Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study

Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding

Abstract—The complexity and dynamism of microservice systems pose unique challenges to a variety of software engineering tasks such as fault analysis and debugging. In spite of the prevalence and importance of microservices in industry, there is limited research on the fault analysis and debugging of microservice systems. To fill this gap, we conduct an industrial survey to learn typical faults of microservice systems, current practice of debugging, and the challenges faced by developers in practice. We then develop a medium-size benchmark microservice system (being the largest and most complex open source microservice system within our knowledge) and replicate 22 industrial fault cases on it. Based on the benchmark system and the replicated fault cases, we conduct an empirical study to investigate the effectiveness of existing industrial debugging practices and whether they can be further improved by introducing the state-of-the-art tracing and visualization techniques for distributed systems. The results show that the current industrial practices of microservice debugging can be improved by employing proper tracing and visualization techniques and strategies. Our findings also suggest that there is a strong need for more intelligent trace analysis and visualization, e.g., by combining trace visualization and improved fault localization, and employing data-driven and learning-based recommendation for guided visual exploration and comparison of traces.

Index Terms—microservices, fault localization, tracing, visualization, debugging

1 Introduction

Microservice architecture [1] is *an architectural style and approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API*. Microservice architecture allows each microservice to be independently developed, deployed, upgraded, and scaled. Thus, it is particularly suitable for systems running on cloud infrastructures and require frequent updating and scaling of their components.

Nowadays, more and more companies have chosen to migrate from the so-called monolithic architecture to microservice architecture [2], [3]. Their core business systems are increasingly built based on microservice architecture. Typically a large-scale microservice system can include hundreds to thousands of microservices. For example, Netflix's online service system [4] uses about 500+ microservices and handles about two billion API requests every day [5]; Tencent's WeChat system [6] accommodates more than 3,000 services running on over 20,000 machines [7].

A microservice system is complicated due to the extremely small grained and complex interactions of its microservices and the complex configurations of the runtime environments. The execution of a microservice system may involve a huge number of microservice interactions. Most of these interactions are asynchronous and involve complex invocation chains. For example, Netflix's online service system involves 5 billion service invocations per day and 99.7% of them are internal (most are microservice invocations); Amazon.com makes 100-150 microservice invocations to build a page [8]. The situation is further complicated by the dynamic nature of microservices. A microservice can have several to thousands of physical instances running on different containers and managed by a microservice discovery service (e.g., the service discovery component of Docker swarm). The instances can be dynamically created or destroyed according to the scaling requirements at runtime, and the invocations of the same microservice in a trace may be accomplished by different instances. Therefore, there is a strong need to address architectural challenges such as dealing with asynchronous communication, cascading failures, data consistency problems, discovery, and authentication of microservices [9].

The complexity and dynamism of microservice systems pose great and unique challenges to debugging, as the developers are required to reason about the concurrent behaviors of different microservices and understand the interaction topology of the whole system. A basic and effective way for understanding and debugging distributed systems is tracing and visualizing system executions [10]. However, microservice systems are much more complex and dynamic than traditional distributed systems. For example, there lacks a natural correspondence between microservices and system nodes in distributed systems, as microservice instances can be dynamically created and destroyed. There-

- *X. Peng is the corresponding author.*
- *X. Zhou, X. Peng, C. Ji, W. Li, and D. Ding are with the School of Computer Science and the Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China, and Shanghai Institute of Intelligent Electronics & Systems, China.*
- *T. Xie is with the University of Illinois at Urbana-Champaign, USA.*
- *J. Sun is with the Singapore University of Technology and Design, Singapore.*

fore, it is not clear whether or how well the state-of-the-art debugging visualization tools for distributed systems can be used for microservice systems.

In spite of the prevalence and importance of microservices in industry, there exists limited research on the subject, with only a few papers on microservices in the software engineering research community, and even fewer in major conferences. The existing research focuses on a wide range of topics about microservices, including design [11], testing [3], [12], [13], [14], deployment [15], [16], [17], verification [18], composition [19], architecture recovery [20], legacy migration [21], and runtime adaptation [22]. There exists little research on the fault analysis and debugging of microservice systems. Moreover, the existing research on microservices is usually based on small systems with few microservices (e.g., 5 microservices or fewer [2]). Such lack of non-trivial open source benchmark microservice systems results in a gap between what the research community can produce and what the industrial practices really need. There have been appeals that practitioners and researchers develop and share a common microservice infrastructure that can emulate the production environments of typical microservice applications for more repeatable and industry-focused empirical studies [23], [24].

To fill this gap and pursue practice-relevant research on microservices, we conduct an industrial survey on fault analysis of typical faults of microservice systems, current practice of debugging, and the challenges faced by the developers. Our survey shows that the current techniques used in practice are limited and the developers face great challenges in microservice debugging. We then conduct an empirical study to further investigate the effectiveness of existing industrial debugging practices and whether the practices can be facilitated by state-of-the-art debugging visualization tools.

To enable our study and also provide a valuable practice-reflecting benchmark for the broad research community, we develop a medium-size benchmark microservice system named TrainTicket [25]. Within our knowledge, our system is the largest and most complex open source microservice system. Upon the system, we replicate the 22 representative fault cases collected in the industrial survey. Based on the benchmark system and replicated fault cases, we empirically evaluate the effectiveness of execution tracing and visualization for microservice debugging by extending a state-of-the-art debugging visualization tool [10] for distributed systems. Based on the study results, we summarize our findings and suggest directions for future research.

In this work, we make the following main contributions:

- We conduct a survey on industrial microservice systems and report the fault-analysis results about typical faults, current practice of debugging, and the challenges faced by the developers.
- We develop a medium-size benchmark microservice system (being the largest and most complex open source microservice system within our knowledge) and replicate 22 representative fault cases upon it. The system and the replicated fault cases can be used as a benchmark for the research community to further conduct practice-relevant research on microservice fault analysis and debugging, and potentially other practice-relevant research on microservices.
- We experimentally evaluate the effectiveness of execution tracing and visualization for microservice debugging and propose a number of visualization analysis strategies for microservice debugging.

In this work we also extend the benchmark system presented in our earlier 2-page poster paper [26] by introducing more microservices (from 24 to 41) and characteristics (e.g., more languages, interaction modes). The replicated fault cases have been released as an open-source project [27], which can be easily integrated into the benchmark system [25]. The details of our industrial survey and empirical study (along with the source code of our open source benchmark system and replicated faults) can be found in our replication package [28].

The rest of the article is structured as follows. Section 2 presents background knowledge of microservice architecture. Section 3 describes the industrial survey, including the process and the results. Section 4 introduces the benchmark system and the 22 replicated fault cases. Section 5 presents the effectiveness evaluation of execution tracing and visualization based on the replicated fault cases and discusses our observations and suggestions. Section 6 discusses threats to validity. Section 7 reviews related work. Section 8 concludes the paper and outlines future work.

2 BACKGROUND

Microservice architecture arises from the broader area of Service Oriented Architecture (SOA) with a focus on componentization of small lightweight microservices, application of agile and DevOps practices, decentralized data management and governance among microservices [2]. With the migration from monolithic architecture to microservice architecture, architectural complexity moves from the code based to the interactions of microservices. The interactions among different microservices must be implemented using network communication. Microservice invocations can be *synchronous* or *asynchronous*. Synchronous invocations are considered harmful due to the multiplicative effect of downtime [9]. Asynchronous invocations can be implemented by asynchronous REST invocations or using message queues. The former provides better performance whereas the latter provides better reliability. As a user request usually involves a large number of microservice invocations and each microservice may fail, the microservices need to be designed accordingly, i.e., taking possible failures of microservice invocations into account.

Microservice architecture is supported by a series of infrastructure systems and techniques. Microservice development frameworks such as Spring Boot [29] and Dubbo [30] facilitate the development of microservice systems by providing common functionalities such as REST client, database integration, externalized configuration, and caching. Microservice systems widely employ container (e.g., Docker [31]) based deployment for portability, flexibility, efficiency, and speed [9]. Microservice containers can be organized and managed by clusters with configuration management, service discovery, service registry, load balancing by using runtime infrastructure frameworks such as

Spring Cloud [32], Mesos [33], Kubernetes [34], and Docker Swarm [35].

The unique characteristics of microservices pose challenges to existing debugging techniques. Existing debugging techniques are designed based on setting up breakpoints, manual inspection of intermediate program states, and profiling. However, these techniques are ineffective for microservices. For instance, due to the high degree of concurrency, the same breakpoint might be reached through very different executions resulting in different intermediate program states. Furthermore, a microservice system contains many asynchronous processes, which requires tracing multiple breakpoints across different processes; such tracing is considerably more challenging than debugging monolithic systems. Besides inspecting intermediate program states, it is equally, if not more, important to comprehend how microservices interact with each other for debugging. Profiling similarly becomes more complicated due to the high dynamism of microservices.

In addition, existing fault localization techniques [36] are ineffective for microservices. Program-slicing-based fault localization [37] works by identifying program statements that are irrelevant to the faulty statement and allowing the developers to investigate the fault based on the remaining statements. Program slicing for microservices is complicated since we must slice through many processes considering different interleavings of the processes. Spectrum-based fault localization [38] computes the suspiciousness of every statement using information such as how many times it is executed in passing test executions or failed test executions, and ranks the statements accordingly so that the developers can focus on the highly suspicious ones. There is no evidence that such techniques work for highly concurrent and dynamic systems such as microservices. Similarly related fault localization techniques are designed mainly for sequential programs, such as statistic-based fault localization [39], and machine-learning-based ones [40], [41].

In recent years, fault localization has been extended to concurrent programs [42], [43], [44] and distributed systems [45], [46], [47]. Both groups of work start with logging thread (and node) level execution information and then locate faults using existing techniques. Applying such techniques to microservices is highly non-trivial since the container instances in microservices are constantly changing, causing difficulty in log checking and overly fragmented logs.

3 Industrial Survey

In order to precisely understand the industry needs, we start with an industrial survey and then proceed with the collection of typical fault cases and the understanding of the current practice on microservice debugging.

3.1 Participants and Process

We identify an initial set of candidates for the survey from the local technical community who have talked about microservice in industrial conferences or online posts (e.g., articles, blogs). These candidates further recommend more candidates (e.g., their colleagues). Among these candidates, we select and invite 46 engineers for interview based on the following criteria. The candidate must have more than 6 years' experience of industrial software development and more than 3 years' experience of microservice development.

Among the invited engineers, 16 of them accept the invitation. The 16 participants are from 12 companies and their feedback is based on 13 microservice systems that they are working on or have worked on. These participants have a broad representation of different types of companies (i.e., traditional IT companies, Internet companies, and non-IT companies), different types of microservice systems (Internet services, enterprise systems) of different scales (50 to more than 1000 microservices), and different roles in development (technical roles and managerial roles). The information of the participants and subject systems are listed in Table 1, including the company (C.), the participant (P.), the subject system, the number of microservices (#S.), and the position of the participant.

Among the 12 companies, C1, C6, C7, C8, C11, and C12 are leading traditional IT companies, of which C1 and C8 are Fortune 500 companies; C3, C4, C5, and C10 are leading Internet companies; C2 and C9 are non-IT companies. The 13 subject systems can be categorized into two types. One type is Internet microservices that serve consumers via the Internet, including A3, A4, A5, A6, A10, and A11. The other type is enterprise systems that serve company employees, including A1, A2, A7, A8, A9, A12, and A13. The number of microservices in these systems ranges from 50 to more than 1000, with a majority of them involving about 100-200 microservices. The 16 participants take different positions in their respective companies. Among these positions, Junior Software Engineer, Staff Software Engineer, Senior Software Engineer, and Architect are technical positions; and Manager is a managerial position that manages the development process and project schedule.

