Empirical Software Engineering (2022) 27: 25
https://doi.org/10.1007/s10664-021-10063-9

Enjoy your observability: an industrial survey of microservice tracing and analysis

Bowen Li[1,2] · Xin Peng[1,2] · Qilin Xiang[1,2] · Hanzhang Wang[3] · Tao Xie[4,5] · Jun Sun[6] · Xuanzhe Liu[4,5]

Accepted: 11 August 2021 / Published online: 30 November 2021
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2021

Abstract
Microservice systems are often deployed in complex cloud-based environments and may involve a large number of service instances being dynamically created and destroyed. It is thus essential to ensure observability to understand these microservice systems' behaviors and troubleshoot their problems. As an important means to achieve the observability, distributed tracing and analysis is known to be challenging. While many companies have started implementing distributed tracing and analysis for microservice systems, it is not clear whether existing approaches fulfill the required observability. In this article, we present our industrial survey on microservice tracing and analysis through interviewing developers and operation engineers of microservice systems from ten companies. Our survey results offer a number of findings. For example, large microservice systems commonly adopt a tracing and analysis pipeline, and the implementations of the pipeline in different companies reflect different tradeoffs among a variety of concerns. Visualization and statistic-based metrics are the most common means for trace analysis, while more advanced analysis techniques such as machine learning and data mining are seldom used. Microservice tracing and analysis is a new big data problem for software engineering, and its practices breed new challenges and opportunities.

Keywords Microservice · Logging · Tracing · Industrial survey

Communicated by: Antonia Bertolino

✉ Xin Peng
pengxin@fudan.edu.cn

[1] School of Computer Science, Fudan University, Shanghai, China

[2] Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China

[3] eBay Inc., San Jose, CA, USA

[4] Peking University, Beijing, China

[5] Key Laboratory of High Confidence Software Technologies, (Peking University), Ministry of Education, Beijing, China

[6] Singapore Management University, Singapore, Singapore

1 Introduction

Microservice architecture has been defined in multiple different ways (Francesco et al. 2017). Some regard it as an architectural style in which a single application is implemented as a suite of small services, each running in its own process and communicating via lightweight mechanisms (often an HTTP resource API) (Lewis and Fowler 2020). Others believe that it is a new implementation approach to Service-Oriented Architecture (SOA) (Newman 2015). Microservice has been the mainstream for cloud native applications. Industrial microservice systems may include hundreds to thousands of services running in a complicated cloud infrastructure and with a large number of service instances created and destroyed dynamically. Understanding and diagnosing problems (e.g., failed requests and high latency) in a microservice architecture is highly complicated as there may be a lot of services involved (Richardson 2019). It is further complicated by the complex cloud-based runtime environment consisting of physical machines, virtual machines, and containers. Therefore, observability has been regarded as an essential requirement for microservice systems.

Widely accepted and practiced in industry, distributed tracing is an important means to achieve observability in microservice architecture, by tracing requests as they flow between services (Richardson 2019). Its implementation usually follows a tracing and analysis pipeline as shown in Fig. 1, which includes the logging of service invocations, collection of generated logs, preprocessing of the collected logs, storage of trace data, and analysis of restored traces.

- **Logging**: A logger is accompanied with each service instance. It produces a span log for each service invocation, recording information such as trace ID, span ID, parent span ID, timestamp, and duration.
- **Collection**: A centralized collector collects span logs from service instances to enable data preprocessing and trace analysis.
- **Preprocessing**: The collected span logs are preprocessed before persisted in storage for trace analysis. The preprocessing usually includes formatting log data and aggregating various metrics.
- **Storage**: The preprocessed trace data is persisted in various forms for further analysis.
- **Analysis**: Operation engineers and developers analyze trace data in different ways for various purposes such as service dependency analysis, anomaly detection, and problem diagnosis.

Google and Facebook have developed their own distributed tracing infrastructures namely Dapper (Sigelman et al. 2010) and Canopy (Kaldor et al. 2017). The OpenTracing project (2020) provides a language-neutral data model for distributed tracing, defining basic concepts such as span and trace. The project also defines an API specification and provides

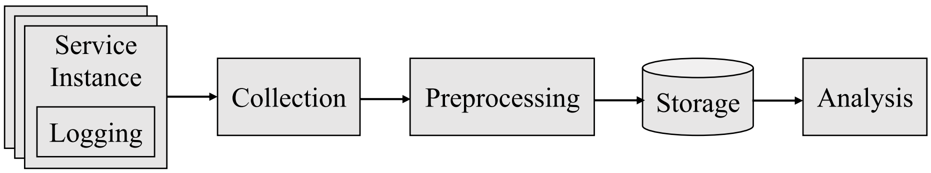


Fig. 1 Microservice Tracing and Analysis Pipeline

frameworks and libraries that have implemented the specification. There have been open-source implementations following the OpenTracing framework, e.g., Skywalking (2020), Zipkin (2020), and Jaeger (2020). Based on the collected traces, various real-time and non-real-time analyses can be conducted for different purposes such as understanding service dependencies and runtime architecture, and troubleshooting failed requests and high latency.

While many companies have started implementing distributed tracing and analysis for microservice systems, it is not clear whether existing approaches fulfill the required observability, given that distributed tracing and analysis is challenging due to two main factors. First, distributed tracing involves implementing logging in a large number of services developed in different languages, and collecting and processing a huge number of logs generated continuously. A large microservice system may produce up to millions of traces per second and each trace may involve a complex invocation chain spanning dozens of services and even multiple data centers. Second, the subsequent trace analysis needs to meet different purposes such as anomaly detection, fault analysis, and debugging (Pham et al. 2017; Zhou et al. 2018, 2019), while being complicated by the huge volume of data and the complexity of microservice system behaviors. The behaviors of a microservice system are influenced by the underlying complex cloud infrastructure including computing and storage systems, and networks. The problems of a microservice system may be rooted from a variety of factors such as environmental configurations, deployment structures, service interactions, and service implementations (Zhou et al. 2021).

To investigate whether existing approaches fulfill the required observability in industrial practices, in this article, we present our industrial survey on microservice tracing and analysis through interviewing developers and operation engineers of microservice systems from ten companies. Except two very small systems, all the surveyed microservice systems have established practices of distributed tracing and analysis. For each of these systems, we investigate the practices used for logging, collection, and preprocessing of trace data, the reasons for the chosen solutions and their pros and cons, different trace analysis applications, and the used data/techniques. We also investigate the faced challenges and expectations for better techniques and solutions.

Our survey results show findings on the pipeline of tracing and analysis along with purposes and techniques of trace analysis. First, large microservice systems commonly adopt a tracing and analysis pipeline as shown in Fig. 1, which includes logging, collection, preprocessing, storage, and analysis. The implementations of the pipeline in different companies reflect different tradeoffs among a series of concerns such as easy-to-implement, flexibility, effort, overhead, and completeness. Second, trace analysis has been widely used for various purposes such as understanding the runtime architectures and service dependencies, and discovering and locating functional or non-functional faults. Trace analysis applications used in the surveyed systems include timeline analysis, service dependency analysis, aggregation analysis, root cause analysis, and anomaly detection. Third, intelligent trace analysis is still in its infancy. Visualization and statistic-based metrics are the most common means for trace analysis, while more advanced analysis techniques such as machine learning and data mining are seldom used.

Our survey results indicate that microservice tracing and analysis is a new big data problem for software engineering, and its practices breed new challenges and opportunities. Example future directions include adaptive log sampling, data fusion for trace analysis, more intelligent trace analysis, and business intelligence by trace analysis. Traditional software engineering techniques such as program analysis, reverse engineering, and debugging need to be further developed and combined with latest advances in cloud computing and data science to facilitate microservice tracing and analysis.

The rest of the article is organized as follows. Section 2 introduces the background knowledge for distributed tracing and observability. Section 3 presents the methodology of our industrial survey. Section 4 summarizes the results of the survey. Section 5 discusses the challenges and opportunities of microservice tracing and analysis. Section 6 introduces some related work and compares it with ours. Section 7 concludes the article and outlines future work.

2 Background

In a computing system, log files record events occurred or messages generated during system operations or software executions. Logging is the act of keeping a log (2020). Traditional logs are generally divided into two categories:

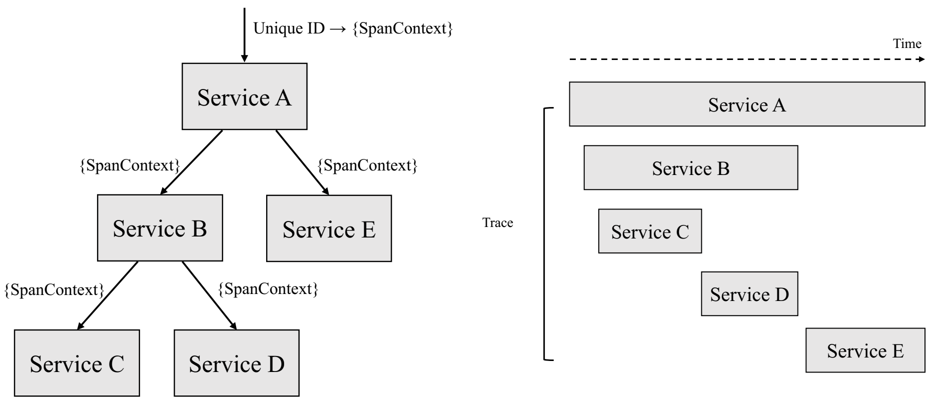
- Application logs record application-specific events such as exceptions manually (e.g., issuing a statement such as *System.out.print()*) or using a logging framework (e.g., Log4j 2020; Logback 2020).
- System logs record events that are logged by operating system components. The operating system itself often predetermines the events to be logged.

