Latent Error Prediction and Fault Localization for Microservice Applications by Learning from System Trace Logs

Xiang Zhou[*]
Fudan University
China

Xin Peng[*][†]
Fudan University
China

Tao Xie
University of Illinois at Urbana-Champaign
USA

Jun Sun
Singapore Management University
Singapore

Chao Ji[*]
Fudan University
China

Dewei Liu[*]
Fudan University
China

Qilin Xiang[*]
Fudan University
China

Chuan He[*]
Fudan University
China

ABSTRACT

In the production environment, a large part of microservice failures are related to the complex and dynamic interactions and runtime environments, such as those related to multiple instances, environmental configurations, and asynchronous interactions of microservices. Due to the complexity and dynamism of these failures, it is often hard to reproduce and diagnose them in testing environments. It is desirable yet still challenging that these failures can be detected and the faults can be located at runtime of the production environment to allow developers to resolve them efficiently. To address this challenge, in this paper, we propose MEPFL, an approach of latent error prediction and fault localization for microservice applications by learning from system trace logs. Based on a set of features defined on the system trace logs, MEPFL trains prediction models at both the trace level and the microservice level using the system trace logs collected from automatic executions of the target application and its faulty versions produced by fault injection. The prediction models thus can be used in the production environment to predict latent errors, faulty microservices, and fault types for trace instances captured at runtime. We implement MEPFL based on the infrastructure systems of container orchestrator and service mesh, and conduct a series of experimental studies with two open-source microservice applications (one of them being the largest open-source microservice application to our best knowledge). The results indicate that MEPFL can achieve high accuracy in intra-application prediction of latent errors, faulty microservices, and fault types, and outperforms a state-of-the-art approach of failure diagnosis for distributed systems. The results also show that MEPFL can effectively predict latent errors caused by real-world fault cases.

CCS CONCEPTS

• **Software and its engineering** → **Cloud computing**; **testing and debugging**.

KEYWORDS

microservices, error prediction, fault localization, tracing, debugging, machine learning

ACM Reference Format:
Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent Error Prediction and Fault Localization for Microservice Applications by Learning from System Trace Logs. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338961

1 INTRODUCTION

Industrial microservice applications have dozens to thousands of microservices running on hundreds to tens of thousands machines. In the production environment, microservice applications typically are fragile [26], often failing due to infrastructure (e.g., network, hardware) failures or application faults, and it is challenging to reproduce and diagnose these microservice application failures in testing environments. Infrastructure failures can be predicted at the cloud infrastructure layer and handled via infrastructure-level adaptations such as the allocation and migration of virtual machines [32]. Application faults traditionally rely on developers to detect and fix via code review, testing, and debugging. However, microservice application faults are complicated due to the complex and dynamic interactions and runtime environments [61]. A microservice can

[*]X. Zhou, X. Peng, C. Ji, D. Liu, Q. Xiang, and C. He are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China and the Shanghai Institute of Intelligent Electronics & Systems, China.
[†]X. Peng is the corresponding author (pengxin@fudan.edu.cn).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338961

have several to thousands of instances that are dynamically created and destroyed according to the scaling requirements. These instances run with complex environmental configurations such as memory/CPU limits of microservices and containers [62, 63]. Microservice applications usually have complex invocation chains, each involving several to dozens of microservice invocations, most of which are asynchronous. Improper environmental configurations or coordination of microservice instances or asynchronous interactions may result in failures at runtime.

To allow developers to resolve microservice application failures efficiently in the production environment, it is desirable and yet challenging that these microservice application failures can be detected and the faults can be located at runtime of the production environment, e.g., based on application logs or system logs. Application logs record the internal status and events during the execution of an application. These application logs are produced by logging statements written by developers in the application. However, application logging is often done in an arbitrary and ad hoc manner and the log messages often contain limited information for failure diagnosis [56, 57]. On the other hand, some existing approaches use system logs to detect anomalies or predict failures of cloud-based systems. These system logs are produced by infrastructure systems such as operating systems (e.g., CentOS [42]), distributed file systems (e.g., HDFS [11]), and container orchestrators (e.g., Kubernetes [30]). However, these approaches can only uncover anomalies or failures within the cloud infrastructure such as computing nodes [32] and storage [24].

To address this challenge, in this paper, by learning from system trace logs, in short as trace logs (a special type of system logs), we focus on predicting three common types of microservice application faults that are specifically relevant to microservice interactions and runtime environments, i.e., multi-instance faults, configuration faults, asynchronous interaction faults. According to an existing empirical study [61], almost half of all studied microservice application faults are of these three types. Different from internal-logic faults, which are localized often based on analyzing internal states and logics, these three types of faults are usually relevant to external characteristics of trace logs such as the number of microservice instances, accesses of global variables/local cache, resource consumption, invocation and execution orders. Recently Pham et al. [40] propose a failure diagnosis approach for distributed systems. This approach locates faults by matching similar failure traces produced using fault injection. This matching-based approach has very limited generalization ability and cannot support failure recognition. In contrast, in our work, we extract a set of features that reflect the dynamic environments and interactions of microservices from trace logs and train a set of prediction models based on the features. The prediction models can be used to not only locate the faults but also recognize the failures by predicting latent errors caused by the faults.