We conduct a face-to-face interview with each of the participants. The participant is first asked to recall a microservice system that he/she is the most familiar with and provide his/her subsequent feedback based on the system. The participant introduces the subject system and the role that he/she takes in the system-development project. Then we interview and discuss with the participant around the following questions:

- Why does your company choose to apply the microservice architecture in this system? Is the system migrated from an existing monolithic system or developed as a new system?
- How does your team design the system? For example, how does your team determine the partitioning of microservices?
- What kinds of techniques and what programming languages are used to develop the system?
- What challenges does your team face during the maintenance of the system?

Afterwards, the participant is asked to recall those fault cases that he/she has handled. For each fault case, the participant is asked to describe the fault and answer the following questions:

- What is the symptom of the fault and how can it be reproduced?

TABLE 1
Survey Participants and Subject Systems

C.	P.	Subject System	#S.	Position
C1	P1	A1: online meeting system	50+	Staff Software Engineer
	P2	A2: collaborative translation system	100+	Senior Software Engineer
C2	P3	A3: personal financial system	100+	Manager
C3	P4	A4: message notification system	80+	Staff Software Engineer
C4	P5	A5: mobile payment system	200+	Architect
C5	P6	A6: travel assistance system	1000+	Senior Software Engineer
C6	P7	A7: OA (Office Automation) system	100+	Manager
C7	P8	A8: product data management system	200+	Architect
	P9	A8: product data management system	200+	Senior Software Engineer
C8	P10	A9: price management system	100+	Manager
	P11	A9: price management system	100+	Senior Software Engineer
C9	P12	A10: electronic banking system	200+	Senior Software Engineer
C10	P13	A11: online retail system	100+	Senior Software Engineer
C11	P14	A12: BPM system	60+	Staff Software Engineer
C12	P15	A13: enterprise wiki system	200+	Senior Software Engineer
	P16	A13: enterprise wiki system	200+	Senior Software Engineer

- What is the root cause of the fault and how many microservices are involved?
- What is the process of debugging? How much time is spent on debugging and what techniques are used?

After the interview, whenever necessary, we conduct follow-up communication with the participants via emails or phone calls to clarify some details.

3.2 General Practice

Our survey shows that most of these companies, not only the Internet companies but also the traditional IT companies, have adopted microservice architecture to a certain degree. Independent development and deployment as well as diversity in development techniques are the main reasons for adopting microservice architecture. Among the 13 surveyed systems, 6 adopt microservice architecture by migrating from existing monolithic systems, while the remaining 7 are new projects using microservice architecture for a comparatively independent business. The migration of some systems is still incomplete: the systems include both microservices and old monolithic modules. The decisions on the migration highly depend on its business value and effort.

Feedback in response to the second question mainly comes from the participants who hold the positions of manager or architect. 4 of 5 choose to take a product perspective instead of a project perspective on the architectural design and consider the microservice partitioning based on the product business model. They express that this strategy ensures stable boundary and responsibility of different microservices.

Among the 13 surveyed systems, 10 use more than one language, e.g., Java, C++, C#, Ruby, Python, and Node.js. One of the systems (A6) uses more than 5 languages. 9 of the participants state that runtime verification and debugging are the main challenges, and they heavily depend on runtime monitoring and tracing of microservice systems. The managers and architects are interested in using runtime monitoring and tracing for verifying the conformance of their systems to microservice best practices and patterns, while the developers are interested in using them for debugging. Debugging remains as a major challenge for almost all of the participants. They often spend days or even weeks analyzing and debugging of a fault.

3.3 Fault Cases

In total, the 16 participants report 22 fault cases as shown in Table 2. For each case, the table lists its reporter, symptom, root cause, and the time (in days) used to locate the root cause. Detailed descriptions of these fault cases (along with the source code of our open source benchmark system and replicated faults) can be found in our replication package [28]. Note that developers take several days to locate the root causes in most cases. These faults can be grouped into 6 common categories as shown in Table 3 based on their symptoms (functional or non-functional) and root causes (internal, interaction, or environment).

Functional faults result in malfunctioning of system services by raising errors or producing incorrect results. **Non-Functional** faults influence the quality of services such as performance and reliability. From Table 3, it can be seen that most of the faults are functional, causing incorrect results (F1, F2, F8, F9, F10, F11, F12, F13, F14, F18, F19, F21, F22), runtime failures (F7, F15, F16), or no response (F20); only 4 of them are non-functional, causing unreliable services (e.g., F3, F5) or long response time (F4, F17).

The root causes of **Internal** faults lie in the internal implementation of individual microservices. For example, F14 is an internal fault caused by a mistake in the calculation of Consumer Price Index (CPI) implemented in a microservice. The root causes of **Interaction** faults lie in the interactions among multiple microservices. These faults are often caused by missing or incorrect coordination of microservice interactions. For example, F1 is caused by the lack of sequence control in the asynchronous invocations of multiple message delivery microservices; F12 is caused by the incorrect behaviors of a microservice resulted from an unexpected state of another microservice. The root causes of **Environment** faults lie in the configuration of runtime infrastructure, which may influence the instances of a single microservice or the instances of a cluster of microservices. For example, F3 and F20 are caused by improper configuration of Docker (cluster level) and JBoss (service level), respectively. These faults may influence the availability, stability, performance, and even functionality of related microservices.

To learn the characteristics of the faults in microservice systems, we discuss with each participant to determine whether the reported fault cases are particular to microservice architecture. The criterion is whether similar fault cases may occur in systems of monolithic architecture. Based on the discussion, we find that internal faults and service-level environment configuration faults are common in both microservice systems and monolithic systems, while interaction faults and cluster-level environment configuration faults are particular to microservice systems.

3.4 Debugging Practice

Based on the survey, we summarize the existing debugging process of microservice systems and identify different maturity levels of the practices and techniques on debugging. We also analyze the effectiveness of the debugging processes of the reported fault cases.

3.4.1 Debugging Process

Our survey shows that all the participants depend on log analysis for fault analysis and debugging. Their debugging processes are usually triggered by failure reports describing

TABLE 2
Microservice Fault Cases Reported by the Participants

Fault	Reporter	Symptom	Root Cause	Time
F1	P1 (A1)	Messages are displayed in wrong order	Asynchronous message delivery lacks sequence control	7D
F2	P2 (A2)	Some information displayed in a report is wrong	Different data requests for the same report are returned in an unexpected order	3D
F3	P2 (A2)	The system periodically returns server 500 error	JVM configurations are inconsistent with Docker configurations	10D
F4	P3 (A3)	The response time for some requests is very long	SSL offloading happens in a fine granularity (happening in almost each Docker instance)	7D
F5	P4 (A4)	A service sometimes returns timeout exceptions for user requests	The high load of a type of requests causes the timeout failure of another type of requests	6D
F6	P5 (A5)	A service is slowing down and returns error finally	Endless recursive requests of a microservice are caused by SQL errors of another dependent microservice	3D
F7	P6 (A6)	The payment service of the system fails	The overload of requests to a third-party service leads to denial of service	2D
F8	P7 (A7)	A default selection on the web page is changed unexpectedly	The key in the request of one microservice is not passed to its dependent microservice	5D
F9	P7 (A7)	There is a Right To Left (RTL) display error for UI words	There is a CSS display style error in bi-directional	0.5D
F10	P8 (A8)	The number of parts of a specific type in a bill of material (BOM) is wrong	An API used in a special case of BOM updating returns unexpected output	4D
F11	P9 (A8)	The bill of material (BOM) tree of a product is erroneous after updates	The BOM data is updated in an unexpected sequence	4D
F12	P10 (A9)	The price status shown in the optimized result table is wrong	Price status querying does not consider an unexpected output of a microservice in its call chain	6D
F13	P11 (A9)	The result of price optimization is wrong	Price optimization steps are executed in an unexpected order	6D
F14	P11 (A9)	The result of the Consumer Price Index (CPI) is wrong	There is a mistake in including the locked product in CPI calculation	2D
F15	P11 (A9)	The data-synchronization job quits unexpectedly	The spark actor is used for the configuration of actorSystem (part of Apache Spark) instead of the system actor	3D
F16	P11 (A9)	The file-uploading process fails	The "max-content-length" configuration of spray is only 2 Mb, not allowing to upload a bigger file	2D
F17	P12 (A10)	The grid-loading process takes too much time	Too many nested "select" and "from" clauses are in the constructed SQL statement	1D
F18	P13 (A11)	Loading the product-analysis chart is erroneous	One key of the returned JSON data for the UI chart includes the null value	0.5D
F19	P13 (A11)	The price is displayed in an unexpected format	The product price is not formatted correctly in the French format	1D
F20	P14 (A12)	Nothing is returned upon workflow data request	The JBoss startup classpath parameter does not include the right DB2 jar package	3D
F21	P15 (A13)	JAWS (a screen reader) misses reading some elements	The "aria-labeled-by" element for accessibility cannot be located by the JAWS	0.5D
F22	P16 (A13)	The error of SQL column missing is returned upon some data request	The constructed SQL statement includes a wrong column name in the "select" part according to its "from" part	1.5D

TABLE 3
Fault Categories

Root Cause \ Influence	Functional	Non-Functional
Internal	F9, F14, F18, F19, F21, F22	F17
Interaction	F1, F2, F6, F7, F8, F10, F11, F12, F13	F5
Environment	F15, F16, F20	F3, F4

TABLE 4
Maturity Levels of Debugging

Maturity Level	Systems	Percentage
Basic Log Analysis	A1, A7, A11	23%
Visual Log Analysis	A2, A3, A8, A9, A10, A12	46%
Visual Trace Analysis	A4, A5, A6, A13	31%

the symptoms and possibly reproduction steps of the failures, and ended when the faults are fixed. The debugging processes typically include the following 7 steps.

- *Initial Understanding (IU)*. The developers get an initial understanding of the reported failure based on the failure report. They may also examine the logs from the production or test environment to understand the failure. Based on the understanding, they may have a preliminary judgement of the root causes or decide to further reproduce the failure for debugging.
- *Environment Setup (ES)*. The developers set up a runtime environment to reproduce the failure based on their initial understanding of the failure. The environment setup includes the preparation of virtual machines, a deployment of related microservices, and configurations of related microservice instances. To ease the debugging processes, the developers usually set up a simplified environment, which for example includes as less virtual machines and microservices as possible. In some cases the developers can directly use the production or test environment that produces the failure for debugging and thus this step can be skipped.
- *Failure Reproduction (FR)*. Based on the prepared runtime environment, the developers execute the failure scenario to reproduce the failure. The developers usually try different data sets to reproduce the failure to get a preliminary feeling of the failure patterns, which are important for the subsequent steps.
- *Failure Identification (FI)*. The developers identify failure symptoms from the failure reproduction executions. The symptoms can be error messages of microservice instances found in logs or abnormal behaviours of the microservice system (e.g., no response for a long time) observed by the developers.
- *Fault Scoping (FS)*. The developers identify suspicious locations of the microservice system where the root causes may reside, for example implementations of individual microservices, environment configurations, or interactions of a group of microservices.
- *Fault Localization (FL)*. The developers localize the root causes of the failure based on the identified suspicious locations. For each suspicious location, the developers confirm whether it involves real faults that cause the failure and identify the precise location of the faults.
- *Fault Fixing (FF)*. The developers fix the identified faults and verify the fixing by rerunning related test cases.

Note that these steps are not always sequentially executed. Some steps may be repeated if the subsequent steps can not be successfully done. For example, the developers may go back to set up the environment again if they find they can not reproduce the failure. Some steps may be skipped if they are not required. For example, some experienced developers may skip environment setup and failure reproduction if they can locate the faults based on the logs from the production or test environment and verify the fault fixing by their special partial execution strategies.

3.4.2 Maturity Levels of Debugging Practices

We find that the practices and techniques on debugging for the 13 systems can be categorized into 3 maturity levels as shown in Table 4.

The first level is basic log analysis. At this level, the developers analyze original execution logs produced by the system to locate faults. The logs record the execution information of the system at specific points, including the time, executed methods, values of parameters and variables, intermediate results, and extra context information such as execution threads. Basic log analysis follows the debugging practices of monolithic systems and requires only common logging tools such as Log4j [48] for capturing and collecting execution logs. To locate a fault, the developers manually examine a large number of logs. Successful debugging at this level depends heavily on the developers' experience on the system (e.g., overall architecture and error-prone microservices) and similar fault cases, as well as the technology stack being used.