These logs are often used to help software developers or operation engineers to conduct anomaly detection, root cause analysis, process mining, or performance optimization for a single service or component.

For modern distributed systems, it is crucial to trace the service invocation chain (i.e., a series of service invocations) for each request. The records of service invocation chains enable the understanding of runtime service dependencies and the analysis of the execution process of each request through different services and nodes. This requirement leads to the appearance of distributed tracing (OpenTracing 2020), which is an approach used to profile and monitor distributed systems, especially those built using a microservices architecture. A trace represents a series of causally related distributed events that encode the end-to-end request workflow through a distributed system (Sridharan 2018). Distributed tracing relies on the logging of service invocations by each service instance. The logged service invocations can then be restored to traces.

According to the OpenTracing specification, a trace is the description of a transaction (i.e., the execution process of a request) as it moves through a distributed system (OpenTracing 2020). A trace has a tree structure consisting of service invocations as shown in Fig. 2a. When a request arrives, a unique trace ID (which is also called correlation ID (Brown 2021)) is generated and assigned, and then carried and propagated throughout the request workflow. As shown in Fig. 2b, each hop along the request workflow is represented as a span, which is a named and timed operation (i.e., service invocation) representing a piece of the workflow. A span also has a unique ID, called span ID. A span context represents trace information that accompanies the distributed transaction, including when it passes the service to service over the network or through a message bus. The span context contains the trace ID, span ID, and any other data that the tracing system needs to propagate to the downstream services. In addition, each service instance can record (for each span) zero or more fine-grained, structured key-value tags, which are called span tags. The span tags are usually used to record the services that generate the span, the database statement, and any other data required for log or trace analysis.

When an instrumentation point is triggered at a service instance, a record is generated as a span log. Span logs are usually asynchronously persisted before being submitted to a



(a) Trace Topology (b) Trace and Span

Fig. 2 Traces and Spans in Distributed Tracing

collector, and then get restored to a trace based on the different logs emitted by different services (Sridharan 2018). An example of span log is shown in Fig. 3. It records the trace ID (Line 2), the span ID (Line 3), the ID of the parent span (Line 4), the timestamp (Line 5) and duration (Line 6) of the invocation (with both units being microsecond), and the service names and addresses of the caller (Lines 7-10) and callee (Lines 11-14). The annotations in

```
1    {
2        "traceId": "5982fe77008310cc80f1da5e10147517",
3        "spanId": "be2d01e33cc78d97",
4        "parentSpanId": "ebf33e1a81dc6f71",
5        "timestamp": 1458702548786000,
6        "duration": 13000,
7        "localEndpoint": {
8          "serviceName": "client",
9          "ipv4": "192.168.1.2",
10         "port": 9411},
11       "remoteEndpoint": {
12         "serviceName": "server",
13         "ipv4": "127.0.0.1",
14         "port": 3306 },
15       "annotations": [{
16           "timestamp": 1458702548786000,
17           "value": "cs"
18         },{
19           "timestamp": 1458702548799000,
20           "value": "cr"
21         },{
22           "timestamp": 1458702548788000,
23           "value": "sr"
24         },{
25           "timestamp": 1458702548796000,
26           "value": "ss"}]
27   }
```

Fig. 3 An Example of Span Log

the span log record the timestamps of the client sending the request (Lines 16-17), the client receiving the response (Lines 19-20), the server receiving the request (Lines 22-23), and the server sending the response (Lines 25-26), respectively.

There are a growing number of industry implementations of distributed tracing systems, including (AWS's X-Ray 2020; Google's Dapper 2020; Alibaba's ARMS 2020; Dynatrace 2020; Splunk 2020). Furthermore, many open-source projects are available to the community: Skywalking (2020), HTrace (2020), Zipkin (2020), and Jaeger (2020). They are often used together with message bus such as Apache Kafka (2020) and storage such as ElasticSearch (2020).

Logs, traces, and metrics are often known as the three pillars of observability and serve their own unique purpose and are complementary (Sridharan 2018). For a microservice system, the logs include application logs, system logs, and also span logs; the traces are produced by the distributed tracing system and restored from span logs; the metrics include both system metrics (e.g., CPU/memory consumption rate) and application metrics (e.g., latency and error rate of service invocations).

3 Methodology

We conduct our study with a duration of four months using interview, being a common qualitative research approach (Kvale 2008). The overall procedure is illustrated in Figure 4. We divide the procedure into three parts: interview planning, interview process, and interview result analysis.

3.1 Interview Planning

At the planning stage, all authors of this article study critical procedures in distributed tracing, brainstorm for research questions, search for appropriate interviewees, and draft interview templates.

First, we search for appropriate interviewees. This study is designed to gain broad and in-depth knowledge of distributed tracing in the real-world context of software industry. Therefore, the interview candidates must be carefully selected. In particular, the candidates

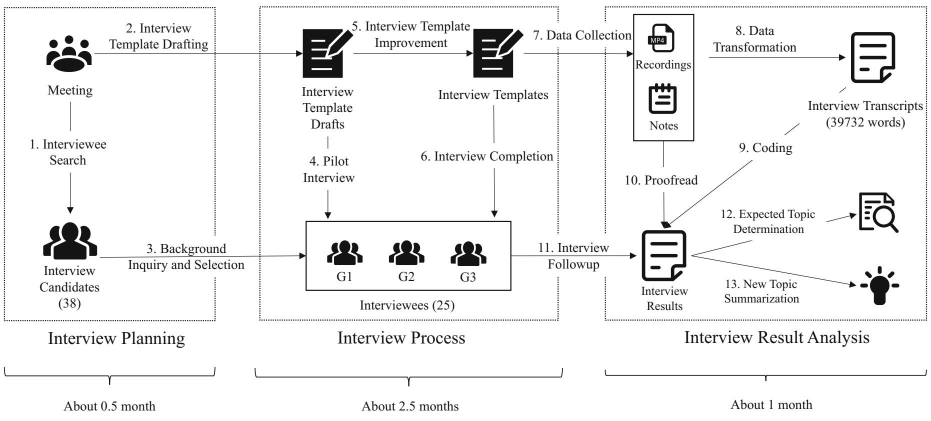


Fig. 4 Research Procedure of Interview

must have hand-on experiences with distributed tracing systems and are currently working on microservice systems as various roles (e.g., application or infrastructure developers, operation engineers).

We identify interview candidates from three sources: technical summits, technical forums, and industrial research partners. Two authors of this paper have previously acquainted with multiple industry partners at conferences (e.g., KubeCon 2019, ASE 2019), and these industry partners are invited as interviewees. An author of this paper owns a software engineering technical forum, and we invite relevant industry partners from the forum. Besides, two interviewed companies are in active partnership with our lab on tracing related research. In total, we invite 38 candidates who are interested in and familiar with distributed tracing in microservice systems.

Then all authors of this article brainstorm and draft the interview templates. We design the interview templates such that interview would take around 30-80 minutes. To partition interviewees into different groups, we create a background interview template to get some necessary information about the interviewees, their company, and their microservice system. The background interview template is as follows.

T1. Background Interview Topics

- **Experience and Background:** Interviewee's working experience, job title, familiarity with distributed tracing and other general information.
- **Company Overview:** Company type, and company size.
- **Microservice System Overview:** Microservice system introduction, the number of services, deployment environment.

With the background interview template, we make the background inquiry with the 38 interview candidates. For experiences, we require the interviewees to be knowledgeable about distributed tracing practices and engineering. For systems, we require the systems that the interviewees work on to adopt microservice architecture. The interviewees from the companies with tracing capabilities are required to have hand-on tracing development experience or use the tracing application for their core job functionality. This information is also later used to split the candidates into groups. We also select the diversified interviewees to work for different sizes and types of company, as well as the microservice system types or tracing capabilities. We finally select 25 interviewees from ten companies. These companies cover five different domains (i.e., E-commerce, entertainment, finance, manufacture, and ERP), and thus can well represent the general practices of industrial microservice systems.

3.2 Interview Process

In this stage, we split interviewees into groups, conduct pilot interview to improve the quality, complete full interviews, and collect results.

Based on the background inquires, we partition the interviewees into three groups, i.e., (G1) six interviewees working on microservice systems that do not have distributed tracing capabilities for now. Still, they are familiar with distributed tracing and work in related domains (e.g., monitoring, site reliability engineering); (G2) ten interviewees who are tracing system developers or operation engineers who develop or maintain the tracing systems in their companies; (G3) nine interviewees who are application developers in their companies and use the data or applications of distributed tracing. We design different interview templates for different groups.

For G1, we explore the reasons why the interviewees do not use distributed tracing in their microservice systems and whether they plan to implement distributed tracing systems in the future. For G2, we focus on the practices that the interviewees have explored and adopted to implement the distributed tracing pipeline, i.e., logging, collection, preprocessing, storage, and analysis. For G3, we explore what kind of analysis the interviewees perform with the distributed tracing systems. Besides, we design the interviews to be semi-structured: we have question sections to be covered comprehensively; still, we allow the interview conversation to go into depth when the interviewee mentions related experience or pain points. During the deep dives, we discover practices and challenges.