In particular, in this paper, we propose MEPFL (Microservice Error Prediction and Fault Localization), a novel approach for latent error prediction and fault localization of microservice applications by learning from trace logs. We design MEPFL to predict latent errors of a microservice application caused by the preceding three common types of microservice application faults. MEPFL further predicts the locations (i.e., microservices) and types of the faults.

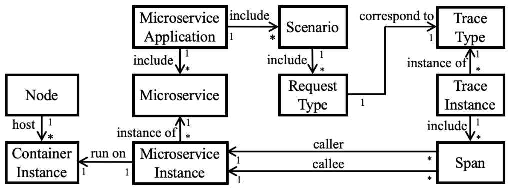


Figure 1: Basic Concepts

MEPFL combines trace-level and microservice-level prediction. The trace-level prediction utilizes application-specific context (e.g., microservices involved in an application) in its feature vectors and treats a trace instance (which records the process of microservice invocations for a user request) as a whole during training and prediction. It includes three models to predict latent errors, relevant microservices, and fault types, respectively. The microservice-level prediction uses a set of general features for microservices in a context-free way. It is based on a model to predict what type of fault the target microservice has. All these preceding models used in trace-level or microservice-level prediction are defined on a set of trace log features selected by correlation analysis.

With two open-source microservice applications (Sock Shop [48] and TrainTicket [36], which is the largest open-source microservice application to our best knowledge), we conduct a series of experimental studies to evaluate the effectiveness of MEPFL[1]. The results show that MEPFL can achieve high recall/precision (0.982/0.998 and 0.896/0.995) and low false positive rate (0.009 and 0.014) in latent error prediction, high Top-1 accuracy (0.933 and 0.937) in faulty microservice prediction, and high recall/precision (0.952/0.983 and 0.930/0.986) in fault type prediction. MEPFL substantially outperforms Pham et al.'s approach [40] in terms of the accuracy of the prediction of faulty microservices and fault types. With the increase of the coverage of trace types in the training data, the overall prediction more and more relies on the trace-level prediction, and the prediction accuracy substantially increases. To further evaluate the effectiveness of MEPFL in localizing real-world microservice application faults, we conduct an experimental study with the fault cases [10] provided by the TrainTicket benchmark; these fault cases are replicated from industrial fault cases. The results show that MEPFL can accurately predict the latent errors caused by these fault cases with a recall of 0.647-0.983 (0.821 on average) and a precision of 0.586-0.984 (0.891 on average).

In this paper, we make the following main contributions.

- A set of features and prediction models for predicting latent errors, faulty microservices, and fault types for microservice applications based on system trace logs.
- A learning-based approach for latent error prediction and fault localization of microservice applications based on fault injection and testing.
- A series of experimental studies with two open-source microservice benchmarks to evaluate the effectiveness of the approach.

2 PRELIMINARIES

Our approach is defined based on a series of concepts about microservice. Figure 1 shows the relationships between these concepts.

[1]All the data of the studies can be found in our replication package [46].

Table 1: Fault Cases Reported by Our Previous Work [61]

Fault Type	Fault Cases	Example
Monolithic	F6, F9, F10, F14-22	F22: a wrong column name included in the constructed SQL statement
Multi-Instance	F8, F11, F12	F12: unexpected output of a microservice when it is in a special state
Configuration	F3, F4, F5, F7	F3: improper configurations of JVM and Docker
Asynchronous Interaction	F1, F2, F13	F1: asynchronous message delivery that lacks sequence control

A microservice application includes a set of scenarios and each scenario includes multiple types of user requests (e.g., different clicks on the same "OK" button on the same web page result in different user requests of the same user request type). Each user request type is mapped to a trace type. Each trace instance records the process of microservice invocations for a user request. The trace type for the trace instance is the trace type mapped from the type of the user request. The trace instance includes a series of spans and each span represents an invocation between two microservice instances (i.e., a caller and a callee).

For example, the TrainTicket application has a Ticket Reservation scenario that includes several user request types (i.e., trace types) such as Query Available Tickets, Confirm Ticket Selection, and Pay. The trace instances of Confirm Ticket Selection include a series of microservice invocations, e.g., the Reserve service invokes the Ticket Info service, Food service, and several other services, and the Ticket Info service in turn invokes the Price service and other services.

A recent trend of microservice applications is the introduction of container orchestrator and service mesh. A container orchestrator such as Kubernetes (k8s) [30] dynamically schedules and manages a large number of container instances. In our system implementation, Kubernetes is used to dynamically deploy microservice applications and manage microservice instances and their environmental configurations during the offline training phase. Service mesh [37] is introduced as a separate layer that handles service-to-service communication. The most recognized implementation is Istio [25]. Our system implementation uses Istio to manage asynchronous microservice interactions during offline training and collect trace logs during both offline training and online prediction.

3 PREDICTION MODELS

3.1 Fault Types

Collected in an industrial survey, our previous work [61] reported 22 representative microservice fault cases (see Table 1). These fault cases can be categorized into the following four types.

