tracer: Path-Based Performance Profiling in Cloud Computing Systems. In *2012 IEEE 36th Annual Computer Software and Applications Conference*. IEEE, Izmir, Turkey, 509–514. https://doi.org/10.1109/COMPSAC.2012.69

[96] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, and Hua Cai. 2013. Toward Fine-Grained, Unsupervised, Scalable Performance Diagnosis for Production Cloud Computing Systems. *IEEE Transactions on Parallel and Distributed Systems* 24, 6 (2013), 1245–1255. https://doi.org/10.1109/TPDS.2013.21

[97] Microsoft. 2023. Azure for gaming. (Jan. 2023). https://azure.microsoft.com/en-us/solutions/gaming/

[98] Microsoft. 2023. Event Tracing for Windows | Microsoft Learn. (Jan. 2023). https://learn.microsoft.com/en-us/windows-hardware/test/wpt/event-tracing-for-windows

[99] J. Mogul, R. Rashid, and M. Accetta. 1987. The Packet Filter: An Efficient Mechanism for User-Level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP '87)*. Association for Computing Machinery, New York, NY, USA, 39–51. https://doi.org/10.1145/41457.37505

[100] Naver. 2023. Bytecode Instrumentation, Not Requiring Code Modifications. (Jan. 2023). https://pinpoint-apm.github.io/pinpoint/techdetail.html#how-bytecode-instrumentation-works

[101] Naver. 2023. Pinpoint | Leading Open-Source APM. (Jan. 2023). https://pinpoint-apm.gitbook.io/pinpoint/

[102] Nginx. 2023. Nginx Documentation - HTTP core module. (Jan. 2023). http://nginx.org/en/docs/http/ngx_http_core_module.html

[103] OpenTelemetry. 2023. High-quality, ubiquitous, and portable telemetry to enable effective observability. (Jan. 2023). https://opentelemetry.io/

[104] OpenTelemetry. 2023. Propagators Distribution. (Jan. 2023). https://opentelemetry.io/docs/reference/specification/context/api-propagators/#propagators-distribution

[105] OpenZipkin. 2023. B3-propagation. (Jan. 2023). https://github.com/openzipkin/b3-propagation

[106] Oracle. 2023. MySQL Client/Server Protocol. (Jan. 2023). https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_PROTOCOL.html

[107] Cuong Pham, Long Wang, Byung Chul Tak, Salman Baset, Chunqiang Tang, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2017. Failure Diagnosis for Distributed Systems Using Targeted Fault Injection. *IEEE Transactions on Parallel and Distributed Systems* 28, 2 (2017), 503–516. https://doi.org/10.1109/TPDS.2016.2575829

[108] Linux posts. 2023. Linux kernel crash dump analysis. (Jan. 2023). http://sklinuxblog.blogspot.com/2018/06/linux-kernel-crash-dump-analysis.html

[109] Kubernetes Documentation-Configuration Best Practices. 2022. (July 2022). Retrieved July, 2022 from https://kubernetes.io/docs/concepts/configuration/overview/

[110] Envoy Project. 2022. Envoy Proxy - Home. (Nov. 2022). https://www.envoyproxy.io/

[111] Envoy Project. 2023. HTTP header manipulation. (Jan. 2023). https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_conn_man/headers#x-request-id

[112] Prometheus. 2023. From metrics to insight. (Jan. 2023). https://prometheus.io/

[113] Redis. 2022. Redis - Remote Dictionary Server. (Nov. 2022). https://redis.io/

[114] Redis. 2023. Redis serialization protocol (RESP) specification. (Jan. 2023). https://redis.io/docs/reference/protocol-spec/

[115] Daniele Rogora, Antonio Carzaniga, Amer Diwan, Matthias Hauswirth, and Robert Soulé. 2020. Analyzing System Performance with Probabilistic Performance Annotations. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 43, 14 pages. https://doi.org/10.1145/3342195.3387554

[116] 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 (SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 401–414. https://doi.org/10.1145/2987550.2987568

[117] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. 2011. Diagnosing Performance Changes by Comparing Request Flows. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX Association, Boston, MA, 43–56. https://www.usenix.org/conference/nsdi11/diagnosing-performance-changes-comparing-request-flows

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

[119] Arjun Satish, Thomas Shiou, Chuck Zhang, Khaled Elmeleegy, and Willy Zwaenepoel. 2018. Scrub: Online Troubleshooting for Large Mission-Critical Applications. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 5, 15 pages. https://doi.org/10.1145/3190508.3190513

[120] Ben Sigelman. 2023. Towards Turnkey Distributed Tracing (June 2016). (Jan. 2023). https://medium.com/opentracing/towards-turnkey-distributed-tracing-5f4297d1736