In this way, we design the three interview templates as follows (note that all questions are designed to be open-ended so that interviewees can elaborate with details).

T2. Interview template for $G1$: the interviewees who work on microservice systems without tracing capability

- Why not equip the microservice systems with distributed tracing?
- What are the challenges of implementing distributed tracing?
- Is there any plan to implement distributed tracing in the future?

T3. Interview template for $G2$: the interviewees who implement and maintain distributed tracing

- How is the trace logging designed and implemented? What are the pros, cons, and challenges?
- How is the trace collection designed and implemented? What are the pros, cons, and challenges?
- How is the trace preprocessing designed and implemented? What are the pros, cons, and challenges?
- How is the trace storage designed and implemented? What are the pros, cons, and challenges?
- How is the trace application designed and implemented? What are the pros, cons and challenges?
- What are the future work and expectations for the tracing systems?

T4. Interview template for $G3$: the interviewees who consume tracing data or use its application

- What are the applications of distributed tracing?
- How do these applications benefit or support your work?
- What are the expectations or opportunities to improve the current practice?

After we draft the interview templates, we conduct pilot interviews with three experienced engineers (one from each group). During the interviews, we take notes of their comments and suggestions. Then according to their feedback, we improve the interview templates and adjust the wording to make sure that the interview questions are unambiguous and meaningful.

Next, we conduct the official interviews. The face-to-face meetings are conducted and later switch to remote video calls due to the coronavirus pandemic. We spend about two months on carrying out all interview sessions. Each session involves at least two researchers as the interviewers. During the interview, one researcher is primarily engaged in asking questions and driving the discussion. The others are mainly responsible for taking notes and asking supplementary questions. Meanwhile, to assure the correctness of our study, we record all of the interview sessions with the interviewee permission. The recordings are done by using a digital voice or screen recorder.

3.3 Interview Result Analysis

We collect the recordings and notes from the interview process, and study them to conclude and sum up the key findings of this study. We convert over 700 minutes' interview recordings into text documents with the aid of a speech recognition application. Then we combine our notes with the text documents into complete interview transcripts (with 39,732 words).

Next, two students code the interview transcripts following the open coding manual (Strauss and Corbin 1990). They gradually discuss and refine the codes and organizing them by hierarchy. During the coding process, the transcripts are cross-validated and corrected with meeting notes. Each interview's outcome is also presented and discussed in our research group. During the discussion, we sum up the discoveries and plan follow-up interviews. We have follow-up interviews with the interviewees of S1-S8 to learn about some missing details and the later discovered practices or applications (such as analysis application).

To learn from the interview results, we determine the expected topics such as tracing implementation and classic usage of tracing. We focus on commonly applied tracing practices, i.e., how to implement the distributed tracing pipeline, as shown in Section 4. We also discover a lot of new knowledge along the way, such as novel trace analysis applications, as shown in Section 4.5. For discoveries, we sum up new topics by discussing and horizontally comparing the findings from different companies. Section 5 presents many discovered tracing practices and applications.

4 Results

Our survey covers ten microservice systems from ten different companies. An overview of these systems is shown in Table 1. The ten companies belong to different industries, including five Internet companies, three finance companies, one manufacturing company, and one IT company. Among these companies, C1, C2, C3, and C6 are Fortune 500 companies. The ten subject systems belong to different business domains, including e-commerce, finance, manufacture, entertainment, and ERP. These systems have different sizes, ranging from 10+ to 100,000+ services and 5K+ to 100K+ lines of code per service. S1 is a huge ecosystem across multiple sub-domains of e-commerce, and thus includes a large number of services. These systems adopt different deployment environments: S1, S2, and S5 use physical machines, virtual machines, and containers at the same time; S6 uses both physical

Table 1 Overview of companies

Sys.	Company	Domain	#Service	Service Size (KLOC)	Deployment
S1	C1 (Internet)	E-commerce	10,000+	50+	PM, VM, CT
S2	C2 (Internet)	E-commerce	3,000+	50+	PM, VM, CT
S3	C3 (Internet)	E-commerce	1,000+	50+	VM, CT
S4	C4 (Internet)	Entertainment	700+	30+	VM
S5	C5 (Finance)	Finance	200+	100+	PM, VM, CT
S6	C6 (Manufacture)	Manufacture	100+	50+	PM, VM
S7	C7 (Finance)	Finance	50+	80+	VM
S8	C8 (Finance)	Finance	40+	20+	VM
S9	C9 (Internet)	E-commerce	40+	5+	VM
S10	C10 (IT)	ERP	10+	20+	VM, CT

PM: Physical Machine; VM: Virtual Machine; CT: Container

machines and virtual machines; S3 and S10 use both virtual machines and containers; S4, S7, S8, and S9 use only virtual machines.

All the ten systems except S9 and S10 (which the interviewees in G1 are working on) have established the practices of distributed tracing and analysis. For S9, the interviewees mention that the company has planned to establish such infrastructures in the near future. For S10, the interviewees say that considering the return on investment they do not plan to introduce tracing into their microservice system as it is small and the traces are very short (e.g., involving two to three services). In contrast, for all the other microservice systems, distributed tracing has been an important part of their microservice infrastructures and supports a variety of development and operation tasks (see the applications in Section 4.5). From the survey, we can see that distributed tracing is less necessary for small microservice systems where the number of services is small and service invocation chains are short. For these small systems, the developers usually can easily understand service interactions and locate individual faulty services and thus do not rely on traces.

The eight systems with distributed tracing (i.e., S1-S8) commonly implement the tracing and analysis pipeline as shown in Fig. 1, which includes five parts, i.e., logging, collection, preprocessing, storage, and analysis. Despite sharing the same logical structure, the implementations of the pipeline in different companies differ in its architecture and constituent components. For example, the pipeline implemented for S1, S2, S3, S4, S5, and S6 uses a message bus to increase the reliability of log transmission and collection (S1 and S3 use their own message buses and the others use Kafka 2020); the pipeline implemented for S1, S2, S3, S4, and S6 uses a stream computing engine to achieve real-time processing of trace data (S1 uses their own stream computing engine and the others use Flink 2021). Some companies (e.g., the companies of S1, S2, S3, S4, and S6) choose to develop their own distributed tracing systems from scratch to implement the pipeline and make it a part of their microservice infrastructures. These companies, especially those large Internet companies, usually have a devoted team to maintain and manage the infrastructures. The pipeline for S5 and S8 is implemented based on the ELK stack, i.e., Logstash (2020) for log collection, ElasticSearch (2020) for log indexing and retrieval, and Kibana (2020) for visualization. The pipeline for S7 uses the all-in-one solution provided by SkyWalking (2020), which implements distributed tracing, service mesh telemetry analysis, metric aggregation, and visualization. These smaller companies have only several engineers who maintain and manage the infrastructures.

The results presented in the remaining of this section are all based on the eight systems with distributed tracing (i.e., S1-S8).

4.1 Logging

The logging practices of the surveyed microservice systems can be summarized from three aspects: logger implementation, i.e., how to implement the span logger and integrate it with service instances; content and format, i.e., what content is recorded in the span logs and in what format; sampling, i.e., how to choose the service invocations to be logged. The results are shown in Table 2, including the implementation techniques, supported languages, coverage of services, log format, and sampling strategy.

4.1.1 Logger Implementation

The surveyed microservice systems adopt the following three logger implementation techniques.

– **Manual Coding**: Service developers manually implement logging in their code following relevant standards and using given Software Development Kits (SDKs). Usually the developers need to initiate a tracer when a service starts and call the tracer before and after a service invocation to produce a span log.
– **Tracing Framework**: The microservice system to be logged deploys a tracing framework in its runtime infrastructure. Service developers need to only implement service invocations as specified by the framework, e.g., using specific APIs for service invocations. The developers do not need to implement additional logging functionalities in their code. For example, when using the Spring Boot framework, the developers need to use a set of predefined APIs (e.g., *RestTemplate*) for service invocation, and the framework implicitly records the span logs.
– **Dynamic Binary Instrumentation**: The microservice system to be logged uses a runtime agent to dynamically inject instrumentation code into service instances. The instrumentation code implements the required logging functionalities. Service developers do not need to do anything extra for logging.

Table 2 Logging practices of the surveyed systems

System	Technique	Supported language	Coverage	Format	Sampling
S1	T-MC, T-TF	Java, Go, Node.js	90%	Customized	√
S2	T-MC, T-TF	Java, C++, Node.js	98%	OpenTracing	√
S3	T-MC, T-DBI	Java	90%	Customized	√
S4	T-TF	Java, C++, Node.js	90%	Customized	√
S5	T-TF	Java	90%	Customized	×
S6	T-TF	Java, C++	98%	Customized	×
S7	T-DBI	Java	100%	Customized	×
S8	T-TF	Java, C++, Go	98%	OpenTracing	×

The characteristics of these techniques, including advantages and disadvantages, are summarized in Table 3. The technique of manual coding is intrusive and requires additional effort, but easy to implement. The developers can flexibly decide the logging position and log content. This technique is error prone, as the developers may make mistakes, e.g., passing a wrong parameter to the logger. The technique of tracing framework is weakly intrusive, as the developers need to load the framework into the development environment and implement service invocations as specified by the framework. This technique requires a little additional effort and is less error prone, but requires a heavy-overhead framework and is inflexible in logging positions and content. The technique of dynamic binary instrumentation is non-intrusive and less error prone, and requires no additional effort. But this technique is language-dependent, as the service to be logged needs to be implemented by a language that supports dynamic instrumentation (e.g., Java). Also this technique is inflexible in logging positions and content.