- **Monolithic Faults**. These faults can cause failures even when the application is deployed in a monolithic mode, i.e., all the microservices are deployed on one node, each microservice has only one instance, and different microservices interact synchronously. These faults are usually due to faults in the internal implementation of microservices and are irrelevant to the runtime environments of the application.
- **Multi-Instance Faults**. These faults are related to the existence of multiple instances of the same microservice at runtime. They are often due to lacking coordination among different instances of the same microservice (e.g., to keep them in consistent states).
- **Configuration Faults**. These faults are related to the environmental configurations of microservices, such as the resource (e.g., memory and CPU) limits. They are often due to improper or inconsistent configurations of microservices and/or their residing environments (e.g., containers and virtual machines (VMs)).
- **Asynchronous Interaction Faults**. These faults are related to the asynchronous invocations among microservices and may cause failures when asynchronous invocations are executed or returned in an unexpected order. They are often due to missing or improper coordination of the sequences of asynchronous invocations.

The mechanisms of monolithic faults are similar to those of faults in monolithic applications. For example, F22 is a monolithic fault caused by an incorrect SQL statement. The failures caused by these faults can be reproduced and debugged on a single machine.

Multi-instance faults are mainly related to inconsistent states between different instances of the same microservice. For example, F12 is caused by an inconsistent state of a microservice. When an invocation chain involves two instances of the same service and their states are different, the request will be denied with an unexpected output. To fix the fault, the developers can add a synchronization mechanism for the involved state variable or move it to external storage (e.g., Redis). Statelessness is a recommended practice for microservice development. However, many microservice applications in practice are migrated from legacy monolithic applications, which are often stateful with state variable definitions (i.e., in-memory session states) [20]. Due to the difficulty of refactoring, the migrations are often halfway, and stateful microservices are common in enterprise applications.

Configuration faults are related to the complex environmental settings of microservices. A microservice is associated with a series of application-level environmental configurations such as the configurations of JVM (e.g., max/min heap memory, stack memory, garbage collection strategy), thread pool, and database connection pool. A microservice is also associated with a series of infrastructure-level environmental configurations such as the configurations of containers (e.g., memory limit, CPU limit, health check, communication security) and VMs. Reconciling these configurations is a challenging task and often incurs mistakes. For example, F3 is due to a conflict between the memory limits of JVM and Docker; this conflict can cause the JVM process to be killed.

Asynchronous interaction faults are usually caused by an unexpected order of executing asynchronous microservice invocations. When a microservice invokes several other microservices asynchronously, the orders of the requests, executions, and responses of the invoked microservices are uncertain. If the developers have unrealistic assumptions, e.g., the invoked microservices shall be executed and returned in the same order of invocations, the developers may introduce asynchronous interaction faults into the applications. For example, F1 lies in an unexpected order of executions and returning of multiple asynchronously invoked microservices and can cause an invoked microservice to be in an abnormal status.

Table 1 shows that 12 out of the 22 fault cases (54.5%) are monolithic faults, while the other 10 fault cases (45.5%) are related to multi-instance, environmental configuration, or asynchronous interaction. Monolithic faults can be found on a single machine by unit or integration testing, and can be located using traditional techniques of fault localization such as assertions and breakpoints [55]. Therefore, we focus on the other three types of faults in this paper.

3.2 Feature Definition

MEPFL aims to predict whether a latent error has occurred, the faulty microservice, and the fault type. MEPFL systematically extracts a set of features from trace instances. Figure 2 shows an example of trace instance, which starts from a root microservice *R*. In the figure, each arrow represents an invocation (i.e., span) and different subscripts of the same capital letter (e.g., D_1 and D_2) represent different invocations of the same microservice (the same or different instances). For this trace instance, the microservice features are calculated for each of the 11 microservice invocations (e.g., R, A_1, D_1, and D_2).

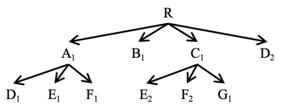

$$\begin{tikzcd}[column sep=small]
 & & & & R \arrow[dlll] \arrow[dl] \arrow[dr] \arrow[drrr] & & & \\
 & A_1 \arrow[dl] \arrow[d] \arrow[dr] & & B_1 & & C_1 \arrow[dl] \arrow[d] \arrow[dr] & & D_2 \\
D_1 & E_1 & F_1 & & E_2 & F_2 & G_1 &
\end{tikzcd}$$

Figure 2: An Example of Trace Instance

Based on the data collected in a previously reported empirical study [61], we summarize a comprehensive set of microservice features from different aspects (i.e., configuration, resource, instance, and interaction). These features can be extracted from trace logs. A feature is defined in one of three scopes: "global" indicates that the feature is a global property of the microservice and is irrelevant to the current trace instance; "trace" indicates that the feature is an overall property of the microservice in the current trace instance; "invocation" indicates that the feature is a property of the current microservice invocation. For example, for the trace instance shown in Figure 2, a "global" feature of D_1 depends on all the instances of the microservice D at that time; a "trace" feature of D_1 depends on D_1 and D_2 in the trace instance; an "invocation" feature of D_1 depends on only D_1 itself in the trace instance. Table 2 shows a complete list of the candidate features. Detailed descriptions of the features can be found in our replication package [46].

Configuration features reflect the environmental configurations of the microservice instance. Resource features reflect the resource consumptions (e.g., memory and CPU) of the microservice instance and its residing node. These features may be relevant to the validity of microservice configurations. Volume support, which enables the microservice to access the persistent data of the node, may also be relevant to multi-instance faults.

Instance features reflect the status of the deployment of the instances of the microservice and their involvement in the current trace instance. These features may be relevant to multi-instance problems.