[121] 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

[122] SQLite. 2022. SQLite Home Page. (Nov. 2022). https://www.sqlite.org

[123] 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

[124] Sysdig. 2023. Security Tools for Containers, Kubernetes, and Cloud – Sysdig. (Jan. 2023). https://sysdig.com/

[125] Open Telemetry. 2023. Open Telemetry's Golang net/http wrapper package. (Jan. 2023). https://pkg.go.dev/go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp

[126] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. 2017. Sieve: Actionable Insights from Monitored Metrics in Distributed Systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3135974.3135977

[127] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.* 53, 1, Article 16 (feb 2020), 36 pages. https://doi.org/10.1145/3371038

[128] VMware. 2023. RabbitMQ: easy to use, flexible messaging and streaming. (July 2023). Retrieved Jul, 2023 from https://www.rabbitmq.com/

[129] W3C. 2023. Trace Context W3C Recommendation 23 November 2021. (Jan. 2023). https://www.w3.org/TR/trace-context/

[130] World Wide Web Consortium (W3C). 2023. Trace Context HTTP Headers Format. (Jan. 2023). https://www.w3.org/TR/trace-context/#trace-context-http-headers-format

[131] 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 (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1242–1252. https://doi.org/10.1145/3468264.3473918

[132] Jianping Weng, Jessie Hui Wang, Jiahai Yang, and Yang Yang. 2018. Root Cause Analysis of Anomalies of Multitier Services in Public Clouds. *IEEE/ACM Transactions on Networking* 26, 4 (2018), 1646–1659. https://doi.org/10.1109/TNET.2018.2843805

[133] wrk2. 2022. wrk2 - A constant throughput, correct latency recording variant of wrk. (July 2022). Retrieved Feb, 2023 from https://github.com/giltene/wrk2

[134] Stephen Yang, Seo Jin Park, and John Ousterhout. 2018. NanoLog: A Nanosecond Scale Logging System. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18)*. USENIX Association, USA, 335–349.

[135] Yong Yang, Long Wang, Jing Gu, and Ying Li. 2022. Capturing Request Execution Path for Understanding Service Behavior and Detecting Anomalies without Code Instrumentation. *IEEE Transactions on Services Computing* 1, 1 (2022), 1–1. https://doi.org/10.1109/TSC.2022.3149949

[136] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. 2016. CloudSeer: Workflow Monitoring of Cloud Infrastructures via Interleaved Logs. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 489–502. https://doi.org/10.1145/2872362.2872407

[137] Jun Zhang, Robert Ferydouni, Aldrin Montana, Daniel Bittman, and Peter Alvaro. 2021. 3MileBeach: A Tracer with Teeth. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 458–472. https://doi.org/10.1145/3472883.3486986

[138] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. 2019. The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 131–146. https://doi.org/10.1145/3341301.3359650

[139] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 603–618.

[140] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, USA, 629–644.

[141] Zipkin. 2023. Zipkin. (Jan. 2023). https://zipkin.io/

A REQUESTS TRAVELING THROUGH A DATA CENTER

Appendices are supporting material that has not been peer-reviewed.

A.1 From End-Hosts to Gateways

Traditional distributed tracing frameworks focus on applica- tion-level components. Traces generated by them can, at most, include sidecars like Envoy [110]. However, the tracing capabilities of DeepFlow, which are built on top of the network information, extend beyond this limitation. If we deploy the DeepFlow Agent on the end-hosts, we can **extend the traces to the physical machines**, as shown in Figure 17. This is what the extended trace path will look like: client processes ⇔ sidecars ⇔ client Pods ⇔ client nodes ⇔ client physical machines ⇔ server physical machines ⇔ server nodes ⇔ server Pods ⇔ sidecars ⇔ server application processes.

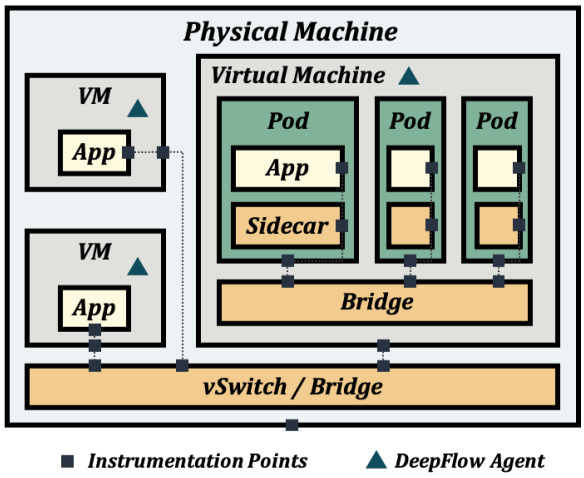


Figure 17: DeepFlow Agent extends traces to physical machines.