From Table 2, it can be seen that six out of eight microservice systems use the technique of tracing framework for logging implementation; the other two microservice systems use the technique of dynamic binary instrumentation. It can be seen that the technique of tracing framework is more preferred, as it makes a good tradeoff between intrusive and non-intrusive logging. The technique of manual coding is used in the three largest microservice systems (i.e., S1-S3) as a complementary technique for two reasons. First, large microservice systems often include legacy services that are developed using outdated techniques, and the technique of tracing framework or dynamic binary instrumentation may not work for these services. Second, these microservice systems many need to add application specific information (e.g., order ID) to the logs to support more sophisticated trace analysis. All of the eight microservice systems support Java for logging and some systems also support Go, Node.js, and C++. In all these microservice systems, the coverage of service logging is above 90%. The reasons for the services that are not covered are usually that these services are not core business services and there is no need to involve them in trace analysis; they are legacy services and it is hard to implement logging for them.

4.1.2 Content and Format

Nearly all the surveyed microservice systems include in their span logs basic information that is necessary for restoring and analyzing a trace. This information includes trace ID, span ID, invoked service name, timestamp, duration, and error tag. These microservice systems also record additional information in the span logs to facilitate fault diagnosis, e.g., order ID for business faults and data center name for infrastructure related faults.

Table 3 Logger implementation techniques

Technique	Intrusion	Advantage	Disadvantage
T-MC: Manual Coding	Intrusive	easy to implement, flexible	additional effort, error prone
T-TF: Tracing Framework	Weakly Intrusive	a little effort, less error prone	heavy-overhead framework, inflexible
T-DBI: Dynamic Binary Instrumentation	Non-Intrusive	no additional effort, less error prone	language dependent, inflexible

S2 and S7 follow the log formats suggested by the OpenTracing specification, which defines a series of fields for spans, e.g., operation name, start time, end time, tags, and span contexts. S2 used customized log formats at first. When OpenTracing became popular in 2017, the company decided to shift to OpenTracing because the company thought that it could benefit from the development of the active OpenTracing community. S7 uses the all-in-one solution provided by SkyWalking, so it naturally follows OpenTracing. The other microservice systems use their own formats for the following respective reasons. S3 believes that OpenTracing cannot meet its customized requirements, e.g., the special format of span ID for indicating the relationships between spans; S4 and S6 develop their own specifications before OpenTracing and think that it is unnecessary to migrate to OpenTracing. S8 believes that OpenTracing is too complicated for S8, and it requires much effort to adapt to OpenTracing based on S8's existing ELK stack. These customized formats define similar concepts that can be mapped to the OpenTracing specification. However, two microservice systems (S1, S5) are planning to migrate to OpenTracing, as it has an active open-source community that provides various tools.

From the preceding analysis, it can be seen that OpenTracing specification should be the first choice for microservice systems considering its active open-source community that provides various tools.

4.1.3 Sampling

Logging brings extra overhead to services and log storage. Therefore, when the service traffic is heavy, log sampling is necessary for the tracing system. As the sampling occurs before the trace is restored, it is called head-based sampling. Among the eight surveyed microservice systems, the four bigger microservice systems (i.e., S1-S4) use head-based sampling. All of them combine two sampling strategies, i.e., fixed probability sampling and adaptive sampling. The former records logs for service invocations with a fixed probability, while the latter dynamically adjusts the probability to ensure that the number of logs produced in a period of time keeps stable. S4 implements an alarm-based adaptive sampling: when an alarm associated with a service instance is raised, the logger changes to log all the service invocations. For the other four microservice systems, which do not use head-based sampling, the explanation is that the logging overhead is neglectable due to the system scale and traffic.

A problem with head-based sampling is that it cannot consider the features of traces such as error state and latency. Therefore, it is usually combined with tail-based sampling, which samples traces in preprocessing.

We discuss tail-based sampling in Section 4.3.

4.2 Collection

Span logs generated by service instances are continuously collected by a centralized collector. The collection mechanism at the service instance side is usually implemented in the following two techniques.

- **Agent-based Log Collection**: An accompanying agent collects the logs of a service instance and pushes them to the collector, as shown in Fig. 5a.
- **Agent-less Log Collection**: A service instance itself sends the logs produced by it to the collector, as shown in Fig. 5b.

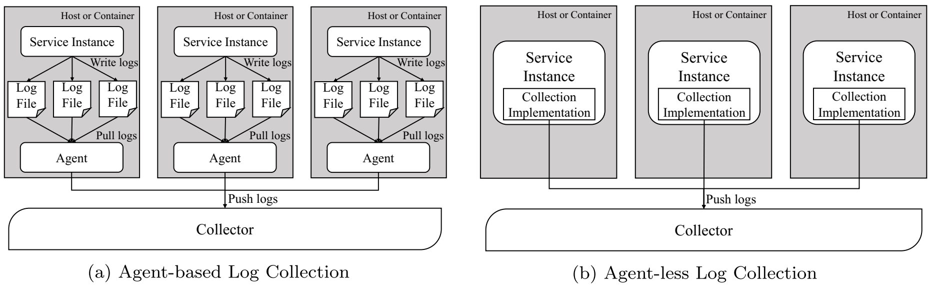


Fig. 5 Agent-based and Agent-less Log Collection

Using agent-based log collection, the developers do not need to consider log collection. Moreover, the agent can perform some local preprocessing, e.g., filtering and formatting logs, and flexibly adjust its collection strategy, e.g., pausing pushing logs when the available system resources are insufficient. The disadvantage of this technique lies in its complexity and the requirements of infrastructure support: each node in the cluster has to install the agent and there are requirements of additional efforts for assuring the operation of the agents. Agent-less log collection is easy to implement and has no requirement of infrastructure support, but requires the developers to embed the collection implementation in their code. Among our surveyed microservice systems, the six bigger systems (i.e., S1-S6) use agent-based log collection as they can afford the additional infrastructure support for the agents, while the other two use agent-less log collection.

4.3 Preprocessing

Preprocessing usually includes data formatting, metric aggregation, and tail-based sampling.

Data formatting transforms the logs produced by different services into the same format. Among the surveyed systems, S1 and S2 involve data formatting, as they include services using different technology stacks and their log formats are slightly different.

Metric aggregation computes aggregated metrics of traces from different dimensions. Among the surveyed microservice systems, S1, S2, S3, S4, and S7 perform metric aggregation in log preprocessing. They aggregate the traces from different dimensions such as URL, service name, and data center, and produce aggregated metrics such as Queries Per Second (QPS), average response time, and error rate. Note that the metrics that can be aggregated are those that are bounded to service requests and can be calculated from trace logs. Other metrics such as CPU usage and memory usage are collected in a way independent of distributed tracing and how to incorporate these metrics into trace analysis is still a challenge.

Tail-based sampling occurs in log preprocessing when all the information (e.g., spans and metrics) of a trace have been available. Therefore, tail-based sampling can make sampling decisions based on the error states and metrics of the traces. Among the surveyed microservice systems, S1, S2, and S4 perform tail-based sampling. For example, S1 selects traces with error tags or incomplete traces (where the traces may be broken by exceptions) in tail-based sampling.

4.4 Storage

After preprocessing, the trace data is persisted in storage for further analysis. Our study shows that it is common to combine multiple different strategies for trace data storage.

S1 uses a self-developed column-oriented database to store the traces and a distributed non-relational database (HBase) to store aggregated metrics and reports generated by preprocessing. S2 uses an open-source column-oriented database (Clickhouse) to store traces and a graph database (Neo4j) to store the topologies of traces. S3 uses HBase to store traces and a self-developed in-memory database to store aggregated metrics generated by preprocessing. Other systems (S4-S8) use ElasticSearch as the backend storage. But the interviewees of S4 say that they are considering to use HBase to store historical traces due to the increasing volume of trace data.

From the results, we can see that for large microservice systems efficient and scalable trace data storage is essential, while for small microservice systems a general search and analytics engine such as ElasticSearch is sufficient.