TABLE 5
Time Analysis of Debugging Practices from Industrial Survey

Fault	Type	#MS	Supported Level	Actually Adopted	Overall Time (H)	Time of Each Step (H)						
						IU	ES	FR	FI	FS	FL	FF
F1	Interaction, Functional	3	basic log	basic log	56	16	6	6	8	8	16	6
F2	Interaction, Functional	6	visual log	visual log	26	4	6	6	3	3	6	4
F3	Environment, Non-Functional	6	visual log	basic log	80	26	4	3	16	20	24	8
F4	Environment, Non-Functional	10+	visual log	visual log	56	16	10	6	6	8	8	4
F5	Interaction, Non-Functional	6	visual trace	basic log	48	16	6	4	4	8	8	3
F6	Interaction, Functional	8	visual trace	visual trace	24	3	3	3	4	4	6	4
F7	Interaction, Functional	6	visual trace	visual trace	16	3	3	3	2	1	2	4
F8	Interaction, Functional	3	basic log	basic log	40	8	6	4	4	8	8	4
F9	Internal, Functional	1	basic log	basic log	4	1	-	-	0.5	1	1	0.5
F10	Interaction, Functional	4	visual log	basic log	32	6	4	4	4	8	4	4
F11	Interaction, Functional	6	visual log	visual log	32	4	6	6	4	4	6	3
F12	Interaction, Functional	4	visual log	basic log	48	8	4	4	4	8	16	4
F13	Interaction, Functional	6	visual log	visual log	48	4	8	4	8	12	12	4
F14	Internal, Functional	1	visual log	visual log	16	4	2	2	1	2	3	3
F15	Environment, Functional	2	visual log	basic log	24	8	2	2	2	3	4	3
F16	Environment, Functional	2	visual log	visual log	16	4	2	2	1	3	3	3
F17	Internal, Non-Functional	1	visual log	visual log	8	4	-	-	-	-	2	2
F18	Internal, Functional	1	basic log	basic log	4	2	-	-	-	0.5	1	1
F19	Internal, Functional	1	basic log	basic log	8	4	-	-	-	-	1	4
F20	Environment, Functional	3	visual log	visual log	24	4	2	3	4	4	4	4
F21	Internal, Functional	1	visual trace	basic log	5	1	1	1	-	-	1	1
F22	Internal, Functional	1	visual trace	basic log	12	4	2	2	-	-	3	3
		3.7			28.5	7	3.5	3	3.1	4.4	6.3	3.5

The second level is visual log analysis. At this level, execution logs are structured and visualized for fault localization. The developers can flexibly retrieve specific execution logs that they are interested in using conditions and regular expressions, and sort the candidate results according to specific strategies of debugging. The selected execution logs can be aggregated and visualized by different kinds of statistical charts. Log retrieval and visualization are usually combined to allow the developers to interactively drill up and down through the data (execution logs). For example, to locate a fault resulting in abnormal execution results for a microservice, the developers can first use a histogram to learn the range and distribution of different results and then choose a specific abnormal result to examine related execution logs. To support visual log analysis, the developers need to use a centralized logging system to collect the execution logs produced in different nodes and include information about the microservice and its instances in execution logs. Log analysis at this level highly depends on tools for log collection, retrieval, and visualization. A commonly used toolset is the ELK stack, i.e., Logstash [49] for log collection, ElasticSearch [50] for log indexing and retrieval, and Kibana [51] for visualization.

The third level is visual trace analysis. At this level, the developers further analyze collected traces of system executions with the support of trace visualization tools. A *trace* is resulted from the execution of a scenario (e.g., a test case), and is composed of user-request trace segments. A user-request trace segment consists of logs that share the same user request ID (created for each user request). In particular, when a user request comes in the front door of the system, the adopted tracing framework [52] creates a unique user request ID, which is passed along with the request to each directly or indirectly invoked microservice. Thus, logs collected for each such invoked microservice record the user request ID. The developers can use visualization tools to analyze user requests' microservice-invocation chains (extracted from the traces) and identify suspicious ranges of microservice invocations and executions. As a microservice can invoke multiple microservices in parallel, the visualization tools usually organize the microservice-invocation chains into a tree structure. For example, a visualization tool can vertically show a nested structure of microservice invocations and horizontally show the duration time of each microservice invocation with colored bars. Analysis at this level highly depends on advanced microservice execution tracing and visualization tools. Commonly used toolset include Dynatrace [52] and Zipkin [53]. Our survey shows that most companies choose to implement their own tracing and visualization tools, as they are specific to the implementation techniques of microservice architecture.

Visual log analysis provides better support for most types of faults than basic log analysis. Flexible log retrieval provides a quick filtering of execution logs. Visualized statistics of microservice executions (e.g., variable values or method execution counts) reveal patterns of microservice executions. These patterns can help locate suspicious microservice executions. For example, for F22, the developers can easily exclude those methods that are executed fewer times than that of failure occurrences based on the statistics. However, locating interaction-related faults often requires the developers to understand microservice executions in the context of microservice-invocation chains. Visual trace analysis further improves visual log analysis by embedding log analysis in the context of traces. For example, for F1, the developers can compare the traces of success scenarios with the traces of failure scenarios, and identify the root cause based on the different orders of specific microservice executions.

Tables 2 and 4 show that there is often a mismatch between the log analysis level and the faults. For example, the system A7 is still at the basic log analysis level, which cannot help locate the fault F8 reported from this system. In such cases, the developers often need to manually investigate a lot of execution logs and code. They usually start with the failure-triggering location in the logs and then examine the logs backwards to find suspicious microservices and check the code of the microservices.

3.4.3 Effectiveness Analysis

To analyze the effectiveness of different debugging practices we collect the maturity levels of debugging practices and the time consumed by each step of the 22 fault cases. Table 5 shows the results, including the fault type, the number of microservices involved in the fault case (#MS), the supported maturity level and the actually adopted maturity level of debugging, and the time consumed for the whole debugging process and individual steps. The last line shows the average of the #MS and the average of the time consumed for the entire debugging process and individual steps.

Note that for some fault cases the maturity levels of the debugging practices adopted by the developers are lower than the levels supported in their teams. For example, for F5 the developers choose to use basic log analysis while they are equipped with visual trace analysis. Moreover, the developers may also combine practices of different levels. For example, when they adopt visual trace analysis or visual log analysis they may also use basic log analysis to examine details.

The time consumed for the whole debugging process and individual steps is obtained from the descriptions of the participants during the interviews. The participants are asked to estimate the time by hours. To validate their estimation, the participants are asked to confirm the estimation with their colleagues and examine the records (e.g., the debugging time indicated by the period between bug assignment and resolution) in their issue tracking systems.

On average the time used to locate and fix a fault increases with the number of microservices involved in the fault: 9.5 hours for one microservice, 20 hours for two microservices, 40 hours for three microservices, 48 hours for more than three microservices. For some fault cases (e.g., F6) the overall time is less than the sum of the time spent on each step. This is usually caused by the simultaneous execution of multiple steps. For example, when confirming a suspicious location of the fault in fault localization, the developers can simultaneously conduct fault scoping to identify more suspicious locations.

We find that the advantages of visual log analysis and visual trace analysis are more obvious for interaction faults. On average, the developers spend 20, 35, 45 hours in these fault cases adopting visual trace analysis, visual log analysis, basic log analysis, respectively.

In general, initial understanding, fault scoping, fault localization are more time consuming than the other steps, as these steps require in-depth understanding and analysis of logs. Also in these steps the advantages of visual log/trace analysis are more obvious. For example, the average time for initial understanding is 3, 7, 21 hours when using visual trace analysis, visual log analysis, basic log analysis, respectively. In some cases (e.g., F9, and F17-19) the developers choose to skip environment setup and failure reproduction, as they can easily identify the failure symptoms from user interfaces or exceptions. In other cases (e.g., F17, F19, F21, F22) the developers choose to skip failure identification and fault scoping, as they can identify potential locations of the faults based on failure symptoms and past experience.

According to the feedback of the participants, 11 out of 13 of them who have experiences of visual log/trace analysis believe that the visual analysis tools and practices are very useful. But how much the tools and practices can help depends on the faulty types and developers' experiences, skills, and preferences.

4 Benchmark System and Fault-Case Replication

Our survey clearly reveals that the existing practices for fault analysis and debugging of microservice systems can be much improved. To conduct research in this area, one of the difficulties faced by researchers is that there is a lack of benchmark systems, which is due to the great complexity in setting up realistic microservice systems. We thus set up a benchmark system: TrainTicket [25]. Our empirical study is based on TrainTicket and the 22 fault cases that are reported in the survey and replicated in the system. The system and the replicated fault cases can be used as a valuable benchmark for the broad research community to further conduct practice-relevant research on microservice fault analysis and debugging, and even other broad types of practice relevant research on microservices.

TrainTicket provides typical train ticket booking functionalities such as ticket enquiry, reservation, payment, change, and user notification. It is designed using microservice design principles and covers different interaction modes such as synchronous invocations, asynchronous invocations, and message queues. The system contains 41 microservices related to business logic (without counting all database and infrastructure microservices). It uses four programming languages: Java, Python, Node.js, and Go. Detailed description of the system (along with the source code of our open source benchmark system and replicated faults) can be found in our replication package [28].

We replicate all the 22 fault cases collected from the industrial survey. In general, these fault cases are replicated by transferring the fault mechanisms from the original systems to the benchmark system. In the following, we describe the replication implementation of some representative fault cases. Descriptions of the replication implementation of the other fault cases can be found in our replication package [28].

F1 is a fault associated with asynchronous tasks, i.e., when we send messages asynchronously without message sequence control. We replicate this fault in the order cancellation process of TrainTicket. In the process, there are two asynchronous tasks being sent, which have no additional sequence control. The first task should always be completed before the second one. However, if the first task is delayed and completed only after the second one, the order reaches an abnormal status, leading to a failure.

F3 is a reliability problem caused by improper configurations of JVM and Docker. JVM's max memory configuration conflicts with Docker cluster's memory limitation configuration. As a result, Docker sometimes kills the JVM process. We replicate this fault in the ticket searching process. We select some microservices that are involved in this process and revise them to be more resource consuming. These revised microservices are deployed in a Docker cluster with conflicting configurations, thus making these microservices sometimes unavailable.

F4 is a performance problem caused by improper configuration of Secure Sockets Layer (SSL) applied for many microservices. The result is frequent SSL offloading at a fine granularity, which slows down the executions of related microservices. We replicate this fault by applying the faulty SSL configuration to every microservice of TrainTicket. Then when a user requests a service (e.g., ticket reservation), he/she will feel that the response time is very long.

F5 is a reliability problem caused by improper usage of a thread pool. The microservice uses a thread pool to process multiple different types of service requests. When the thread pool is exhausted due to the high load of a type of service requests, another type of service requests will fail due to timeout. We replicate this fault in the ticket reservation service, which serves both the ticket searching process and the ticket booking process. When the load of ticket searching is high, the thread pool of the service will be exhausted and the ticket booking requests to the service will fail due to timeout.

F8 is caused by missing or incorrect parameter passing along an invocation chain. We replicate this fault in the order cancellation process. When a VIP user tries to cancel a ticket order, the login token saved in Redis [54] (an in-memory data store) is not passed to some involved microservices that require the token. This fault causes that the user gets unexpectedly lower ticket refund rate.

F10 is caused by an unexpected output of a microservice, which is used in a special case of business processing. We replicate this fault in the ticket booking process. In the ticket ordering service, we implement two APIs, which respectively serve for general ticket ordering and ticket ordering of some special stations. The API for special ticket ordering sometimes returns an unexpected output that is not correctly handled, thus making the ticket booking process fail.