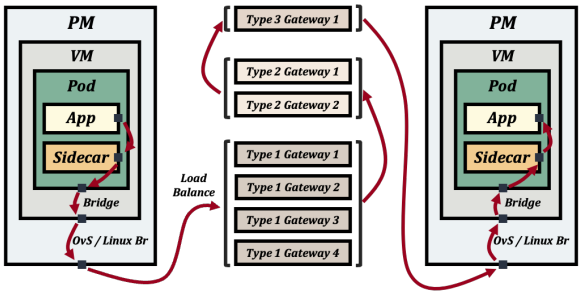
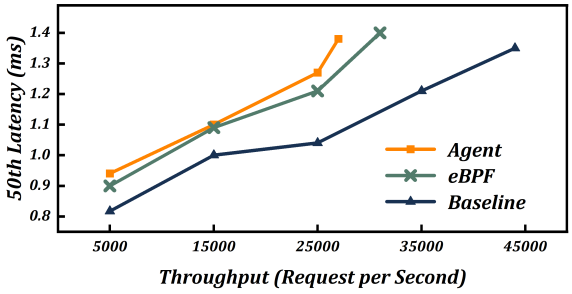


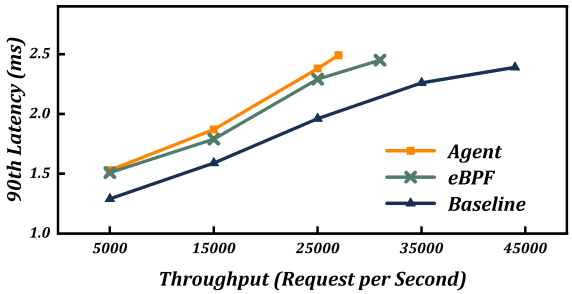
Figure 18: DeepFlow Agent extends traces to gateways.

Most importantly, DeepFlow can **include gateways in its traces**. DeepFlow Agent can run directly on L7 gateways (such as server load balancers) and treat them like standard host machines. Using the de facto standard, X-Request-IDs, we can easily track the requests across L7 gateways. On the other hand, since the majority of the L4 gateways do not modify the TCP sequence, we can utilize it to trace the requests that traverse the gateway. If we mirror the traffic on the top-of-rack switch to a physical machine dedicated to DeepFlow Agent, we can achieve coverage of the gateway (Figure 18).

We have now completed the full coverage of a request in the data center.



(a) Performance impact of DeepFlow Agent on the 50th latency.



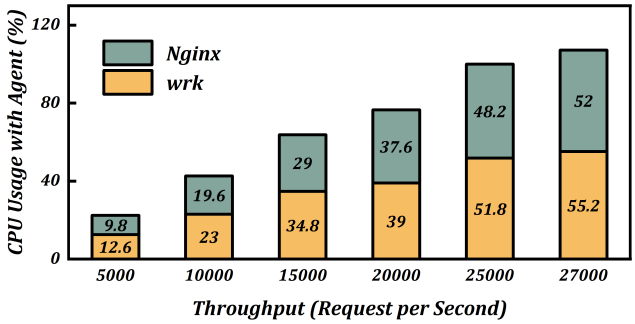
(b) Performance impact of DeepFlow Agent on the 90th latency.

Figure 19: Performance impact of DeepFlow Agent on the throughput and latency.

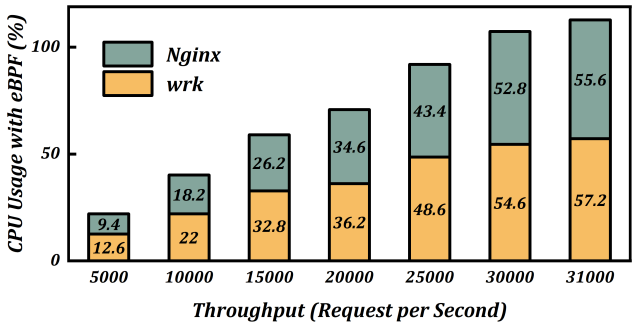
B SUPPLEMENTARY EVALUATION OF DEEPFLOW AGENT

In this supplementary experiment, we evaluate DeepFlow Agent's performance impact on the monitored applications. This test is conducted in a single VM with 8 vCPUs and 16 GB of RAM. We utilize wrk2 [133] as the load generator and Nginx [32] as the server. The performance impact is measured in three scenarios: no DeepFlow (denoted by Baseline), only running the eBPF module of DeepFlow Agent (denoted by eBPF), and executing the complete functionality of DeepFlow Agent (denoted by Agent). Then, we record the end-to-end latency as well as the CPU usage of the client and the server. Note that, in this testbed evaluation, the computational workload of Nginx takes only about 1 *ms*, which is considerably smaller than that of a real-world service. Thus, the performance impact of DeepFlow is overestimated. In a production application scenario, the influence of DeepFlow Agent will be much smaller.