Interaction features reflect the status of interactions (especially asynchronous interactions) with other microservices. Among these features, ET and RSC reflect the status of the current microservice invocation itself; AIT and CEO reflect the status of asynchronous invocations to other microservices. For example, for A_1 in Figure 2, the ET and RSC features reflect the execution time and response code of A_1 itself; the AIT and CEO features reflect the status of the invocations (which are asynchronous invocations) from A_1 to D_1, E_1, and F_1. To provide a unified encoding for different microservices, we develop the following convention for the AIT and CEO features: (1) assuming that there are N microservices asynchronously invoked by the current microservice, sort the invoked microservices by the invocation order and number them from 1 to N; (2) for each invoked microservice, extract a feature AIT-i ($1 \leq i \leq N$) for capturing its execution time; (3) for each pair of invoked microservices, set a feature CEO-i-j ($1 \leq i, j \leq N$) for capturing whether their execution order is consistent with their invocation order. For example, for A_1 in Figure 2, there are 3 AIT features, and AIT-1 for capturing the execution time of D_1; there are 3 CEO features, and CEO-1-2 for capturing whether the execution order of D_1 and E_1 is consistent with their invocation order. These features can be relevant to asynchronous interaction faults.

The preceding features can be used in the microservice-level prediction model. To support trace-level prediction, we further derive a set of trace-level features for each trace instance by absorbing the features of each microservice involved in the trace instance. If a microservice is invoked multiple times in the trace instance, we treat the feature values of the last invocation as the values of the corresponding trace-level features. Because the influences of many faults are cumulative and some other faults terminate traces at the last invocation, the last invocation of a microservice may provide the most significant indication for latent errors. For example, for the trace instance in Figure 2, there are 8 RSC features (i.e., *R*.RSC, *A*.RSC, *B*.RSC, *C*.RSC, *D*.RSC, *E*.RSC, *F*.RSC, and *G*.RSC), each corresponding to an involved microservice; and *D*.RSC is the RSC value of D_2. These trace-level features incorporate application-specific context, and thus can better predict latent errors and fault locations if the models have been trained with trace instances of similar trace types. In addition to these derived features, we also define 3 trace-level features that aggregate the execution information of involved microservices, i.e., TET (the execution time of the current trace instance), TMN (the number of microservices that are invoked in the current trace instance), and TIN (the number of microservice instances that are invoked in the current trace instance).

3.3 Model Design

MEPFL supports four prediction models, which allow us to combine both trace-level and microservice-level prediction. Trace-level prediction includes latent error prediction, faulty microservice prediction, and fault type prediction, each supported by a prediction model (i.e., the LE model, FM model, and FT model). Microservice-level prediction includes a microservice-status prediction model (in short as an MS model).

Trace-level prediction models are defined on trace-level features. These models incorporate application-specific microservices and their interactions in their feature definitions, and thus can utilize application-specific context in the prediction when similar trace instances (e.g., trace instances that share common microservices with the current trace instance) have been included in the training corpus. The microservice-level prediction model, on the other hand, is defined on microservice-level features in an application-independent way, and thus generalizes across different applications.

All the four prediction models are classification models. The three trace-level models treat the given trace instance as a whole. The LE model is a binary-classification model that classifies the

Table 2: Feature Definition

Category	Feature	Description	Scope
Configuration	ML	memory limit of the current microservice relative to the node memory limit	Invocation
	CL	CPU limit of the current microservice relative to the node CPU limit	Invocation
	VS	whether volume support is enabled for the current microservice	Global
Resource	MC	memory consumption of the current microservice instance relative to its memory limit	Invocation
	CC	CPU consumption of the current microservice instance relative to its CPU limit	Invocation
	NMC	memory consumption of the current node relative to its memory limit	Invocation
	NCC	CPU consumption of the current node relative to its CPU limit	Invocation
Instance	IN	number of instances for the current microservice in the whole system	Global
	IIN	number of instances (for the current microservice) invoked in the current trace instance	Trace
	SVA	ratio of the shared variables that are accessed in the current invocation	Invocation
	CA	whether the cache is accessed in the current invocation	Invocation
	SA	whether the third-party storage (e.g., database) is accessed in the current invocation	Invocation
	TN	number of threads of the current microservice instance	Invocation
	LT	life time of the current microservice instance since its creation	Invocation
	NIN	number of microservice instances residing in the current node	Invocation
	NMN	number of microservices whose instances reside in the current node	Invocation
Interaction	ET	execution time of the current microservice invocation	Invocation
	RSC	HTTP response status code of the current microservice invocation	Invocation
	AIT	time of an asynchronous invocation in the current microservice execution	Invocation
	CEO	whether the execution order of a pair of asynchronous invocations is consistent with their invocation order	Invocation

trace instance into two classes (with error or not); the FM model is a multi-label classification model that predicts one or multiple microservices in the trace instance to be faulty; and the FT model is a multi-label classification model that predicts one or multiple fault types. The microservice-level model treats the microservices involved in the given trace instance individually. The MS model is a single-label classification model that predicts an error status, which can be one of the three fault types or a special type "No Fault".