F11 is a fault that occurs in asynchronous updating of data, caused by the missing of sequence control. When the bill of material (BOM) tree is updated in an unexpected order, the resulting tree is incorrect. But when the user turns on the "strict mode" on product BOM services, the resulting tree is rebuilt when the BOM tree includes some negative numbers, leading to a correct tree. We replicate this fault in the order cancellation process, which includes two microservices (payment service and cancel service) that asynchronously set the same value in the database. Due to the missing of sequence control, the two microservices may set the value in a wrong sequence, thus causing an incorrect value. But if the user turns on the "strict order" mode on the order service, the incorrect value will be corrected eventually.

F12 is caused by an unexpected output of a microservice when it is in a special state. We replicate this fault in the ticket booking process. We introduce state *admDepStation/admDesStation* for the ticket reservation service instance to indicate the departure/destination station of which the administrator is examining the tickets. if no administrator examining things happened, the corresponding ticket reservation service instance will be without state. If the departure/destination station of a ticket reservation request is *admDepStation/admDesStation*, and the ticket reservation service is accessed by the same request thread twice or more times including both with and without state instance, the request will be denied with an unexpected output and the ticket booking process returns an error.

For each of these preceding faults, we create a development branch for its replication in the fault case repository [27]. Researchers using the repository can easily mix and match different faults to produce a faulty version of TrainTicket including multiple faults.

5 Empirical Study

Our empirical study with the TrainTicket system and the replicated fault cases includes two parts. In the first part, we investigate the effectiveness of existing industrial debugging practices for the fault cases. In the second part, we develop a microservice execution tracing tool and two trace visualization strategies for fault localization based on a state-of-the-art debugging visualization tool [10] for distributed systems, and investigate whether it can improve the effectiveness of debugging interaction faults. A group of 6 graduate students who are familiar with TrainTicket and have comparable experiences of microservice development serve as the developers for conducting debugging independently. For each fault case, the developers locate and fix the faults based on a given failure report, and the developers who debug with different practices are different. To provide a fair comparison, we randomly select a developer for each fault case and each practice to allow a developer to use different practices for different fault cases. The developers follow the general process presented in Section 3.4.1 for debugging. For any step, if the developers cannot complete in two hours they can choose to give up and the step and the whole process fail.

5.1 Debugging with Industrial Debugging Practices

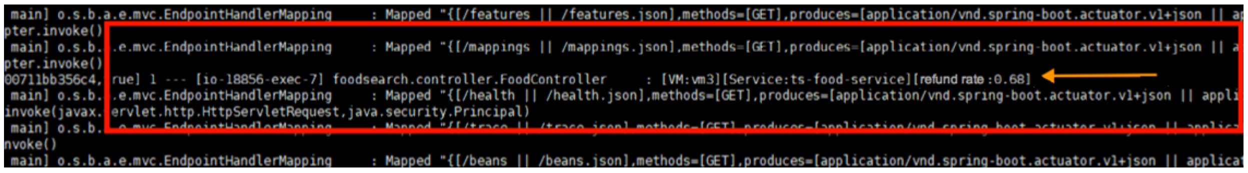
In this part of the study, we investigate the effectiveness of the debugging practices of the three maturity levels by qualitative analysis and quantitative analysis respectively. For each fault case three developers are selected to debug with the practices of different maturity levels. The tools provided for different maturity levels are as follows.

- **Basic Log Analysis**. The developers use command line tools to grab and analyze logs.
- **Visual Log Analysis**. The developers use the ELK stack, i.e., Logstash [49] for log collection, ElasticSearch [50] for log indexing and retrieval, and Kibana [51] for visualization.
- **Visual Trace Analysis**. The developers use both the ELK stack and Zipkin [53] for debugging.

5.1.1 Qualitative Analysis

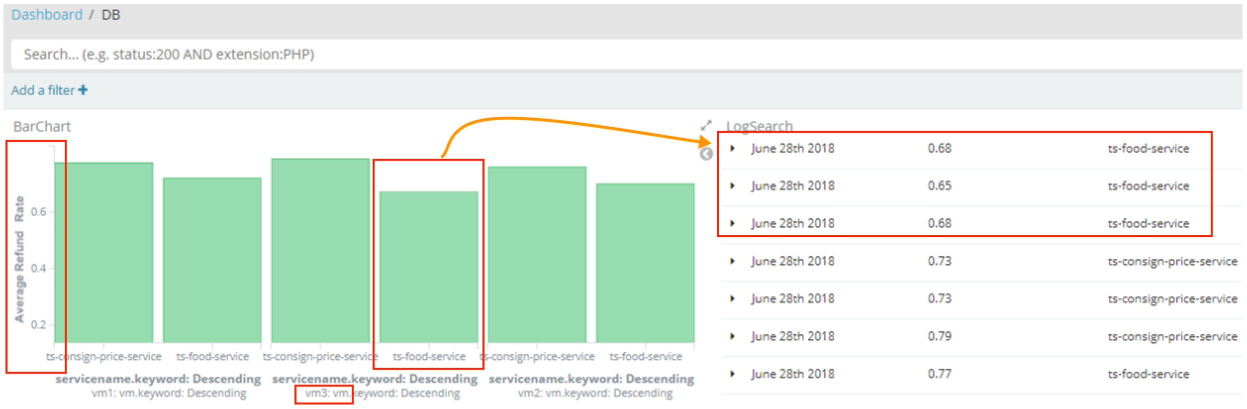
We qualitatively compare different levels of practices based on the debugging processes of F8 as shown in Figure 1.

Figure 1(a) presents a snapshot of basic log analysis, which shows the logs captured from a container running a microservice of food service. The developers identify a suspicious log fragment in the red box and find that the food refund rate is 68%, which is lower than the predefined VIP refund rate for food ordering. Thus they can regard the calculation of food refund rate as a potential fault location. A shortage of basic log analysis is the lack of context of



```
main] o.s.b.a.e.mvc.EndpointHandlerMapping      : Mapped "{[/features || /features.json],methods=[GET],produces=[application/vnd.spring-boot.actuator.v1+json || ap
pter.invoke()
main] o.s.b. .e.mvc.EndpointHandlerMapping      : Mapped "{[/mappings || /mappings.json],methods=[GET],produces=[application/vnd.spring-boot.actuator.v1+json || a
pter.invoke()
00711bb356c4, rue] 1 --- [io-18856-exec-7] foodsearch.controller.FoodController     : [VM:vm3][Service:ts-food-service][refund rate :0.68]
main] o.s.b. .e.mvc.EndpointHandlerMapping      : Mapped "{[/health || /health.json],methods=[GET],produces=[application/vnd.spring-boot.actuator.v1+json || appli
invoke(javax. ervlet.http.HttpServletRequest,java.security.Principal)
main] o.s.b.
nvoke()
main] o.s.b.a.e.mvc.EndpointHandlerMapping      : Mapped "{[/beans || /beans.json],methods=[GET],produces=[application/vnd.spring-boot.actuator.v1+json || applicat
```

(a) Basic Log Analysis



Dashboard / DB

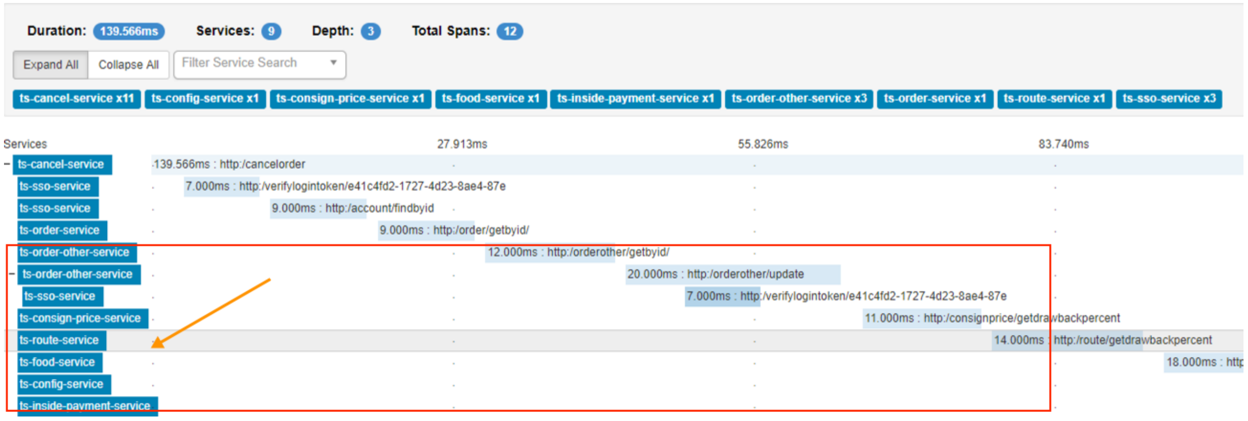
Search... (e.g. status:200 AND extension:PHP)

Add a filter +

BarChart
Average Refund Rate
0.6
0.4
0.2
ts-consign-price-service
ts-food-service
ts-consign-price-service
ts-food-service
ts-consign-price-service
ts-food-service
servicename.keyword: Descending
vm1: vm.keyword: Descending
servicename.keyword: Descending
vm3: vm.keyword: Descending
servicename.keyword: Descending
vm2: vm.keyword: Descending
LogSearch

June 28th 2018	0.68	ts-food-service
June 28th 2018	0.65	ts-food-service
June 28th 2018	0.68	ts-food-service
June 28th 2018	0.73	ts-consign-price-service
June 28th 2018	0.73	ts-consign-price-service
June 28th 2018	0.79	ts-consign-price-service
June 28th 2018	0.77	ts-food-service

(b) Visual Log Analysis



```
Services                          27.913ms                    55.826ms                    83.740ms
- ts-cancel-service          139.566ms : http:/cancelorder
  ts-sso-service             7.000ms : http:/verifylogintoken/e41c4fd2-1727-4d23-8ae4-87e
  ts-sso-service             9.000ms : http:/account/findbyid
  ts-order-service           9.000ms : http:/order/getbyid/
  ts-order-other-service     12.000ms : http:/orderother/getbyid/
- ts-order-other-service     20.000ms : http:/orderother/update
   ts-sso-service            7.000ms : http:/verifylogintoken/e41c4fd2-1727-4d23-8ae4-87e
  ts-consign-price-service   11.000ms : http:/consignprice/getdrawbackpercent
  ts-route-service           14.000ms : http:/route/getdrawbackpercent
  ts-food-service            18.000ms : http
  ts-config-service
  ts-inside-payment-service
```

(c) Visual Trace Analysis

Fig. 1. Qualitative Comparison of Different Levels of Debugging Practices

microservice invocation, which makes it hard for the developers to analyze and understand the logs in the context of user requests and invocation chains.

Figure 1(b) presents a snapshot of visual log analysis. It shows the histograms of average refund rate of the instances of two related microservices (food service and consign service) in different virtual machines, as well as corresponding logs. The refunds of these two services are both included in the ticket refund. As the failure symptom is low ticket refund rate, the developers choose the lowest bar which shows the average food refund rate in VM3 (see the red box in Figure 1(b)) to check the logs. From the logs the developers find that the lowest food refund rate is 65%, and thus regard the calculation of food refund rate as a potential fault location. Compared with basic log analysis, visual log analysis provides aggregated statistics of variables and quality attributes (e.g., response time), and thus can help developers to identify suspicious microservices and instances. However, it lacks the context of user requests and invocation chains, and thus can not support the analysis of microservice interactions.

Figure 1(c) presents a snapshot of visual trace analysis. It shows the entire trace of the order cancellation process, including the nested invocations of microservices, and the consumed time of each invocation. The developers find that the ticket cancellation process invokes not only the food service and the consign service, but also the config service and route service. Then they further analyze the logs of the config service and find that a suspicious general refund rate (which can be 36% in the lowest case) is used by the ticket cancellation process to calculate the final refund rate. They thus regard the calculation of general refund rate in the config service as a potential fault location. This fault localization is more precise than the localization supported by the basic log analysis and the visual log analysis.