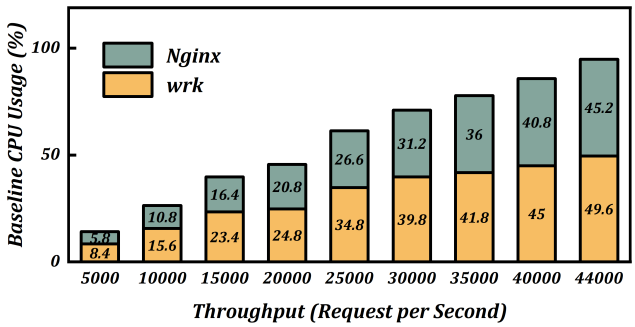
Figure 19 shows the results. The baseline Nginx service gets a throughput of 44, 000 RPS without enabling the DeepFlow Agent.



(a) Resource consumption when DeepFlow Agent is fully functioning.



(b) Resource consumption with the eBPF module of DeepFlow.



(c) Resource consumption at the baseline settings.

Figure 20: Resource consumption of the client (wrk) and the server (Nginx).

It is, however, reduced to 31,000 RPS when eBPF is activated. As mentioned previously, the reason for the decrease in throughput is that we evaluate DeepFlow under the theoretically strictest conditions. When all features of DeepFlow Agent are turned on, the maximum throughput drops to 27,000 RPS.

C UNPROCESSED DATA FROM THE QUESTIONNAIRE

To better understand how DeepFlow helps improve observability in production settings, we designed a questionnaire for users with the following questions:

(1) Is the distributed tracing framework you use open-source or self-developed?
(2) How many kernel versions do you have in your online production environment?
(3) How many programming languages are used in your project?
(4) How many microservice components do you have in your project?
(5) How many lines of code does your project have for a single application or a single microservice component?
(6) Before using DeepFlow, how long did it take to instrument a single program or a single microservice component?
(7) Before using DeepFlow, how many lines of code did you need to modify for instrumenting a single program or a single microservice component (including framework initialization and connection)?
(8) Compared with other distributed tracing frameworks, how much has DeepFlow reduced your workload?
(9) Before using DeepFlow, what was the average time from discovering a fault to fixing the problem (including cross-department communication time)?
(10) Having deployed DeepFlow, what is the average time from discovering a fault to fixing the problem (including cross-department communication time)?
(11) Where has DeepFlow helped you the most?

We collected 10 valid results, which are shown in Table 4 and Table 5.

Q	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10
1●	O	S	O	O	O	O	S	O	O	S
2	2-5	5-10	2-5	2-5	Unknown	2-5	2-5	2-5	2-5	2-5
3	2-5	2-5	2-5	2-5	2-5	2-5	2-5	2-5	2-5	2-5
4	2-5	>100	5-10	>100	20-100	10-20	5-10	10-20	2-5	>100
5	100-1k	3k-5k	3k-5k	3k-5k	>5k	>5k	100-1k	1k-3k	3k-5k	>5k
6	Days	Days	Hrs	1Hr	Mins	Hrs	Hrs	Mins	Hrs	1Hr
7	(20,100]	(0,20]	>100	(0,20]	0	>100	>100	0	(20,100]	(20,100]
8	20%-50%	50%-80%	20%-50%	50%-80%	50%-80%	20%-50%	>80%	50%-80%	20%-50%	0%
9	1Hr	Hrs	Hrs	Hrs	Hrs	Mins	1Hr	Mins	Hrs	1Hr
10	1Hr	Hrs	1Hr	Mins	1Hr	Mins	1Hr	Mins	1Hr	1Hr

● O denotes open-source, and S denotes self-developed.

Table 4: Questionnaire answers for multiple-choice questions.

Q11	Where has DeepFlow helped you the most?
1	It helps me to check network status and response latency between two microservices, making slow request troubleshooting easier.
2	Its non-intrusive characteristic can help detect previous blind spots in the system, such as components written in Golang or Rust. But it is not very useful for Java components, since skywalking is already sufficient for us.
3	Locating problems with network data non-intrusively.
4	Microservice Network Fault Location.
5	Network problem diagnosis.
6	It complements existing observability tools by providing more detailed traces and enriching the set of metrics.
7	It can capture the time consumption of services and middleware at the network level. Besides, a lot of work is reduced by its non-intrusive characteristic.
8	Non-intrusive, low-cost deployment.
9	(Empty)
10	It can help us find some problems in the system, but we haven't found a way to locate the problem precisely.

Table 5: Questionnaire answers for the short answer question.