4 APPROACH

Figure 3 provides an overview of MEPFL, which includes an offline training phase and an online prediction phase. The offline training phase executes the target microservice application in the testing environment and trains the prediction models based on the collected trace logs. To train the models, we need not only trace logs under successful executions but also trace logs under erroneous executions. Therefore, we design a series of fault injection strategies to systematically obtain a series of faulty versions of the application. We use existing automated test cases to drive the application and its faulty versions to execute and produce trace logs. These test cases simulate user requests from the clients (e.g., browsers and mobile apps) and trigger microservice invocation chains. To ensure the diversity of traces, an additional control is imposed on the runtime environment to make the application executed under different settings. We then prepare a training corpus for the prediction models by collecting and analyzing trace logs from the executions. For each trace instance, we automatically construct a trace-level training sample and a set of microservice-level training samples by feature extraction and error/fault labeling. Based on the training corpus, we use machine learning techniques (e.g., Random Forests, K-Nearest Neighbors, Multi-Layer Perceptron) to train three prediction models that can predict for each trace whether a latent error occurs, which microservice the fault resides in, and what type of fault it is, respectively.

The online prediction phase monitors the execution of the application in the production environment and predicts latent errors and fault locations (faulty microservices and fault types). It continuously collects and analyzes trace logs from the running application. For each trace instance, it extracts features in a similar way to what has been done in preparing the training corpus, and then uses the prediction models to predict whether a latent error occurs, which microservice the fault resides in, and what type of fault it is.

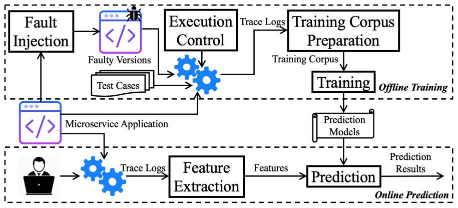


Figure 3: Approach Overview

4.1 Fault Injection

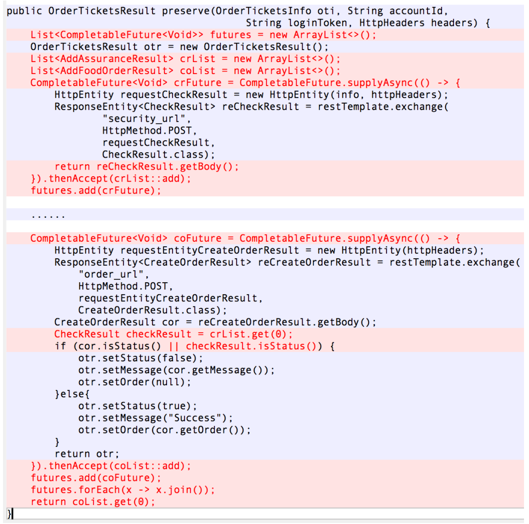
Fault injection produces a series of faulty versions of the target microservice application by introducing different types of faults into different parts of the application. For each faulty version, we inject a specific type of fault into a specific microservice in a semi-automatic way. We design a fault injection strategy for each type of fault. Each fault injection strategy is parameterized with the following information: (1) a **precondition** that specifies the conditions to be met by the location of a microservice where the fault is injected; (2) a **code transformation method** that specifies how to generate a patch to transform a microservice into a faulty one; (3) an **expected failure symptom** that specifies the expected symptom if the injected fault has caused a failure.

We next present details of the parameters for each fault type.

Multi-Instance Faults. The precondition is that the microservice accesses third-party cache databases such as Redis [44]. The code transformation method is to replace the data accesses to third-party cache databases with data accesses to global variables or local cache. As multi-instance faults usually cause incorrect data accesses, the expected failure symptom includes injected failure response, incorrect output, or various assertion failures or exceptions (e.g., null pointer exceptions).

Configuration Faults. Configuration faults are usually related to consumption of various resources (e.g., memory and CPU) and irrelevant to the business logics of microservices. Therefore, configuration faults can be injected into any part of any microservice. The code transformation method is to inject resource-intensive code into target locations of microservices, e.g., memory-intensive code that loads a lot of data into memory or computing-intensive code that includes a lot of iterations. The expected failure symptom includes microservice instance restarted, timeout exception, or resource-consumption-related exceptions (e.g., out-of-memory exceptions).

Asynchronous Interaction Faults. The precondition for asynchronous interaction faults is that the microservice invokes multiple



```
public OrderTicketsResult preserve(OrderTicketsInfo oti, String accountId,
                                        String loginToken, HttpHeaders headers) {
    List<CompletableFuture<Void>> futures = new ArrayList<>();
    OrderTicketsResult otr = new OrderTicketsResult();
    List<AddAssuranceResult> crList = new ArrayList<>();
    List<AddFoodOrderResult> coList = new ArrayList<>();
    CompletableFuture<Void> crFuture = CompletableFuture.supplyAsync(() -> {
        HttpEntity requestCheckResult = new HttpEntity(info, httpHeaders);
        ResponseEntity<CheckResult> reCheckResult = restTemplate.exchange(
                "security_url",
                HttpMethod.POST,
                requestCheckResult,
                CheckResult.class);
        return reCheckResult.getBody();
    }).thenAccept(crList::add);
    futures.add(crFuture);

    ......

    CompletableFuture<Void> coFuture = CompletableFuture.supplyAsync(() -> {
        HttpEntity requestEntityCreateOrderResult = new HttpEntity(httpHeaders);
        ResponseEntity<CreateOrderResult> reCreateOrderResult = restTemplate.exchange(
            "order_url",
            HttpMethod.POST,
            requestEntityCreateOrderResult,
            CreateOrderResult.class);
        CreateOrderResult cor = reCreateOrderResult.getBody();
        CheckResult checkResult = crList.get(0);
        if (cor.isStatus() || checkResult.isStatus()) {
            otr.setStatus(false);
            otr.setMessage(cor.getMessage());
            otr.setOrder(null);
        }else{
            otr.setStatus(true);
            otr.setMessage("Success");
            otr.setOrder(cor.getOrder());
        }
        return otr;
    }).thenAccept(coList::add);
    futures.add(coFuture);
    futures.forEach(x -> x.join());
    return coList.get(0);
}
```