Compared with visual log analysis, visual trace analysis supports the understanding of microservice executions in the context of user requests and invocation chains.

5.1.2 Quantitative Analysis

The results of the study are shown in Table 6, including the time used for the whole debugging process and that of each

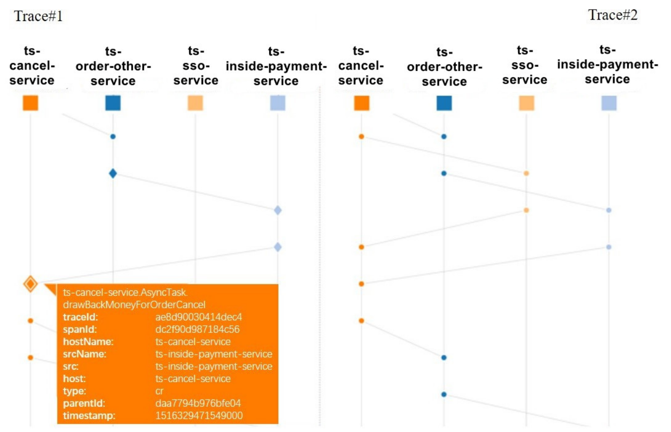
TABLE 6
Time Analysis of Debugging with Industrial Debugging Practices

Fault	Maturity Level	Overall Time (H)	Time of Each Step (H)						
			IU	ES	FR	FI	FS	FL	FF
F1	basic log	6.3	2	0.5	0.5	1	1	1.5	0.5
	visual log	4	1.3	0.3	0.3	0.7	0.3	0.7	0.2
	visual trace	3.3	1	0.3	0.3	0.6	0.4	0.6	0.2
F2	basic log	6.8	1.8	1.2	0.9	1	0.8	1.3	0.6
	visual log	4	1	0.6	0.4	0.7	0.6	0.8	0.3
	visual trace	3.1	0.6	0.6	0.4	0.6	0.6	0.6	0.3
F3	basic log	failed	3	1	1	failed	failed	failed	failed
	visual log	failed	2.6	1.2	0.6	1.2	failed	failed	failed
	visual trace	failed	2.2	1	0.6	1	failed	failed	failed
F4	basic log	failed	2	1	0.3	1	failed	failed	failed
	visual log	failed	1.1	0.9	0.4	1	failed	failed	failed
	visual trace	failed	1	0.8	0.3	1	failed	failed	failed
F5	basic log	5	2	0.6	0.6	0.7	0.7	0.6	0.2
	visual log	4.4	1.3	0.4	0.7	0.6	0.4	0.7	0.3
	visual trace	3.2	0.9	0.4	0.6	0.4	0.4	0.6	0.2
F6	basic log	2.8	0.6	0.3	0.3	0.4	0.6	0.4	0.2
	visual log	2.1	0.4	0.3	0.3	0.3	0.2	0.3	0.3
	visual trace	1	0.2	0.1	0.1	0.1	0.1	0.3	0.3
F7	basic log	5.3	1.6	1.2	0.9	0.6	0.6	0.7	0.4
	visual log	3.9	0.8	0.8	0.8	0.3	0.4	0.6	0.4
	visual trace	2.8	0.4	0.4	0.4	0.3	0.3	0.6	0.4
F8	basic log	5.4	1.6	0.6	0.6	1	0.8	0.6	0.3
	visual log	3.6	0.6	0.6	0.8	0.9	0.9	0.4	0.2
	visual trace	3.2	0.6	0.6	0.8	0.6	0.6	0.4	0.2
F9	basic log	1.8	0.3	0.2	0.2	0.3	0.3	0.4	0.1
	visual log	-	-	-	-	-	-	-	-
	visual trace	-	-	-	-	-	-	-	-
F10	basic log	4	1.2	0.3	0.3	0.8	0.6	0.6	0.3
	visual log	3.6	1	0.3	0.3	0.6	0.6	0.6	0.3
	visual trace	3	0.8	0.3	0.3	0.4	0.4	0.6	0.2
F11	basic log	6	1.6	0.8	0.8	1	0.6	1	0.3
	visual log	5	1.1	0.6	0.4	1	0.6	0.8	0.2
	visual trace	2.9	0.4	0.4	0.4	0.6	0.4	0.6	0.2
F12	basic log	10	4	0.8	0.8	1	1.4	1.2	0.8
	visual log	6	2	0.6	0.6	0.6	1	0.8	0.4
	visual trace	3.3	1	0.6	0.4	0.4	0.4	0.6	0.2
F13	basic log	6.3	2	0.7	0.7	0.8	1	1	0.3
	visual log	5.4	1.6	0.6	0.6	0.6	0.6	0.8	0.4
	visual trace	4.3	1	0.6	0.3	0.4	0.6	0.9	0.3
F14	basic log	1	0.2	0.1	0.1	0.2	0.2	0.1	0.1
	visual log	1.1	0.3	0.1	0.1	0.2	0.2	0.1	0.1
	visual trace	0.8	0.2	0.1	0.1	0.1	0.1	0.1	0.1
F15	basic log	1	0.2	0.2	0.2	0.1	0.1	0.1	0.1
	visual log	0.4	0.1	-	-	-	-	0.2	0.1
	visual trace	0.4	0.1	-	-	-	-	0.2	0.1
F16	basic log	2.1	0.6	0.3	0.3	0.3	0.3	0.2	0.2
	visual log	1.8	0.4	0.3	0.3	0.3	0.3	0.2	0.2
	visual trace	2	0.4	0.3	0.3	0.4	0.4	0.2	0.2
F17	basic log	2.6	1	0.3	0.3	0.2	0.2	0.4	0.2
	visual log	1.6	0.4	0.3	0.3	0.2	0.2	0.1	0.1
	visual trace	1.7	0.4	0.3	0.3	0.1	0.3	0.2	0.1
F18	basic log	1.3	0.4	0.2	0.2	0.2	0.2	0.1	0.1
	visual log	1	0.4	-	-	0.2	0.2	0.2	0.1
	visual trace	1.1	0.4	-	-	0.2	0.2	0.2	0.1
F19	basic log	0.7	0.3	-	-	0.1	0.1	0.1	0.1
	visual log	-	-	-	-	-	-	-	-
	visual trace	-	-	-	-	-	-	-	-
F20	basic log	2.2	0.8	0.3	0.3	0.4	0.2	0.2	0.1
	visual log	0.8	0.3	-	-	0.2	0.1	0.1	0.1
	visual trace	0.8	0.3	-	-	0.2	0.1	0.1	0.1
F21	basic log	1.6	0.4	0.2	0.2	0.3	0.2	0.2	0.1
	visual log	-	-	-	-	-	-	-	-
	visual trace	-	-	-	-	-	-	-	-
F22	basic log	0.4	0.2	-	-	-	-	0.1	0.1
	visual log	-	-	-	-	-	-	-	-
	visual trace	-	-	-	-	-	-	-	-

step. A mark "-" means that the developer skips the step. If all the steps are skipped, it means that the fault can be easily identified and fixed with lower level practices (e.g., basic log analysis) thus there is no need for higher level practices. A mark "failed" means that the developer fails to complete the step. If a step of a debugging process fails, the whole process fails also.

The developers fail in F3 and F4 with all the three levels of industrial practices. Both of them are non-functional **Environment** faults. For F9, F19, F21, F22, the developers easily locate and fix the faults with basic log analysis. All of them are **Internal** faults. For the other faults, there is a general trend of reduced debugging time with the employment of higher level debugging practices (from basic log, visual log, to visual trace analysis). In these fault cases **Interaction** faults are the ones that benefit the most from higher levels of debugging practices.

Similar to the industrial survey, initial understanding, fault scoping, fault localization are more time consuming than the other steps, and environment setup and failure reproduction are sometimes skipped. The time used for



ts-cancel-service.AsyncTask.
drawBackMoneyForOrderCancel

traceId:	ae8d90030414dec4
spanId:	dc2f90d987184c56
hostName:	ts-cancel-service
srcName:	ts-inside-payment-service
src:	ts-inside-payment-service
host:	ts-cancel-service
type:	cr
parentId:	daa7794b976bfe04
timestamp:	1516329471549000

Fig. 2. Trace Visualization by ShiViz

environment setup and failure reproduction varies with the employed debugging practices. According to the feedback from the developers, they often try to make a simplest failure reproduction based on the initial understanding, so the accuracy of the initial understanding influences the time used for environment setup and failure reproduction

5.2 Debugging with Improved Trace Visualization

From the above, we observe that tracing and visualizing can potentially help fault analysis and debugging of microservice systems. Thus, in this part of the study, in order to better support fault analysis and debugging of microservice systems, we investigate the effectiveness of the state-of-the-art distributed system debugging techniques for microservice system debugging.

5.2.1 Tracing and Visualization Approach

ShiViz [10] is a state-of-the-art debugging visualization tool for distributed systems. It visualizes distributed system executions as interactive time-space diagrams that explicitly capture distributed ordering of events in the system. ShiViz supports pairwise comparison of two traces by highlighting their differences. It compares the distribute-system nodes and events from two traces by names and descriptions, and highlights the nodes or events (in one trace) that do not appear in the other. ShiViz supports the selection of a part of a trace for comparison. For example, we can select a user-request trace segment, i.e., the events for a specific user request based on the request ID.

Figure 2 presents an example of trace visualization by ShiViz, which shows the nodes (colored boxes at the top), the node timelines (vertical lines), events (circles on timelines), and partial orders between events (edges connecting events). The rhombuses (events) on the left side highlight the differences between the two traces, and we can also click to see the detail of the rhombuses.

This pairwise comparison can be used to locate suspicious nodes and events in microservice system debugging when execution information (e.g., service name, user request ID, and invoked method) is added to the names and descriptions of nodes and events, by treating a microservice unit as a distribute-system node. We can leverage ShiViz to visualize the traces of microservices by transforming the trace logs into the log formats of ShiViz. However, a basic problem for visualizing microservice traces is how to map microservice units to nodes. Microservice instances run on containers and can be dynamically created or destroyed together with their containers. Moreover, the instance that is assigned to handle a microservice invocation is uncertain. Therefore, microservice instances (containers) cannot be treated as the nodes in trace visualization. We thus propose the following two visualization strategies for microservice traces.

- *Microservice as Node (Service-Level Analysis)*. All the instances of the same microservice are treated as one node. Thus the events at different instances of the same microservice are aggregated to the same node.
- *Microservice State as Node (State-Level Analysis)*. All the instances of the same microservice and with the same state (determined based on predefined state variables or expressions) are treated as one node. Thus the events at different instances of the same microservice and during the same state are aggregated to the same node. This technique depends on predefined state variables or expressions of each microservice.

A trace resulted from executing a microservice system's scenario (e.g., test case) may include many user requests (e.g., a ticket query request triggered by a button click on a web page). In the trace, a user-request trace segment resulted from each user request includes a series of microservice invocations. Furthermore, each microservice invocation may involve multiple execution events occurring on different microservice instances such as sending/receiving an invocation request or a call back. Therefore, an execution tracing tool needs to log all the execution events and attach a user request ID (*request ID* in short) and a microservice invocation ID (*invocation ID* in short) to each log. Our tool is implemented in Java. For REST invocations, we use the servlet filter and interceptor to inject tracing information of the caller into the http header of the *HTTPRequest* object. The injected information includes the request ID, invocation ID, microservice name, instance ID, IP address, port number, along with the class name and method name of the invoked microservice and the caller, respectively. Such information is sent together with the http request to the callee. For message queues, we use a message channel interceptor to inject and catch the interaction queue data. Based on such tracing information, each microservice instance records tracing logs, which are then collected by our central logging system.

Based on the two visualization strategies, we leverage ShiViz to diagnose microservice faults by pairwise comparison of traces. This characteristic makes ShiViz superior to previous microservice tracing/visualization tools, e.g., ELK stack, Zipkin.