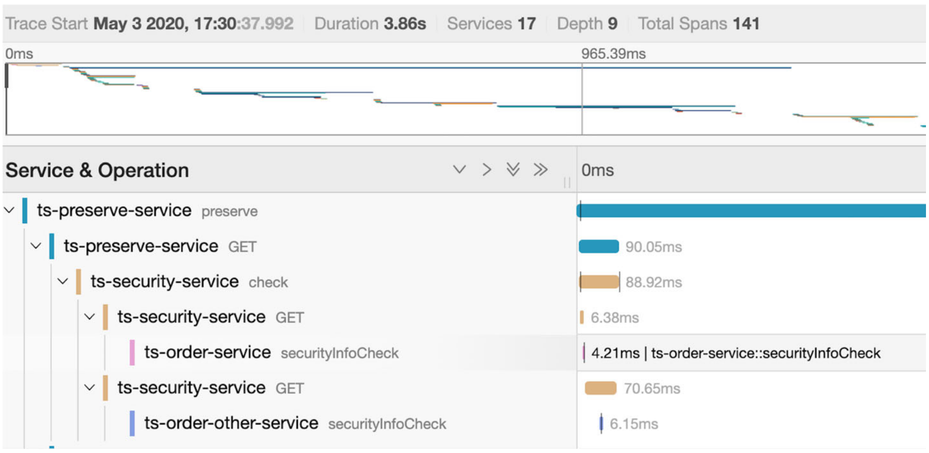
4.5 Analysis

An overview of the trace analysis applications is shown in Table 4. Operation engineers and developers rely on trace analysis for different purposes such as system understanding, anomaly detection, and problem diagnosis. They use various trace analysis applications provided by open-source tools (e.g., SkyWalking 2020; Zipkin 2020; Jaeger 2020) or customized tools developed by themselves. These applications combine span logs, application logs, and system metrics, and use different analysis techniques including visualization, statistics (i.e., statistical calculation of related metrics), and rule-based decision.

Timeline Analysis Timeline analysis visualizes the execution process and time consumption of a single trace. It uses a Gantt chart to show the nested service invocations (spans) and the duration of each invocation. The timeline analysis interface provided by Jaeger 2020 is

Table 4 Trace analysis application overview

Application	Purpose	Data	Technique	Applied in
Timeline Analysis	visualize the execution process and time consumption of a single trace for performance analysis	span log	visualization	S1, S2, S3, S4, S5, S6, S7, S8
Service Dependency Analysis	provide an overview of service dependencies at runtime by visualization	span log	statistics, visualization	S1, S2, S3, S4, S5, S6, S7
Aggregation Analysis	group traces from different dimensions and aggregate related metrics to provide an abstract view for traces	span log, application log, system metrics	statistics, visualization	S1, S2, S3, S4, S7
Root Cause Analysis	use trace analysis to locate the root causes of various kinds of faults	span log, application log, system metrics	statistics, visualization	S1, S2, S3, S4, S5, S6, S7, S8
Anomaly Detection	detect possible anomalies of service execution and generate timely alerts	span log	rule-based decision, visualization	S1, S2, S3, S4, S7


Trace Start May 3 2020, 17:30:37.992 | Duration 3.86s | Services 17 | Depth 9 | Total Spans 141

0ms 965.39ms

Service & Operation | 0ms
- ts-preserve-service preserve
  - ts-preserve-service GET | 90.05ms
    - ts-security-service check | 88.92ms
      - ts-security-service GET | 6.38ms
        - ts-order-service securityInfoCheck | 4.21ms | ts-order-service::securityInfoCheck
      - ts-security-service GET | 70.65ms
        - ts-order-other-service securityInfoCheck | 6.15ms

Fig. 6 Timeline Analysis in Jaeger

shown in Fig. 6, which depicts the trace data of TrainTicket[1] (Zhou et al. 2021), an open-source microservice benchmark. It describes the spans of a trace according to the order of invocation. Each span is depicted as a bar indicating its duration of time and is nested in its parent span. Some companies use additional visual information to better support the analysis. For example, the timeline analysis in S4 uses the darkness of the span color to indicate the time duration. The data used for timeline analysis includes trace spans and span duration. The used technique is visualization. Timeline analysis is usually used to narrow down the range of troubleshooting for time-related problems such as service invocation latency and timeout.

Service Dependency Analysis Service dependency analysis provides an overview of the service dependencies at runtime using flow-graph-based visualization.

The service dependency analysis interface provided by Jaeger (2020) is shown in Fig. 7, which depicts the service dependencies of TrainTicket. The interface describes the dependencies between services, and the numbers of service invocations are labeled on the edges. Some companies use additional visual information to highlight important metrics. For example, the service dependency analysis in S1, S3, and S4 uses the thickness of service dependency edges to indicate the frequency of service invocations (calculated by counting the number of invocations between two services in a given time period); the analysis in S4 uses the darkness of the service color to indicate the degree that other services depend on the current service (calculated by counting the number of invocations to a service in a given time period). In S1, S2, S3, S5, and S7, the service dependency analysis is extended to include various infrastructure services such as message queue, distributed cache, and database (e.g., MySQL and Mongo).

Service dependency analysis is usually used to understand the runtime architecture and service dependencies. As it highlights the upstream and downstream relationships among services, it can also be used to analyze error propagation chains. In some systems (e.g., S1), service dependency analysis is further used in runtime change impact analysis. Operation

[1] https://github.com/FudanSELab/train-ticket

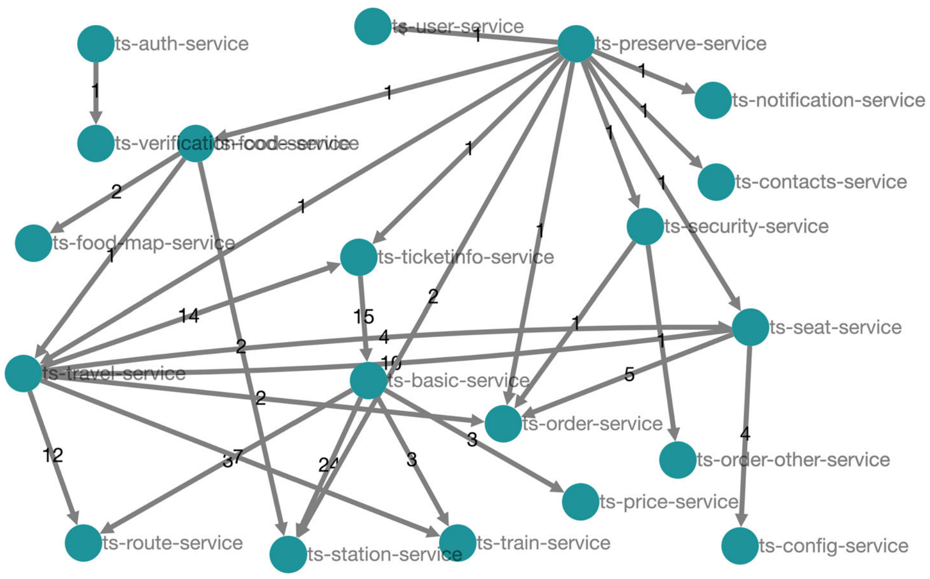


Fig. 7 Service Dependency Analysis in Jaeger

engineers compare the service dependencies before and after a release at runtime to understand the change impact and determine the minimal range of rollback in case of failure. In other systems (e.g., S7), service dependency analysis is used to observe the status of canary release, e.g., to examine the traffic of the new service versions.

Aggregation Analysis Aggregation analysis groups traces from different dimensions and computes aggregated metrics to provide an abstract view for traces. The traces with exactly the same set of services and the same invocation orders can be grouped into a path. The paths belonging to the same business scenario can be further grouped into a business flow. Along with the trace grouping, relevant metrics are aggregated. These metrics include not only service invocation metrics obtained from span logs (e.g., service invocation count and error rate), but also system metrics (e.g., CPU/memory consumption rate) obtained from system monitoring tools (e.g., Prometheus 2020) and application events (e.g., numbers of different types of exceptions) obtained from application logs. Besides general application events such as raised exceptions, application-specific events are collected and aggregated. For example, E-commerce systems (S1-S3) aggregate core business events such as successful payment; S4 aggregates VIP subscription events; S7 aggregates important financial events such as deposit changes. The system metrics are usually associated with traces based on timestamps. To associate application events with traces, microservice systems usually choose to embed trace IDs in application logs. The results of aggregation analysis can be visualized for developers and operation engineers to understand the business flows and architecture and discover potential problems.

Root Cause Analysis Root cause analysis uses trace analysis to locate the root causes of various kinds of faults such as functional failures and performance issues. Basic root cause analysis can be analyzing a single trace to learn the upstream and downstream relationships between services and identify error propagation chains. One can also compare different traces of the same type to understand the differences between failing and successful traces (Zhou et al. 2021). Advanced root cause analysis queries traces within a given time range and meeting certain conditions (e.g., involving invocations of specific services) and analyzes various aggregated metrics (e.g., error rate, average latency, and CPU consumption) to locate root causes. This analysis allows the analysis of various metrics in the context of service interactions. It requires efficient queries of trace data and the association of various internal or external metrics with traces. For example, S1, S2, S3, and S4 provide trace visualization that associates traces with various applications and system metrics such as different kinds of exceptions, and CPU and memory consumption in a given time range. This analysis can help developers and operation engineers to narrow down the range of troubleshooting, e.g., by determining whether the root cause lies in code defects or infrastructure failures.

Anomaly Detection Anomaly detection detects possible anomalies of service execution and generates timely alerts. It continuously analyzes trace metrics (e.g., latency) to identify possible anomalies according to predefined rules, e.g., jitter or exceeding the threshold. Moreover, it can report abnormal traces as the context for further troubleshooting. Based on the traces, one can check the upstream and downstream services of the anomaly location. For example, S4 provides anomaly detection for API availability and supports the corresponding trace context analysis.