Figure 4: Injection of Asynchronous Interaction Fault

microservices in succession. The code transformation method is to introduce data dependencies between different microservice invocations. If the invocations are synchronous, we need to first transform them into asynchronous ones. The expected failure symptom includes injected failure response, or various assertion failures or exceptions (e.g., null-pointer exceptions).

Figure 4 shows an example of injecting an asynchronous interaction fault, where the red code is introduced for fault injection. This example includes two synchronous microservice invocations. The code transformation changes the invocations to asynchronous ones using Java *CompletableFuture* and introduces objects (*crList* and *coList*) to store the return values. It introduces a data dependency between the two asynchronous invocations via the variable *checkResult*. Due to the dependency, the program can enter an unexpected state if the two invoked microservices are executed in a different order.

To inject a fault of a specific type into the target application, we first identify a location (in the application) that satisfies the precondition and use the code transformation method to introduce a fault at the location. Then we check whether the resulting faulty microservice can be successfully compiled and if not we manually repair the compilation problem. After that, we further verify the fault injection result by executing the corresponding test cases multiple times (e.g., 30 times). As the failures caused by these faults are probabilistic, the fault injection is regarded to be successful if the expected failure symptom is observed in one of the executions. Each successfully injected fault is incorporated into the target application to produce a faulty version.

4.2 Execution Control

The execution controller automatically deploys the target application and its faulty versions, executes different application versions with existing automated test cases, and collects trace logs. For faulty versions, the controller further manipulates the runtime environment of the application so that it is executed under different settings, including different numbers of microservice instances, environmental configurations, and asynchronous interaction sequences. For each setting, the controller executes the test cases involving the faulty microservice for a given number (e.g., 10) of times to get multiple trace instances under the same setting. The automatic deployment, execution manipulation, and trace log collection are implemented based on the infrastructure support of container orchestrator and service mesh (see Section 5).

For a faulty version with an injected multi-instance fault, the controller manipulates the number of instances of the faulty microservice. The controller executes the faulty microservice with 1 to a given number (e.g., 10) of instances while the other microservices running with the default numbers of instances.

For a faulty version with an injected configuration fault, the controller manipulates the environmental configurations (e.g., memory limit, CPU limit) of the faulty microservice, e.g., with the memory limit or CPU limit increased from 1% to 10%, while the other microservices running with default configurations.

For a faulty version with an injected asynchronous interaction fault, the controller manipulates the execution sequences of asynchronous invocations. The controller causes the microservices that are asynchronously invoked by the faulty microservice to be executed in different orders. For example, for a microservice *A* that asynchronously invokes three microservices *B*, *C*, and *D* in that order, the controller first forces the invoked microservices to be executed in the same order of invocation (i.e., *B*, *C*, *D*) and then forces them to be executed in other orders (e.g., *D*, *C*, *B*). As the microservices that are asynchronously invoked by a microservice in a trace instance are usually not so many (e.g., fewer than 6), the controller can try different execution orders to achieve certain coverage, e.g., covering all the partial orders of any pair of microservices.

During the executions of the application and its faulty versions, the controller continually collects the trace logs. To provide the features required by the prediction models, the trace logs record not only the sequence of the spans (i.e., microservice invocations) of each trace instance, but also the environmental configurations, resource consumptions, and number of the instances of the involved microservices.

4.3 Preparation of Training Corpus

For each trace instance, we generate a trace-level training sample by extracting trace-level features and labeling its error status, fault type, and fault location. For each microservice involved in the trace instance, we generate a microservice-level training sample by extracting microservice-level features and labeling its fault status.

We extract the features based on the feature definitions given in Section 3.2. We first extract microservice-level features based on the spans included in the trace instance, and then aggregate these microservice-level features into trace-level features.

The error status of a trace instance can be 0 or 1. 0 indicates that the trace instance has a latent error caused by one of the three types of faults, and 1 indicates the other cases. Note that the other cases include that the trace instance has no errors or has other unknown errors (e.g., errors caused by monolithic faults). We label the error status of a trace instance as 0 if the following four conditions are satisfied. First, the result of the trace instance is not consistent with the expected result in the test case. Second, the trace instance is produced by a faulty version of the application, and the faulty microservice is included in the trace instance. Third, the trace instance is produced in a faulty setting, i.e., for a multi-instance fault, the faulty microservice has more than one instance; for a configuration fault, the faulty microservice runs with resource limits (e.g., memory or CPU limit); for an asynchronous interaction fault, the execution order of the microservices asynchronously invoked by the faulty microservice is not the same as their invocation order. Fourth, the error is not caused by monolithic faults. To check whether the fourth condition is satisfied, we automatically deploy the same application version in a monolithic environment and run the same test case; if the test case passes, we conclude that the error is not caused by monolithic faults.