5.2.2 Debugging Methodology

Based on the tracing tool and the two visualization strategies, we define a debugging methodology as shown in Figure 3 for our empirical study based on general debugging practices of distributed systems. The rationale of the process is based on the assumption that the fault-revealing parts of a failure trace are different from the corresponding parts in a success trace, and are shared with another failure trace.

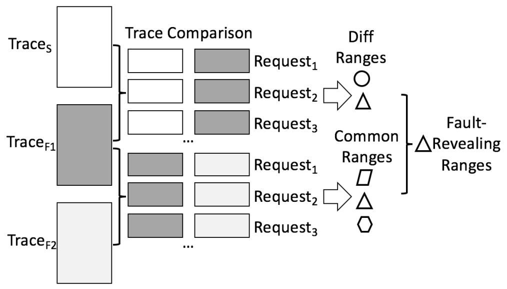


Fig. 3. Debugging Methodology

For each fault, we collect a set of traces resulted from executing the same scenario (with different parameter values) including both success traces and failure traces. We select a success trace Trace_S and two failure traces Trace_{F1} and Trace_{F2} for comparison. We first compare Trace_S and Trace_{F1} based on user requests. For example, a scenario of ticket booking includes a series of user requests such as ticket query, train selection, and passenger selection, and the events for each of these requests in two traces are compared separately. To ease the selection of user requests, we attach a readable label (e.g., "ticket query") for each request ID in the logs. Based on the comparison, we can obtain a set of diff ranges DR, each of which are multiple consecutive events that are different between the two traces. We then compare between Trace_{F1} and Trace_{F2} by user requests to confirm the ranges in DR that are shared in the two traces. The confirmed ranges in DR are identified as fault-revealing ranges.

The purpose of this study is to investigate whether trace-based debugging can benefit from improved tracing and visualization. Therefore, we select 12 fault cases from all the 22 ones for the study according to the following two criteria. First, they are **Interaction** or **Environment** faults, as **Internal** faults can not benefit from trace analysis. Second, their debugging time is more than 1 hour by visual trace analysis (see Table 6). The selected fault cases are: F1, F2, F3, F4, F5, F7, F8, F10, F11, F12, F13, F16. For each fault case a developer who did not debug the fault using the industrial debugging practices is selected to locate and fix the fault.

5.2.3 Qualitative Analysis

The benefits that different debugging steps can obtain from the improved trace visualization are different. For initial understanding the developers can get an impression of the overall differences between success traces and failure traces, including the number and size of diff ranges. Some developers can even directly identify suspicious fault locations in initial understanding. Environment setup, failure reproduction, and failure identification can indirectly benefit from the analysis through more accurate initial understanding of the failure. Fault scoping and fault localization can directly benefit from the analysis by identifying diff ranges. Fault fixing can indirectly benefit from the analysis in the verification of fault fixing.

Figure 4 shows an example of service-level analysis for faulty microservice invocation in F10, which is caused by incorrect handling of an unexpected output of one of the APIs of a microservice. The developers can see there are two differences (rhombus) in the red box, indicating that the invoked APIs are different (although the invoked

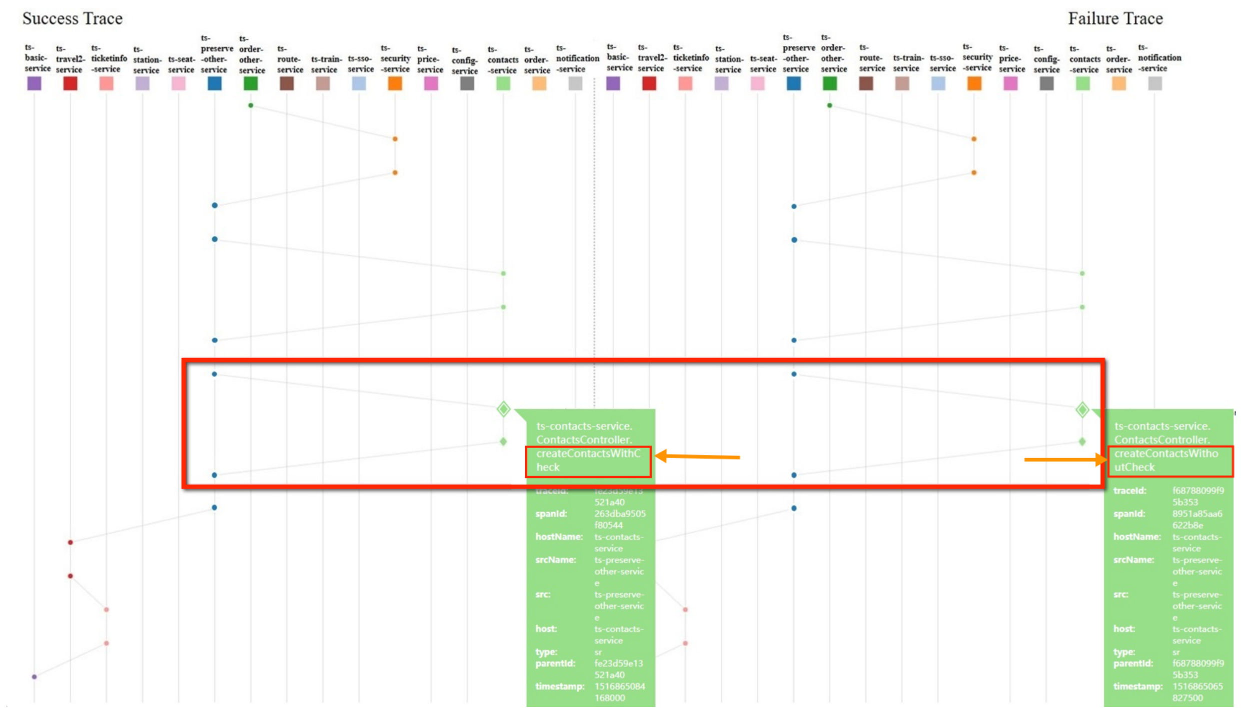


Fig. 4. An Example of Service Level Analysis for Faulty Microservice Invocation

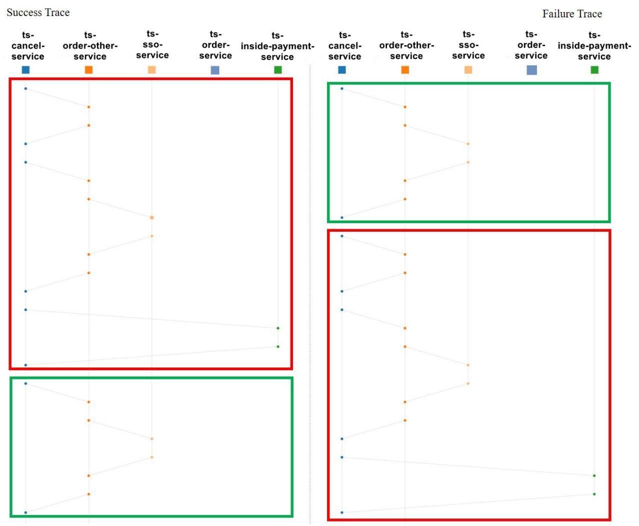
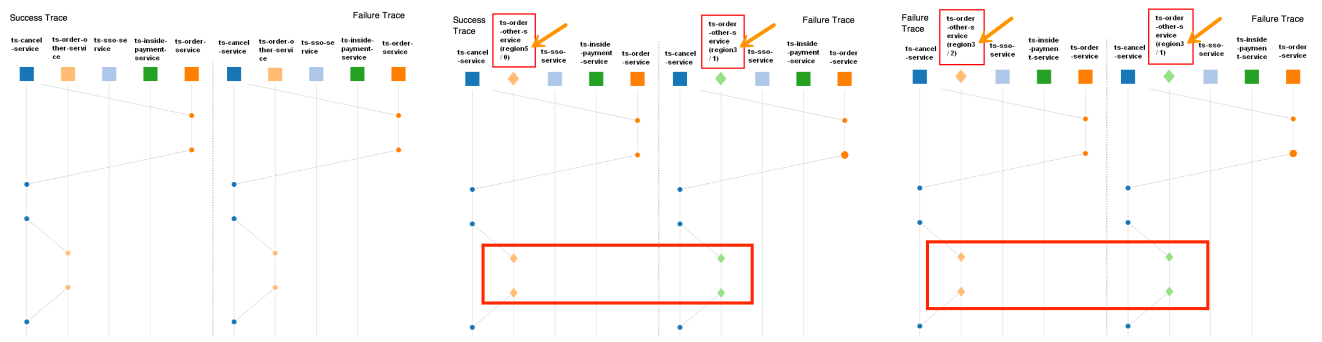


Fig. 5. An Example of Service Level Analysis for Faulty Interaction Sequence

microservices are the same). As the corresponding range in the comparison of failure traces involves no differences, they can identify this range as a potential fault location.

Figure 5 shows an example of service-level analysis for faulty interaction sequence in F1, which is caused by missing coordination of asynchronous invocations. Based on the structures of the events, the developers can identify two pairs of corresponding ranges in the success trace and the failure trace, showing as red boxes and green boxes respectively. They can find that the orders of the two ranges in the two traces are different. As the orders have no difference in the comparison of failure traces, they can identify the difference as a potential fault location.

Figure 6 shows an example of service- and state-level debugging for F12. When conducting service-level analysis as shown in Figure 6(a), there is no difference between the failure trace and the success trace. Then the developers choose to use state-level analysis to refine the comparison of the traces by introducing state variables or expressions. Based on the understanding of the system, the developers choose to try the following two state variables of the ticket order service: *administrator examining station* indicating the departure/destination station of which the administrator is examining the tickets, and *order processing thread pool* indicating whether the limit of the pool is not reached, reached, or exceeded (0, 1, or 2). As the *administrator examining station* has more than 20 different values, the developers choose to use state expression instead of the state variable, e.g., use the country region information: *GetRegion (administrator examining station)*. Then the developers get the state-level analysis results as shown in Figure 6(b). It can be seen that the nodes for the ticket order service are annotated with the combined states (“region5”/0 and “region3”/1) and thus the differences (nodes and events) between the failure trace and the success trace are identified and highlighted as rhombuses (in the red box). The developers further examine the comparison between the two failure traces as shown in Figure 6(c), which compares the ticket order service with the same administration state (“region3”) and different thread pool states (2 and 1). These two comparisons provide preliminary evidence that the state of administrator examining “region3” of the ticket order service is relevant to the fault and the range highlighted in Figure 6(b) is a candidate fault-revealing range.


Success Trace
Failure Trace
ts-cancel-service
ts-order-other-service
ts-sso-service
ts-inside-payment-service
ts-order-service
ts-order-other-service (region5 / 0)
ts-order-other-service (region3 / 1)
ts-order-other-service (region3 / 2)

(a) Service Level Analysis (b) State Level Analysis (Success Vs. Failure) (c) State Level Analysis (Failure Vs. Failure)

Fig. 6. An Example of Service- and State-Level Analysis

When necessary, the developers can introduce more state variables/expressions for comparison, paying the price of having more nodes in the visualized trace. The developers can gradually adjust the strategies and attempt different combinations of state variables/expressions. Heuristics can be applied to identify such combinations. For instance, desirable state variables/expressions are likely built from static variables, singleton-member variable, or key-values in temp storage, e.g., in Redis [54].

We find that all the successful analyses are based on the following four tactics on comparing a failure trace and a success trace.

T1 (Single-Event Difference). The fault-revealing range is a single event, and the difference lies only in the descriptions (e.g., invoked method) of the event.

T2 (Single-Range Difference). The fault-revealing range involves different interaction sequences among nodes.

T3 (Multiple-Range Difference). The execution orders of multiple fault-revealing ranges are different.

T4 (Multiple-Request Difference). The execution orders of multiple user requests are different.

Among these tactics, T1 does not involve differences in node interaction sequences, while the other three tactics do. The tactics used for the analysis of each fault case are also shown in Table 7. It can be seen that tactics may be combined for debugging to locate a fault, because a fault may involve multiple fault-revealing ranges at different levels. The difficulty of debugging increases from T1 to T4 with the analysis of trace differences in a larger scope.