From Table 4 and the preceding analysis, it can be seen that timeline analysis, service dependency analysis, and root cause analysis are basic analysis applications equipped by nearly all the companies using distributed tracing. This result can be explained from two aspects. First, these applications provide the basic capabilities required to diagnose functional and non-functional faults and understand the runtime behaviors of microservice systems. Second, these applications are provided by open-source tools such as Jaeger and thus are easy to equip.

In contrast, aggregation analysis and anomaly detection are often used in large microservice systems (e.g., S1-S4) for two main reasons. First, these systems are large and complex, and it is thus more difficult for the developers to achieve timely fault diagnosis using the basic analysis applications. Second, this analysis requires more advanced data processing and analysis such as stream data analysis, data mining, and machine leaning, which can be afforded by only large companies. Note that although root cause analysis is necessary for all the companies, it is implemented in different ways. Large companies often implement semi-automatic root cause analysis that uses more advanced techniques (e.g., data mining, machine learning) and incorporates logs and metrics into trace data analysis, while small companies often use basic analysis capabilities such as trace and dependency visualization and rely more on human expertise.

In some microservice systems, different applications are combined for a specific purpose. For example, S2 combines service dependency analysis with aggregation analysis for root cause analysis. It aggregates traces into paths and business flows, and then incorporates aggregated metrics such as error rate to identify possible error propagation chains. S1 and S3 combine anomaly detection with root cause analysis to automatically suggest candidate root causes for the detected anomalies.

4.6 Summary

Our survey indicates that industrial microservice tracing and analysis has widely followed the distributed tracing practices and developed various trace analysis applications for different purposes. Their tracing and analysis pipelines have a similar structure, which includes logging, collection, preprocessing, storage, and analysis. Large companies, especially Internet companies whose businesses are run as online services, may choose to develop and maintain their own distributed tracing systems due to the following two considerations. First, their large-scale microservice systems often have strict requirements on the reliability and performance of trace data processing, typically not satisfied by open-source solutions. Second, their large-scale microservice systems have a high demand on various customized trace analysis applications, typically not provided by open-source solutions. In contrast, small companies often choose to use the all-in-one solutions provided by open-source projects such as SkyWalking (2020), as they cannot afford a dedicated team for the development and maintenance of distributed tracing systems.

The implementations of the pipeline in different companies reflect the considerations of legacy services and the tradeoffs between a series of concerns. Large microservice systems often have a long history of evolution and include legacy services developed using outdated techniques. Therefore, a unified framework (e.g., tracing framework) may not work for all the services, and complementary techniques such as manual coding for logger implementation may be used. The tradeoff between the convenience for developers and infrastructure investment influences multiple design decisions. More advanced solutions such as tracing-framework-based logging and agent-based log collection relieve the developers from the implementation of distributed tracing, but require the investment on microservice infrastructures. Large companies usually choose to build the required infrastructures to improve efficiency and quality, while small companies usually choose to adopt simpler solutions to avoid building and maintaining additional infrastructures. Moreover, container-based deployment facilitates the application of more advanced solutions, as their components (e.g., agent, collector, preprocessor) can be deployed in containers and orchestrated and managed together with service instances by platforms such as Kubernetes.

The design decisions for logging include logger implementation, content and format, and sampling. Logger implementation focuses on the tradeoff between intrusive and non-intrusive implementation techniques. Tracing framework is more preferred, as it makes a good tradeoff between intrusive and non-intrusive logging. Manual coding is used as a complementary technique in large microservice systems to support legacy services and include application specific information in logs. The companies widely follow the log content and format defined by the OpenTracing specification. Although some companies use customized formats, their customized formats are based on similar concepts that can be mapped to the OpenTracing specification. Moreover, there is a trend of migrating to the OpenTracing specification to benefit from its active open-source community. The sampling strategies for logging (and also preprocessing) focus on the tradeoff between overhead and completeness. Sampling can alleviate the overhead to services and log storage while losing important traces for trace analysis tasks such as root cause analysis.

The design decisions for log collection focus on the tradeoff between agent-based and agent-less techniques. Agent-based log collection relieves the developers from the consideration of log collection and supports flexible customization of log collection strategy. Therefore, companies often choose to use agent-based log collection if they can afford the additional infrastructure support for the agents.

Preprocessing usually includes data formatting, metric aggregation, and tail-based sampling. Large microservice systems are more likely to include such preprocessing as they may have more complex technology stacks causing differences in log formats and a larger volume of trace data requiring more efficient data analysis.

Trace data storage is more challenging for large microservice systems, which have a large volume of trace data. They often combine different storage strategies such as relational database, column-oriented database, graph database, distributed non-relational database, and in-memory database to support a variety of highly efficient trace analysis.

Trace analysis has been widely used for understanding the runtime architectures and dependencies of microservice systems, and discovering and locating functional or non-functional faults. Visualization and statistic-based metrics are the most common means for trace analysis, while more advanced analysis techniques such as machine learning and data mining are seldom used. Some companies have tried to incorporate application logs and system metrics into trace analysis to provide a more comprehensive view for developers and operation engineers. However, the alignment between different monitoring data is difficult. The huge amount of trace data has been the main challenge for trace analysis. To mitigate the challenge, some companies have tried to provide flexible filtering and aggregation of trace data. In summary, most companies have established practices of preliminary trace analysis, but the application of intelligent analysis techniques is still in its infancy.

Service mesh has been regarded as an emerging solution for microservice tracing. It is a dedicated infrastructure layer for facilitating service-to-service communications, often using a sidecar proxy (Sidecar 2020). Service mesh manages traffic flows between services. Thus service developers do not need to care about the implementation of distributed tracing. However, none of our surveyed systems use service mesh for tracing, and the reasons include three aspects. First, the service mesh is still new, and the technology accumulation is insufficient. Second, service mesh has a high demand for the infrastructure. For example, service mesh implementations such as Istio (2020) require that the services are deployed on Kubernetes (2020). Third, the cost of migrating from the current tracing framework to service mesh is high.

5 Discussion

Distributed tracing is essential for a microservice system to acquire the required observability. Our survey indicates that microservice tracing and analysis is a new big data problem for software engineering. We have seen challenges and opportunities, which call for further research on the topics, as supported by our findings from different systems (see Table 5).

Adaptive Log Sampling The basic problem for log sampling strategy is how to keep a low overhead while ensuring that the required span logs are available for trace analysis. Anomalies and faults are rare, but at the same time, anomaly detection and root cause analysis are sensitive to the availability of relevant traces. Therefore, adaptive sampling is necessary to satisfy the requirements of anomaly detection and root cause analysis. The current adaptive sampling practices revealed by our study use only simple rules to adjust the sampling rate in

Table 5 Overview of challenges and opportunities

Challenges and opportunities	Supporting systems
Adaptive Log Sampling	S1, S2, S3, S4
Data Fusion for Trace Analysis	S1, S2, S3, S4, S5, S7
More Intelligent Trace Analysis	S1, S2, S3, S4
Business Intelligence by Trace Analysis	S1, S2

logging dynamically, and thus cannot ensure the capturing of valuable traces. Considering the nature of log generation and processing, it may be crucial to perform trace examination in preprocessing and build a feedback control loop between logging and preprocessing. In this way, preprocessing can continuously analyze the restored traces and estimate various properties such as likelihood of anomaly, relevance to root cause analysis, and coverage of different dimensions (e.g., business scenarios and data centers). These results are then fed back to the loggers of relevant service instances to enable them to adjust the sampling strategies accordingly.

Data Fusion for Trace Analysis Microservice faults may be rooted in environmental configurations, deployment, service interactions, and service implementation (Zhou et al. 2021). Incidents of systems and cloud infrastructure may also influence the traces of services. Therefore, trace analysis for anomaly detection and root cause analysis requires to combine other data such as environmental configurations, system metrics, and application logs. The data fusion is challenging as different types of data are specified from various dimensions and generated at different frequencies. For example, resource metrics (e.g., CPU/memory consumption rates) are generated at fixed intervals (e.g., every one minute). Thus we can only roughly associate a trace with the metrics with the closest timestamps. Moreover, the combined analysis of different types of data is also challenging, as there are complicated confounding effects of environmental factors (e.g., environmental configurations and resource consumptions) in anomaly detection and root cause analysis.

More Intelligent Trace Analysis We notice that some companies have established successful practices of AIOps (artificial intelligence for IT operations) at the system and infrastructure level, enabling them to predict incidents of networks and servers. These incidents have relatively limited types (e.g., disk damage, bad memory, network latency) and involve limited nodes, and thus are easier to simulate and predict. In contrast, intelligent trace analysis is more difficult, as microservice faults can be rooted in many different sources. Moreover, microservice traces may involve a lot of services and even multiple data centers, and there is a huge number of traces produced every second. These characteristics make microservice trace analysis more challenging. Our study reveals some additional difficulties in practice. First, the quality of the trace data is low and problems such as incomplete or erroneous traces are popular. This difficulty often occurs in large systems that include a large number of services developed by different teams, in different time periods, and using different technology stacks. Second, trace data annotation is often expensive and challenging. The feedback in our survey indicates that the annotation is challenging as it is hard to define the classification of faults and identify problematic traces for annotation. Moreover, trace data annotation is greatly influenced by the ever-changing architecture of a microservice system. It is desired that traditional debugging techniques can be further developed and applied in microservice trace analysis to facilitate anomaly detection and root cause analysis. In addition, unsupervised, semi-supervised, and weakly supervised learning techniques may be important due to the lack of trace data annotation.