If the trace error status is 1, we label the error status of the corresponding trace-level training sample as 1 and the fault status of each microservice-level training sample to "No Fault".

If the trace error status is 0, we label the error status of the corresponding trace-level training sample to 0, its fault type to the injected fault type, and its fault location to the faulty microservice. For each microservice-level training sample, we label its fault status to the injected fault type if it has an injected fault and "No Fault" if it has no injected fault.

4.4 Training

To select relevant features for the training, we conduct a correlation analysis between the errors caused by the three types of faults and the candidate features (see Table 2) based on TrainTicket [36]. For each trace type, we use the corresponding test cases to run the original application and its faulty versions with different types of injected faults 1,000 times. During the process, we manipulate the running system so that it is executed with different settings (e.g., the numbers of instances, environmental configurations, and execution orders of asynchronous invocations).

Table 3 lists the Pearson correlation coefficient [8] between the errors caused by different types of faults and the candidate features for five trace types. The complete correlation analysis results can be found in our replication package [46]. We choose the features whose correlation coefficients are higher than 0.3 for at least one fault type and one trace type. Based on the criterion, we select all the candidate features except VS, SA, LT, NMN and their corresponding trace-level features for the training.

Before training the prediction models, we follow a standard process to preprocess the training data, including imputation, category encoding, discretization, and feature scaling. Imputation replaces missing data in the training samples with typical values, i.e., the means for continuous values and the most frequent values for other kinds of values. Category encoding transforms categorization values (e.g., RSC) into integer values from 0 to N-1 (N is the number of categories of a feature). Discretization transforms continuous values (e.g., TN) into integer values from 0 to K-1 by dividing the range of the values into K partitions. Feature scaling normalizes the ranges of feature values into [0, 1].

We choose the following three machine learning techniques to train the prediction models.

- Random Forests (RF): an ensemble learning technique for classification that constructs a multitude of decision trees at training time and outputs the class that is the mode of the classes (classification) of the individual trees.

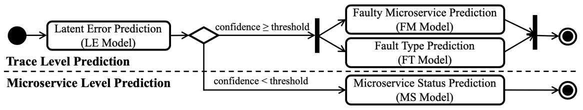


Figure 5: MEPFL Prediction Process

- K-Nearest Neighbors (KNN): a non-parametric classification technique that determines the class of an object by a plurality vote of its neighbors.
- Multi-Layer Perceptron (MLP): a feed-forward artificial neural network model that maps a set of input data onto a set of appropriate outputs.

To optimize these classification models, we use the grid search technique to search the hyper-parameter (e.g., "hidden_layer_sizes" for MLP) space for the best cross-validation score. The technique exhaustively generates candidates from a grid of parameter values specified with some initial parameters, and then scans all the possible combinations to find the best cross-validation score.

4.5 Feature Extraction and Prediction

The online prediction phase includes two steps. The feature extraction step extracts microservice- and trace-level features from trace logs. The prediction step uses the prediction models to predict latent errors, fault locations, and types.

Figure 5 shows the prediction process, which combines the trace- and microservice-level prediction models. Given a trace instance captured online, MEPFL first uses the LE model to predict whether the trace instance has latent errors and determines whether trace-level prediction has the required confidence level (obtained along with the prediction from the LE model). If the confidence is not lower than a predefined threshold, MEPFL uses the FM model and the FT model to predict the faulty microservices and the fault types, respectively. Otherwise, MEPFL uses the MS model to predict latent errors, fault locations, and types. Note that the MS model predicts the fault status of each microservice involved in the given trace instance. If any microservice is predicted to be faulty, MEPFL regards the trace instance as erroneous and all the faulty microservices together with their fault types as the results of fault location and type prediction.

5 IMPLEMENTATION

Our implementation includes four major components: the fault injector, execution controller, log processor, and predictor.

The fault injector currently supports the fault injection of Java microservices. It uses JavaParser 3.8 [27] to parse and manipulate the source code of microservices to implement fault injection. Other parsers can be introduced in the future to support fault injection of microservices implemented in other languages.

The execution controller integrates TestNG 6.8 [51] to implement automated scheduling and execution of test cases. At the system infrastructure level, the execution controller integrates Kubernetes 1.10 and Istio 1.0 to implement automatic microservice application deployment, configuration, and runtime manipulation in the offline training. It uses Kubernetes REST APIs to dynamically deploy microservice applications and manage microservice instances and their environmental configurations. It customizes the Istio implementation to control the execution/returning sequences

Table 3: The Pearson Correlation Coefficient of Microservice Level Features and Latent Errors