T4 is relatively hard to use, as it involves complex interactions among different user requests. F2 is an example for which T4 must be used. Due to the extensive usage of asynchronous interactions in microservice systems, the processing orders of user requests do not always correspond to their receiving order. If there are interactions among different user requests, it is likely that a fault will be introduced due to erroneous coordination of processing user requests. F2 is an example of this case. As the trace analysis involves a large number of events across multiple user requests, and the events of different requests are interleaved, F2 cannot be effectively analyzed based on existing visualization techniques, unless the differences between success and failure traces are reflected in the trace comparison of a single request. For F2, the developer spends a lot of time in seeking the root cause. But for F13, as the trace analysis involves a much smaller number of events across multiple user requests compared to F2, and the events of different requests on ShiViz can be easily distinguished, F13 can be effectively analyzed, and less time consumed.

5.2.4 *Quantitative Analysis*

The time-analysis results of debugging with improved trace visualization are shown in Table 7. Among the 12 fault cases, the developers fail in 2 cases (F3 and F4) in which they also fail with visual trace analysis. For F16, the developers succeed but use more time than visual trace analysis. These three cases are all **Environment** faults (F3 and F4 are non-functional, F16 is functional); such result suggests that debugging such faults cannot benefit from trace analysis.

In all the other 9 fault cases, the developers achieve improved debugging effectiveness with decreased average debugging time from 3.23 hours to 2.14 hours. Note that these 9 fault cases are all **Interaction** faults. For these faults, fault localization, initial understanding, failure reproduction are the three steps that benefit the most from the analysis. The time used for these steps is reduced by 49%, 28%, and 24%, respectively compared with visual trace analysis.

Table 7 also shows the detailed analysis processes of the developers on each of the 12 fault cases, including the used visualization strategy, number of nodes (#N.), number of events (#E.), number of user requests (#UR.), number of fault-revealing ranges identified in each analysis (#FR.), number of events in fault-revealing ranges in each analysis (#FE.), and hit (i.e., the analysis that succeeds in identifying at least one true fault-revealing range, 'Y' indicates successfully identified, 'N' indicates failed) in each analysis. A mark "-" indicates that the developers fail in identifying the ranges or events. The results show that each debugging with a combination of success and failure traces involves about 7-22 nodes (representing services or service states) and hundreds to thousands of events. These events belong to 2-7 user requests, and the traces of each user request are compared separately.

Each successful analysis identifies several fault-revealing ranges with dozens of events. For some fault cases (F1, F2, F7, F8, F10, and F13), service-level analysis can effectively identify the fault-revealing ranges (Hit is 'Y'). Some other fault cases (F5, F11, and F12) require service state-level analysis to identify the fault-revealing ranges. For F12, both developers successfully identify one of the issue states.

TABLE 7
Empirical Study Results of Debugging with Improved Trace Visualization

Fault	Overall Time (H)	Time of Each Step (H)							Strategy	#N.	#E.	#UR.	#FR.	#FE.	Hit	Tactic
		IU	ES	FR	FI	FS	FL	FF								
F1	2	0.6	-	-	0.6	0.6	0.2	0.1	Service	7	228	4	1	8	Y	T3
F2	2.4	0.6	0.6	0.4	0.4	0.4	0.6	0.3	Service	21	2706	3	2	9	Y	T4
F3	failed	1	0.6	0.6	0.8	failed	failed	failed	Service	11	398	2	2	39	N	T3
F4	failed	0.8	0.8	0.3	1	failed	failed	failed	Service	22	1704	5	-	-	N	T3
F5	2.2	0.6	0.4	0.3	0.4	0.3	0.2	0.1	State	17	972	2	3	255	N	T1, T2
F7	2.6	0.8	-	-	0.8	0.6	0.3	0.2	Service	22	1705	3	3	24	Y	T1
F8	2.8	0.6	0.6	0.6	0.4	0.4	0.4	0.2	Service	7	178	4	1	4	Y	T1
F10	2.1	0.6	0.2	0.2	0.3	0.3	0.4	0.1	Service	16	960	6	2	28	Y	T1
F11	2.3	0.4	0.4	0.3	0.4	0.4	0.4	0.1	State	9	303	4	3	24	Y	T1, T2, T3
F12	1.3	0.3	0.2	0.2	0.4	0.1	0.1	0.1	State	9	210	5	3	4	Y	T1, T2
F13	1.6	0.3	0.2	0.2	0.3	0.3	0.2	0.2	Service	12	372	7	3	57	Y	T4
F16	2.3	0.3	0.3	0.3	0.4	0.4	0.4	0.2	Service	6	233	2	2	3	N	T3

There are also unsuccessful cases (F3, F4), indicating that the developers fail in locating at least one fault-revealing range. The main reason is that the two cases are environmental faults, and they are not sufficiently supported by our debugging methodology.

5.3 Findings

Our study shows that most fault cases except those caused by environmental settings can benefit from trace visualization, especially those related to microservice interactions. By treating microservices or microservice states as the nodes, we can further improve the effectiveness of microservice debugging using state-of-the-art debugging visualization tools for distributed systems. A difficulty for state-level tracing and visualization mainly lies in the definition of microservice states. As a microservice system may have a large number of microservices and state variables/expressions, it highly depends on the experience of the developers to achieve effective and efficient fault analysis by identifying a few key states that can help reveal the faults.

A challenge for the trace visualization lies in the huge number of nodes and events. Large-scale industrial microservice systems have hundreds to thousands of microservices and tens of thousands to millions of events in a trace. Such a number of nodes and events can make the visualization analysis infeasible. This problem can be alleviated from two aspects. First, better trace visualization techniques such as zoom in/out and node/event clustering are required to allow the developers to focus on suspicious scopes. For example, node/event clustering can adaptively group cohesive nodes and events together, and thus reduce the number of nodes and events to be examined by progressively disclosing information. Second, fault localization techniques such as spectrum based fault localization [38], [55] and delta debugging [56] can be combined with visualization analysis for microservice debugging. On the one hand, the combination can suggest suspicious scopes in traces by applying statistical fault localization on microservice invocations, and on the other hand, the combination can provide results of code-level fault localization (e.g., code blocks) within specific microservices.

In view of the great complexity caused by the scale of microservice interactions and the dynamics of infrastructure, we believe that debugging of microservices needs to be supported in a data-driven way. For instance, one way is to combine human expertise and machine intelligence for guided visual exploration and comparison of traces. The supporting tools can take full advantage of the large amount of data produced by runtime monitoring and historical analysis and make critical suggestion and guidance during the visual exploration and comparison of traces. For example, the tools can suggest suspicious scopes in traces and sensitive state variables that may differentiate success and failure traces based on probabilistic data analysis, or recommend historical fault cases that share similar patterns of traces. Based on these suggestions and guidance, the developers can dig into possible segments of traces or add relevant state variables to trace comparison and visualization. These actions are in turn collected and used by the tools as feedback to improve further suggestion and guidance.

6 Threats to Validity

One common threat to the external validity of our studies lies in the limited participants and fault cases. The industrial experiences learned from these participants may not represent other companies or microservice systems that have different characteristics. The fault cases collected from the industrial participants may not cover more complex faults or other different fault types. One major threat to the internal validity of the industrial survey lies in the accuracy of the information (e.g., time of each debugging step) collected from the participants. As such information is not completely based on precise historical records, some of the information may not be accurate.

The threats to the external validity of the empirical study mainly lie in the representativeness of the benchmark system. The system currently is smaller and less complex (e.g., less heterogeneous) than most of the surveyed industrial systems, despite being the largest and most complex open source microservice system within our knowledge. Thus some experiences of debugging obtained from the study may not be valid for large industrial systems.

There are three major threats to the internal validity of the empirical study. The first one lies in the implementation of the fault cases based on our understanding. The understanding may be inaccurate, and the replication of some faults in a different system may not fully capture the essential characteristics of the fault cases. The second one lies in the uncertainty of runtime environments such as access load and network traffic. Some faults may behave differently with different environment settings, thus need different debugging strategies. The third one lies in the differences of the experiences and skills of the developers who participate in the study. These differences may also contribute to the differences of debugging time and results with different practices.

7 Related Work

Some researchers review the development and status of microservice research using systematic mapping study and literature review. Francesco et al. [2] present a systematic mapping study on the current state of the art on architecting microservices from three perspectives: publication trends, focus of research, and potential for industrial adoption. One of their conclusions is that research on architecting microservices is still in its initial phases and the balanced involvement of industrial and academic authors is promising. Alshuqayran et al. [57] present a systematic mapping study on microservice architecture, focusing on the architectural challenges of microservice systems, the architectural diagrams used for representing them, and the involved quality requirements. Dragoni et al. [58] review the development history from objects, services, to microservices, present the current state of the art, and raise some open problems and future challenges. Aderaldo et al. [23] present an initial set of requirements for a candidate microservice benchmark system to be used in research on software architecture. They evaluate five open source microservice systems based on these requirements and the results indicate that none of them is mature enough to be used as a community-wide research benchmark. Our open source benchmark system offers a promising candidate to fill such vacancy. Our industrial survey well supplements these previous systematic mapping studies and literature reviews.

There has been some research on debugging concurrent programs [38], [59], [60] and distributed systems [10], [61], [62], [63]. Asadollah et al. [64] present a systematic mapping study on debugging concurrent and multicore software in the decade between 2005 and 2014. Bailis et al. [61] present a survey on recent techniques for debugging distributed systems with a conclusion that the state-of-the-art of debugging distributed systems is still in its infancy. Giraldeau et al. [62] propose a technique to visualize the execution of distributed systems using scheduling, network, and interrupt events. Aguerre et al. [63] present a simulation and visualization platform that incorporates a distributed debugger. Beschastnikh et al. [10] discuss the key features and debugging challenges of distributed systems and present a debugging visualization tool named ShiViz, which our empirical study investigates and extends. In contrast to such previous research, our work is the first to focus on debugging support for microservice systems.

8 Conclusion

In this work, we have presented an industrial survey to conduct fault analysis on typical faults of microservice systems, current industrial practice of debugging, and the challenges faced by the developers. Based on the survey results, we have developed a medium-size benchmark microservice system (being the largest and most complex open source microservice system within our knowledge) and replicated 22 representative fault cases from industrial ones based on the system. These replicated faults have then been used as the basis of our empirical study on microservice debugging. The results of the study show that, by using proper tracing and visualization techniques or strategies, tracing and visualization analysis can help debugging for locating various kinds of faults involving microservice interactions. Our findings from the study also indicate that there is a need for more intelligent trace analysis and visualization, e.g., by combining techniques of trace visualization and improved fault localization, and employing data-driven and learning-based recommendation for guided visual exploration and comparison of traces.

Industrial microservice systems are often large and complex. For example, industrial systems are highly heterogeneous in microservice interactions and may use not only REST invocations and message queues but also remote procedure calls and socket communication. Moreover, industrial systems are running on highly complex infrastructures such as auto-scaling microservice cluster and service mesh (a dedicated infrastructure layer for service-to-service communication [65]). Such complexity and heterogeneity pose additional challenges on execution tracing and visualization. Our future work plans to further extend our benchmark system to reflect more characteristics of industrial microservice systems, and explore effective trace visualization techniques and the combination of fault localization techniques (e.g., spectrum-based fault localization [38] and delta debugging [56]) for microservice debugging. Moreover, we plan to explore a more technology-independent way to inject tracing information at every service invocation via service mesh tools such as Linkerd [66] and Istio [67].

Acknowledgments

This work was supported by the National Key Research and Development Program of China under Grant No. 2018YFB1004803. Tao Xie's work was supported in part by National Science Foundation under grants no. CNS-1513939, CNS-1564274, and CCF-1816615.

References

[1] J. Lewis and M. Fowler, "Microservices a definition of this new architectural term," 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html

[2] P. D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*, 2017, pp. 21–30.