Business Intelligence by Trace Analysis The feedback in our survey prompts us to think about trace analysis from a broader and newer perspective of value delivery. Trace analysis can implement business intelligence and deliver value to a broader range of stakeholders. A trace records the execution process of a request through a number of services, nodes, and even data centers. Thus it can associate the execution records of different levels such as resource consumptions and business transactions to enable the dig of value for different stakeholders. For example, we can analyze the contributions of different nodes and data centers that are involved in business transactions to analyze the return on investment of cloud infrastructure. Therefore, it may be a promising direction that distributed tracing and trace analysis are used to associate execution records of different levels to deliver value to different stakeholders.

6 Related Work

More than ten years back, distributed tracing systems (e.g., Magpie (Barham et al. 2003) and XTrace (Fonseca et al. 2007)) are proposed and focused on performance analysis of physical machines and operating systems in detail. However, these systems do not apply to modern complicated distributed systems, such as microservice systems. In 2010, Google proposed Dapper (Sigelman et al. 2010), a production tracing infrastructure for distributed systems. Dapper first adopts some design choices such as the use of sampling and restrictive instrumentation to make it applicable to complicated distributed systems consisting of thousands of machines. Ever since, many other industrial companies (e.g., eBay 2020; Netflix 2020) have built their own distributed tracing systems.

Distributed tracing is also a hot topic in the open-source community. Nowadays, there are many active projects for distributed tracing implementation, e.g., Pinpoint (2020), Zipkin (2020), Jaeger 2020, SkyWalking (2020). As distributed tracing becomes widespread in complicated distributed systems, specifications such as OpenTracing (2020), OpenCensus (2020), and OpenTelemetry (2020) have emerged. They allow developers to instrument their application code without locking them into any one particular framework.

Distributed tracing is widely used on anomaly detection (Zhou et al. 2019; Barham et al. 2004; Chen et al. 2004; Mace et al. 2015), diagnosis of steady-state problems (Chen et al. 2004; Fonseca et al. 2010; Fonseca et al. 2007; Reynolds et al. 2006; Sambasivan et al. 2011; Sigelman et al. 2010), profiling (Chanda et al. 2007; Sigelman et al. 2010), resource-usage attribution (Fonseca et al. 2008; Thereska et al. 2006), and evaluation of microservice architecture (Engel et al. 2018; Bogner et al. 2019). Mace et al. (2015) launched a project called Pivot Tracing to help diagnose unanticipated problems in distributed applications on the fly. Pham et al. (2017) introduced an approach to automating failure diagnosis in distributed systems by reconstructing and comparing execution traces. Zhou et al. (2021) proposed a visualization analysis to compare two different traces for service- and state-level debugging. Zhou et al. (2019) proposed an approach to predicting latent error and localizing fault for microservice applications by learning from system trace logs. Lenarduzzi and Panichella (2021) used distributed tracing to debug serverless-based microservices. Engel et al. (2018) proposed an evaluation approach for microservice architectures using trace data. While the preceding previous research focused on specific aspects of trace analysis, our research provides a comprehensive and detailed introduction to trace analysis in the industry, and summarizes the challenges and problems that still exist.

Quite a few industry surveys were conducted for microservice architecture. Di Francesco et al. (2018) performed an empirical study on migration practices toward the adoption of microservices in industry. Haselböck et al. (2018) investigated the importance of different areas of microservice design. Ten microservice experts were interviewed to understand the importance and relevance of the microservices design areas. Justus *et al.* (Bogner et al. 2019) contributed a qualitative study with insights into industry adoption and implementation of microservices by analyzing 14 service-based systems during 17 interviews. Zhang et al. (2019) carried out a series of industrial interviews with 13 different types of companies to investigate the gap between the ideal visions and real industrial practices along with the benefits of microservices from the industrial experiences. Zhou et al. (2021) conducted an industry survey on microservices, reporting on the visualization practices of distributed tracing in the industry. The preceding previous research comprehensively investigated the practice of microservice systems in the industry but did not discuss distributed tracing in depth. Our research comprehensively introduces the practices and applications of distributed tracing in industrial microservice systems in detail, and summarizes the existing challenges and problems. There were some surveys for studying logging practices for large systems, e.g., Yuan et al. (2012). To the best of our knowledge, our research is the very first empirical study on distributed tracing.

Dynamic tracing is also used in the analysis of a single monolithic system. Previous related research (Chen et al. 2017; Taibi and Systä 2019) used dynamic tracing to collect and mine the execution processes of a single monolithic system to help the decisions of microservice decomposition and migration.

7 Conclusion

In this article, we have presented an industrial survey on microservice tracing and analysis through interviewing developers and operation engineers of microservice systems from ten companies. Our survey indicates that distributed tracing and analysis has been an important part of the infrastructure for industrial microservice systems, commonly equipped with a tracing and analysis pipeline (which includes logging, collection, preprocessing, storage, and analysis). However, the implementations of the pipeline in different companies differ in its architecture and constituent components. Large companies tend to develop and maintain their own distributed tracing systems, while small companies often choose to use the all-in-one solutions provided by open-source projects. The implementations reflect the considerations of legacy services and the tradeoffs between a series of concerns.

Our survey also indicates that microservice tracing and analysis is a new big data problem for software engineering, and its practices breed new challenges and opportunities, including adaptive log sampling, data fusion for trace analysis, more intelligent trace analysis, and business intelligence by trace analysis. In particular, most companies have established practices of preliminary trace analysis, but the application of intelligent analysis techniques is still in its infancy. The quality of the trace data, the lack of trace data annotation, and the ever-changing nature of microservice architecture are the main difficulties.

In future work, we plan to adapt traditional software engineering techniques such as program analysis, reverse engineering, and debugging to facilitate microservice tracing and analysis.

Acknowledgements This work was supported by the National Key Research and Development Program of China under Grant No. 2018YFB1004803.

References

Alibabacloud.Com: Alibabacloud (2020). https://www.alibabacloud.com/help/doc-detail/42781.htm?spm=a2c63.l28256.b99.2.7def74893VnsEp

Aws.Amazon.Com: AWS (2020). https://aws.amazon.com/xray/

Barham P., Donnelly A., Isaacs R., Mortier R. (2004) Using Magpie for request extraction and workload modelling. In: 6Th USENIX symposium on operating systems design and implementation (OSDI), pp 259–272

Barham P., Isaacs R., Mortier R., Narayanan D. (2003) Magpie: Online modelling and performance-aware systems. In: 9Th workshop on hot topics in operating systems (hotOS), pp 85–90

Bogner J., Fritzsch J., Wagner S., Zimmermann A. (2019) Microservices in industry: Insights into technologies, characteristics, and software quality. In: IEEE International conference on software architecture companion (ICSA-c), pp 187–195

Bogner J., Schlinger S., Wagner S., Zimmermann A. (2019) A modular approach to calculate service-based maintainability metrics from runtime data of microservices. In: 20Th international conference on product-focused software process improvement (PROFES), pp 489–496

Brown K (2021) Cloud adoption patterns. https://kgb1001001.github.io/cloudadoptionpatterns/Cloud-Native-DevOps/Correlation-ID/

Chanda A., Cox A. L., Zwaenepoel W. (2007) Whodunit: Transactional profiling for multi-tier applications. In: 2Nd ACM SIGOPS/eurosys european conference on computer systems (eurosys), pp 17–30

Chen M. Y., Accardi A. J., Kiciman E., Patterson D. A., Fox A., Brewer E. A. (2004) Path-based failure and evolution management. In: 1St symposium on networked systems design and implementation (NSDI), pp 309–322

Chen R., Li S., Li Z. (2017) From monolith to microservices: A dataflow-driven approach. In: 24Th asia-pacific software engineering conference (APSEC), pp 466–475

Developer.Ebay.Com: Ebay developers program (2020). https://developer.ebay.com/

Di Francesco P., Lago P., Malavolta I. (2018) Migrating towards microservice architectures: an industrial survey. In: IEEE International conference on software architecture (ICSA), pp 29–39

Dynatrace.Cn: Dynatrace (2020). https://www.dynatrace.cn/

Elasticsearch.Com: Elasticsearch (2020). https://www.elastic.co/products/elasticsearch

Engel T., Langermeier M., Bauer B., Hofmann A. (2018) Evaluation of microservice architectures: a metric and tool-based approach. In: Information systems in the big data era, pp 74–89

Flink.com: Apache flink (2021). https://flink.apache.org/

Fonseca R., Dutta P., Levis P., Stoica I. (2008) Quanto: Tracking energy in networked embedded systems. In: 8Th USENIX symposium on operating systems design and implementation (OSDI), pp 323–338

Fonseca R., Freedman M., Porter G. (2010) Experiences with tracing causality in networked services. In: 1St internet network management workshop/workshop on research on enterprise monitoring

Fonseca R., Porter G., Katz R. H., Shenker S., Stoica I. (2007) X-trace: A pervasive network tracing framework. In: 4Th symposium on networked systems design and implementation (NSDI), pp 271–284

Francesco P. D., Malavolta I., Lago P. (2017) Research on architecting microservices: trends, focus, and potential for industrial adoption. In: IEEE International conference on software architecture (ICSA), pp 21–30

Haselböck S., Weinreich R., Buchgeher G. (2018) An expert interview study on areas of microservice design. In: 11Th IEEE conference on service-oriented computing and applications (SOCA), pp 137–144

Htrace.Org: Htrace (2020). http://htrace.org/

Istio.Io: Istio (2020). https://istio.io/

Istio.Io: Sidecar (2020). https://istio.io/docs/reference/config/networking/sidecar/

Jaegertracing.Io: Jaegertracing (2020). https://www.jaegertracing.io/

Kafka.Com: Apache Kafka (2021). http://kafka.apache.org/

Kaldor J., Mace J., Bejda M., Gao E., Kuropatwa W., O'Neill J., Ong K. W., Schaller B., Shan P., Viscomi B., Venkataraman V., Veeraraghavan K., Song Y. J. (2017) Canopy: an end-to-end performance tracing and analysis system. In: 26Th symposium on operating systems principles (SOSP), pp 34–50

Kibana.Com: Kibana (2020). https://www.elastic.co/products/kibana

Kubernetes.Io: Kubernetes (2020). https://kubernetes.io/

Kvale S. (2008) Doing interviews. Sage

Lenarduzzi V., Panichella A. (2021) Serverless testing: Tool vendors' and experts' points of view. IEEE Softw 38(1):54–60

Lewis J, Fowler M (2020) Microservices. https://martinfowler.com/articles/microservices.html

Logging.Apache.Org: Apache (2020). https://logging.apache.org/log4j/2.x/

Logstash.Com: Logstash (2020). https://www.elastic.co/products/logstash

Mace J., Roelke R., Fonseca R. (2015) Pivot tracing: Dynamic causal monitoring for distributed systems. In: 25Th symposium on operating systems principles (SOSP), pp 378–393

Naver.Github.Io: Dynatrace (2020). http://naver.github.io/pinpoint/

Netflix.Com: Netflix (2014). https://www.netflix.com/

Newman S. (2015) Building microservices - designing fine-grained systems, 1st edn. O'Reilly

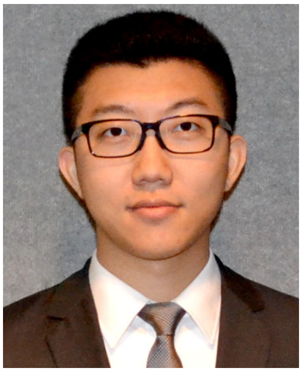
Opencensus.Io: Opencensus (2020). https://opencensus.io/
Opentelemetry.Io: Opentelemetry (2020). https://opentelemetry.io/
Opentracing.Io: Opentracing (2020). https://opentracing.io/
Pham C., Wang L., Tak B., Baset S., Tang C., Kalbarczyk Z. T., Iyer R. K. (2017) Failure diagnosis for distributed systems using targeted fault injection. IEEE Trans Parallel Distrib Syst (TPDS) 28(2):503–516
Prometheus.Io: Prometheus (2020). https://prometheus.io/
Qos.ch: Qos (2020). http://logback.qos.ch/
Reynolds P., Killian C., Wiener J., Mogul J., Shah M., Vahdat A. (2006) Pip: Detecting the unexpected in distributed systems. In: 3Rd symposium on networked systems design and implementation (NSDI), pp 115–128
Richardson C. (2019) Microservices patterns: With examples in java. Manning Publications Co
Sambasivan R. R., Zheng A. X., Rosa M. D., Krevat E., Whitman S., Stroucken M., Wang W., Xu L., Ganger G. R. (2011) Diagnosing performance changes by comparing request flows. In: 8Th symposium on networked systems design and implementation (NSDI)
Sigelman B. H., Barroso L. A., Burrows M., Stephenson P., Plakal M., Beaver D., Jaspan S., Shanbhag C. (2010) Dapper a large-scale distributed systems tracing infrastructure
Skywalking.Org: Apache Skywalking (2020). https://skywalking.apache.org/
Splunk.Com: Splunk observability (2021). https://www.splunk.com/
Sridharan C. (2018) Distributed systems observability: a guide to building robust systems. O'Reilly Media, Inc
Strauss A., Corbin J. (1990) Open coding. In: Strauss A, Corbin J (eds) Basics of qualitative research: Grounded theory procedures and techniques. Sage Publications, London, pp 101-121
Taibi D., Systä K (2019) From monolithic systems to microservices: A decomposition framework based on process mining. In: 9Th international conference on cloud computing and services science (CLOSER), pp 153–164
Thereska E., Salmon B., Strunk J. D., Wachs M., Abd-el-malek M, Hernandez J. L, Ganger G. R (2006) Stardust: tracking activity in a distributed storage system. In: ACM SIGMETRICS Joint international conference on measurement and modeling of computer systems (SIGMETRICS), pp 3–14
Wikipedia.Org: Wikipedia (2020). https://en.wikipedia.org/wiki/Log_file
Yuan D., Park S., Zhou Y. (2012) Characterizing logging practices in open-source software. In: 34Th international conference on software engineering (ICSE), pp 102–112
Zhang H., Li S., Jia Z., Zhong C., Zhang C. (2019) Microservice architecture in reality: an industrial inquiry. In: IEEE International conference on software architecture (ICSA), pp 51–60
Zhou X., Peng X., Xie T., Sun J., Ji C., Li W., Ding D. (2021) Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. IEEE Trans Softw Eng (TSE) 47(2):243–260
Zhou X., Peng X., Xie T., Sun J., Ji C., Liu D., Xiang Q., He C. (2019) Latent error prediction and fault localization for microservice applications by learning from system trace logs. In: 27Th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering (ESEC/SIGSOFT FSE), pp 683–694
Zhou X., Peng X., Xie T., Sun J., Li W., Ji C., Ding D. (2018) Delta debugging microservice systems. In: 33Rd ACM/IEEE international conference on automated software engineering (ASE), pp 802–807
Zipkin.Io: Zipkin (2020). https://zipkin.io/

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Bowen Li is a master student of the School of Computer Science, Fudan University, China. He received his bachelor's degree from Nanjing University in 2019. His research interests include cloud-native software and AIOps.

Xin Peng received the bachelor's and PhD degrees in computer science from Fudan University, in 2001 and 2006, respectively. He is a professor of the School of Computer Science, Fudan University, China. His research interests include data-driven intelligent software development, cloud-native software and AIOps, software engineering for AI and cyber-physical-social Systems. His work won the ICSM 2011 Best Paper Award, the ACM SIGSOFT Distinguished Paper Award at ASE 2018, the IEEE TCSE Distinguished Paper Awards at ICSME 2018/2019/2020, and the IEEE Transactions on Software Engineering 2018 Best Paper Award.

Qilin Xiang is a master student of the School of Computer Science, Fudan University, China. He received his bachelor's degree from University of Electronic Science and Technology of China in 2018. His research interests include cloud-native software and AIOps.



Hanzhang Wang is an applied researcher at eBay. He is leading a team of developers and researchers working on AIOps and AI4SE solutions. His recent research interests include intelligent observability, AIOps, RCA, SE, ML and graph algorithms. He has multiple publications in top venues, including FSE, ASE, VLDB, TSC, CIKM, etc. The research outcomes have been transferred into multiple production products for anomaly detection, root cause analysis, developer velocity, etc. Hanzhang also serves as the company-wide university parentship program manager. He received Ph.D. degree in computing science from University of Michigan and joined eBay in 2018.

Tao Xie is a Chair Professor in the Department of Computer Science and Technology at Peking University, Beijing, China, and Vice Director of Key Lab of High Confidence Software Technologies (PKU), Ministry of Education. He received an NSF CAREER Award, ACM SIGSOFT Distinguished Service Award, IEEE CS TCSE Distinguished Service Award, ASE 2021 Most Influential Paper Award, and various industrial faculty awards and distinguished/best paper awards. He is a co-Editor-in-Chief of the Wiley journal of Software Testing, Verification and Reliability (STVR). He served as the ISSTA 2015 Program Chair, Tapia 2017/2018 Program/General Chair, and an ICSE 2021 Program Co-Chair. He was selected by Lero as a David Lorge Parnas Fellow in 2019. He was selected as an ACM Distinguished Scientist in 2015, an IEEE Fellow in 2018, and an AAAS Fellow in 2019.

Jun Sun is currently an associate professor at Singapore Management University (SMU). He received Bachelor and PhD degrees in computing science from National University of Singapore (NUS) in 2002 and 2006. In 2007, he received the prestigious LEE KUAN YEW postdoctoral fellowship. He has been a faculty member since 2010 and was a visiting scholar at MIT from 2011-2012. Jun's research interests include software engineering, cyber-security and formal methods.

Xuanzhe Liu is now an associate professor in Peking University. His recent interests include distributed systems, software engineering, machine learning, etc. He has published over 70 referred papers in prestigious conferences like WWW, ICSE, FSE MobiCom, MobiSys, etc., and journals like ACM TOSEM/TOIS and IEEE TSE/TMC, etc. He received various Best Paper Awards from WWW 2019, ICSS 2010, CyberC 2009, etc. He was recognized as the Young Scientist Award that was co-awarded by China Computer Federation (CCF) and IEEE Computer Society and the Rising Star Award of IEEE Technical Committee on Services Computing.