[3] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic resilience testing of microservices," in *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*, 2016, pp. 57–66.

[4] Netflix.Com, "Netflix," 2018. [Online]. Available: https://www.netflix.com/

[5] SmartBear, "Why you can't talk about microservices without mentioning netflix," 2015. [Online]. Available: https://smartbear.com/blog/develop/why-you-cant-talk-about-microservices-without-ment/

[6] Wechat.Com, "Wechat," 2018. [Online]. Available: https://www.wechat.com/

[7] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, "Overload control for scaling wechat microservices," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, 2018, pp. 149–161.

[8] A. Deb, "Application delivery service challenges in microservices-based applications," 2016. [Online]. Available: http://www.thefabricnet.com/application-delivery-service-challenges-in-microservices-based-applications/

[9] Amazon.Com, "Amazon," 2017. [Online]. Available: https://d0.awsstatic.com/whitepapers/microservices-on-aws.pdf

[10] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems," *Commun. ACM*, vol. 59, no. 8, pp. 32–37, 2016.

[11] S. Hassan and R. Bahsoon, "Microservices and their design trade-offs: A self-adaptive roadmap," in *IEEE International Conference on Services Computing, SCC 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, 2016, pp. 813–818.

[12] G. Schermann, D. Schöni, P. Leitner, and H. C. Gall, "Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies," in *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*, 2016, p. 12.

[13] A. de Camargo, I. L. Salvadori, R. dos Santos Mello, and F. Siqueira, "An architecture to automate performance tests on microservices," in *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services, iiWAS 2016, Singapore, November 28-30, 2016*, 2016, pp. 422–429.

[14] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, "Performance engineering for microservices: Research challenges and directions," in *Companion Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, 2017, pp. 223–226.

[15] P. Leitner, J. Cito, and E. Stöckli, "Modelling and managing deployment costs of microservice-based cloud applications," in *Proceedings of the 9th International Conference on Utility and Cloud Computing, UCC 2016, Shanghai, China, December 6-9, 2016*, 2016, pp. 165–174.

[16] W. Hasselbring, "Microservices for scalability: Keynote talk abstract," in *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ICPE 2016, Delft, The Netherlands, March 12-16, 2016*, 2016, pp. 133–134.

[17] S. Klock, J. M. E. M. van der Werf, J. P. Guelen, and S. Jansen, "Workload-based clustering of coherent feature sets in microservice architectures," in *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*, 2017, pp. 11–20.

[18] A. Panda, M. Sagiv, and S. Shenker, "Verification in the age of microservices," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, 2017, pp. 30–36.

[19] I. L. Salvadori, A. Huf, R. dos Santos Mello, and F. Siqueira, "Publishing linked data through semantic microservices composition," in *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services, iiWAS 2016, Singapore, November 28-30, 2016*, 2016, pp. 443–452.

[20] G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. Salle, "Microart: A software architecture recovery tool for maintaining microservice-based systems," in *2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017*, 2017, pp. 298–302.

[21] J. Lin, L. C. Lin, and S. Huang, "Migrating web applications to clouds with microservice architectures," in *Applied System Innovation (ICASI), 2016 International Conference on*. IEEE, 2016, pp. 1–4.

[22] S. Hassan and R. Bahsoon, "Microservices and their design trade-offs: A self-adaptive roadmap," in *IEEE International Conference on Services Computing, SCC 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, 2016, pp. 813–818.

[23] C. M. Aderaldo, N. C. Mendonca, C. Pahl, and P. Jamshidi, "Benchmark requirements for microservices architecture research," in *1st IEEE/ACM International Workshop on Establishing the Community-Wide Infrecaseructure for Architecture-Based Software Engineering, ECASE@ICSE 2017, Buenos Aires, Argentina, May 22, 2017*, 2017, pp. 8–13.

[24] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[25] Microservice.System.Benchmark, "Trainticket," 2018. [Online]. Available: https://github.com/FudanSELab/train-ticket/

[26] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 323–324.

[27] Fault.Replication, "Fault replication," 2017. [Online]. Available: https://github.com/FudanSELab/train-ticket-fault-replicate

[28] Replication.Package, "Fault analysis and debugging of microservice systems," 2018. [Online]. Available: https://fudanselab.github.io/research/MSFaultEmpiricalStudy/

[29] SpringBoot.Com, "Spring boot," 2018. [Online]. Available: http://projects.spring.io/spring-boot/

[30] Dubbo.Com, "Dubbo," 2017. [Online]. Available: http://dubbo.io/

[31] Docker.Com, "Docker," 2018. [Online]. Available: https://docker.com/

[32] SpringCloud.Com, "Spring cloud," 2018. [Online]. Available: http://projects.spring.io/spring-cloud/

[33] Mesos.Com, "Mesos," 2018. [Online]. Available: http://mesos.apache.org/

[34] Kubernetes.Com, "Kubernetes," 2018. [Online]. Available: https://kubernetes.io/

[35] DockerSwarm.Com, "Docker swarm," 2018. [Online]. Available: https://docs.docker.com/swarm/

[36] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Software Eng.*, vol. 42, no. 8, pp. 707–740, 2016.

[37] R. A. Santelices, Y. Zhang, S. Jiang, H. Cai, and Y. Zhang, "Quantitative program slicing: separating statements by relevance," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 1269–1272.

[38] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "Spectrum-based multiple fault localization," in *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, 2009, pp. 88–99.

[39] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, 2005, pp. 15–26.

[40] W. E. Wong and Y. Qi, "Bp neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 4, pp. 573–597, 2009.

[41] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Software Eng.*, vol. 32, no. 8, pp. 608–624, 2006.

[42] E. H. da S. Alves, L. C. Cordeiro, and E. B. de Lima Filho, "Fault localization in multi-threaded C programs using bounded model checking," in *2015 Brazilian Symposium on Computing Systems Engineering, SBESC 2015, Foz do Iguacu, Brazil, November 3-6, 2015*, 2015, pp. 96–101.

[43] F. Koca, H. Sözer, and R. Abreu, "Spectrum-based fault localization for diagnosing concurrency faults," in *Testing Software and Systems - 25th IFIP WG 6.1 International Conference, ICTSS 2013, Istanbul, Turkey, November 13-15, 2013, Proceedings*, 2013, pp. 239–254.

[44] I. Laguna, D. H. Ahn, B. R. de Supinski, S. Bagchi, and T. Gamblin, "Probabilistic diagnosis of performance faults in large-scale parallel applications," in *International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012*, 2012, pp. 213–222.

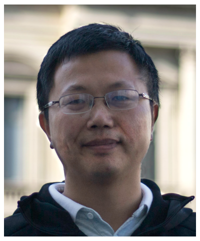
[45] G. Qi, L. Yao, and A. V. Uzunov, "Fault detection and localization in distributed systems using recurrent convolutional neural networks," in *Advanced Data Mining and Applications - 13th International Conference, ADMA 2017, Singapore, November 5-6, 2017, Proceedings*, 2017, pp. 33–48.

[46] A. B. Sharma, H. Chen, M. Ding, K. Yoshihira, and G. Jiang, "Fault detection and localization in distributed systems using invariant relationships," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*, 2013, pp. 1–8.

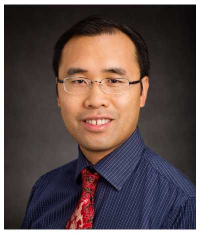
[47] C. Pham, L. Wang, B. Tak, S. Baset, C. Tang, Z. T. Kalbarczyk, and R. K. Iyer, "Failure diagnosis for distributed systems using

targeted fault injection," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, pp. 503–516, 2017.
[48] Log4j.Com, "Log4j," 2017. [Online]. Available: https://logging.apache.org/log4j/2.x/
[49] Logstash.Com, "Logstash," 2018. [Online]. Available: https://www.elastic.co/products/logstash
[50] Elasticsearch.Com, "Elasticsearch," 2018. [Online]. Available: https://www.elastic.co/products/elasticsearch
[51] Kibana.Com, "Kibana," 2018. [Online]. Available: https://www.elastic.co/products/kibana
[52] Dynatrace.Com, "Dynatrace," 2013. [Online]. Available: https://www.dynatrace.com/
[53] Zipkin.Com, "Zipkin," 2016. [Online]. Available: https://zipkin.io/
[54] Redis.Io, "redis.io," 2016. [Online]. Available: https://redis.io/
[55] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, 2012, pp. 378–381. [Online]. Available: https://doi.org/10.1145/2351676.2351752
[56] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002.
[57] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *9th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2016, Macau, China, November 4-6, 2016*, 2016, pp. 44–51.
[58] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *CoRR*, vol. abs/1606.04036, 2016.
[59] S. Park, R. W. Vuduc, and M. J. Harrold, "UNICORN: a unified approach for localizing non-deadlock concurrency bugs," *Softw. Test., Verif. Reliab.*, vol. 25, no. 3, pp. 167–190, 2015.
[60] ——, "Falcon: fault localization in concurrent programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 245–254.
[61] P. Bailis, P. Alvaro, and S. Gulwani, "Research for practice: tracing and debugging distributed systems; programming by examples," *Commun. ACM*, vol. 60, no. 7, pp. 46–49, 2017.
[62] F. Giraldeau and M. Dagenais, "Wait analysis of distributed systems using kernel tracing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 8, pp. 2450–2461, 2016.
[63] C. Aguerre, T. Morsellino, and M. Mosbah, "Fully-distributed debugging and visualization of distributed systems in anonymous networks," in *GRAPP & IVAPP 2012: Proceedings of the International Conference on Computer Graphics Theory and Applications and International Conference on Information Visualization Theory and Applications, Rome, Italy, 24-26 February, 2012*, 2012, pp. 764–767.
[64] S. A. Asadollah, D. Sundmark, S. Eldh, H. Hansson, and W. Afzal, "10 years of research on debugging concurrent and multicore software: a systematic mapping study," *Software Quality Journal*, vol. 25, no. 1, pp. 49–82, 2017.
[65] W. Morgan, "What's a service mesh? and why do i need one?" 2017. [Online]. Available: https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/
[66] Linkerd, "Linkerd," 2018. [Online]. Available: https://linkerd.io/
[67] Istio, "Istio," 2018. [Online]. Available: https://istio.io/

Xiang Zhou is a PhD student of the School of Computer Science at Fudan University, China. He received his master degree from Tongji University in 2009. His PhD work mainly concerns on the development and operation of microservice systems.



Xin Peng is a professor of the School of Computer Science at Fudan University, China. He received Bachelor and PhD degrees in computer science from Fudan University in 2001 and 2006. His research interests include data-driven intelligent software development, software maintenance and evolution, mobile and cloud computing. His work won the Best Paper Award at the 27th International Conference on Software Maintenance (ICSM 2011), the ACM SIGSOFT Distinguished Paper Award at the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2018), the IEEE TCSE Distinguished Paper Award at the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME 2018).



Tao Xie is a professor and Willett Faculty Scholar in the Department of Computer Science at the University of Illinois at UrbanaChampaign, USA. His research interests are software testing, program analysis, software analytics, software security, intelligent software engineeirng, and educational software engineering. He is a Fellow of the IEEE.

Jun Sun is currently an associate professor at Singapore University of Technology and Design (SUTD). 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 of SUTD since 2010. He was a visiting scholar at MIT from 2011-2012. Jun's research interests include software engineering, formal methods, program analysis and cyber-security.

Chao Ji is a Master student of the School of Computer Science at Fudan University, China. He received his Bachelor degree from Fudan University in 2017. His work mainly concerns on the development and operation of microservice systems.

Wenhai Li is a Master student of the School of Computer Science at Fudan University, China. He received his Bachelor degree from Fudan University in 2016. His work mainly concerns on the development and operation of microservice systems.

Dan Ding is a Master student of the School of Computer Science at Fudan University, China. She received her Bachelor degree from Fudan University in 2017. Her work mainly concerns on the development and operation of microservice systems.