Trace Types \ Features		Configuration			Resource				Instance									Interaction			
		ML	CL	VS	MC	CC	NMC	NCC	IN	IIN	SVA	CA	SA	TN	LT	NIN	NMN	ET	RSC	AIT	CEO
Trace Type 1	Instance	0.02	0.12	0.22	0.36	0.12	0.19	0.09	0.83	0.63	0.67	0.78	0.22	0.39	0.05	0.19	0.03	0.58	0.05	0.17	0.23
	Config	0.61	0.20	0.11	0.77	0.36	0.46	0.42	0.12	0.03	0.11	0.21	0.25	0.25	0.12	0.15	0.05	0.65	0.43	0.34	0.16
	Interaction	0.10	0.22	0.03	0.01	0.18	0.06	0.13	0.07	0.14	0.45	0.12	0.03	0.54	0.16	0.09	0.11	0.70	0.47	0.62	0.79
Trace Type 2	Instance	0.04	0.09	0.27	0.23	0.05	0.23	0.12	0.79	0.46	0.73	0.66	0.06	0.62	0.15	0.12	0.11	0.54	0.46	0.16	0.14
	Config	0.23	0.67	0.20	0.33	0.76	0.48	0.25	0.03	0.23	0.22	0.12	0.03	0.34	0.14	0.15	0.04	0.32	0.66	0.45	0.17
	Interaction	0.22	0.18	0.13	0.21	0.13	0.02	0.01	0.16	0.11	0.36	0.46	0.18	0.59	0.07	0.63	0.03	0.47	0.21	0.66	0.68
Trace Type 3	Instance	0.03	0.17	0.18	0.15	0.11	0.14	0.02	0.87	0.90	0.56	0.78	0.23	0.17	0.17	0.42	0.14	0.46	0.31	0.10	0.19
	Config	0.70	0.13	0.13	0.83	0.46	0.36	0.12	0.02	0.22	0.19	0.09	0.12	0.20	0.02	0.14	0.09	0.33	0.45	0.29	0.21
	Interaction	0.01	0.09	0.09	0.15	0.03	0.07	0.10	0.04	0.13	0.23	0.33	0.16	0.13	0.11	0.23	0.03	0.45	0.29	0.79	0.72
Trace Type 4	Instance	0.10	0.10	0.15	0.22	0.04	0.05	0.10	0.69	0.36	0.77	0.66	0.11	0.33	0.01	0.17	0.08	0.28	0.60	0.11	0.23
	Config	0.39	0.74	0.19	0.67	0.71	0.49	0.18	0.06	0.33	0.10	0.12	0.24	0.01	0.11	0.23	0.03	0.39	0.17	0.15	0.28
	Interaction	0.14	0.21	0.21	0.17	0.11	0.03	0.14	0.11	0.12	0.33	0.43	0.13	0.32	0.09	0.12	0.04	0.50	0.25	0.73	0.47
Trace Type 5	Instance	0.13	0.07	0.07	0.20	0.12	0.11	0.09	0.70	0.87	0.60	0.56	0.24	0.35	0.03	0.33	0.05	0.21	0.19	0.08	0.12
	Config	0.60	0.16	0.22	0.71	0.66	0.63	0.24	0.03	0.21	0.14	0.23	0.23	0.09	0.05	0.08	0.15	0.46	0.30	0.47	0.15
	Interaction	0.05	0.09	0.03	0.36	0.16	0.15	0.08	0.09	0.26	0.13	0.22	0.12	0.41	0.16	0.06	0.09	0.36	0.61	0.54	0.63

of synchronous microservice invocations based on its sidecar. Istio sidecar is a kind of intelligent proxies that mediate and control the network communication between microservices [25].

The log processor uses Kubernetes and Istio to capture trace logs. To capture information about variable accesses required by some features such as SVA and CA, we use JDK to capture the runtime JVM thread dump and heap dump, and analyze the call stack and variable values in the dumps. The log processor includes a pipeline to collect and process trace logs. The pipeline runs a RESTful service based on Spring Boot 1.5.9 [50] to collect distributed log data produced by Istio and uses Kafka 2.0.0 [28] to stream the log data. It then uses Spark 2.3.2 [49] to process the data to produce training samples and uses HDFS (Hadoop Distributed File System) 2.7 [22] to store the processed data.

We implement the predictor based on scikit-learn [47], a machine learning library for Python. The threshold of the confidence score for adopting the trace-level prediction is 70%.

6 EVALUATION

To evaluate the effectiveness of MEPFL, we conduct a series of experimental studies to answer the following research questions.

- **RQ1**: How accurate is MEPFL in predicting latent errors, faulty microservices, and fault types of microservice applications?
- **RQ2**: How does the training coverage of trace types influence the prediction accuracy?
- **RQ3**: How effective is MEPFL when it is used to predict latent errors caused by real fault cases?

These studies are based on two open-source microservice applications: Sock Shop [48] and TrainTicket [36]. Sock Shop is a small-scale benchmark application that is widely used in microservice research [13, 31, 43]. It has 8 microservices and 10 trace types. TrainTicket is a medium-scale benchmark application that has been recently released, having 40 microservices and 86 trace types. It replicates a variety of faults based on a study of industrial fault cases in microservice applications. All the studies are conducted on a private cloud with 20 virtual machines.

6.1 Prediction Accuracy (RQ1)

To answer RQ1, we evaluate the accuracy of MEPFL using a corpus of trace logs with automatically injected faults. For Sock Shop, we inject 32 faults (including 10 multi-instance faults, 10 configuration faults, and 12 asynchronous interaction faults) into 10 trace types. We obtain 243,606 trace instances by test-driven automatic execution, 32,643 (13.4%) of which have latent errors. For TrainTicket, we inject 142 faults (including 46 multi-instance faults, 52 configuration faults, and 44 asynchronous interaction faults) into 86 trace types. We obtain 1,214,637 trace instances, 211,347 (17.4%) of which have latent errors. The fault injection process is automated. But when the injected faults cause compilation failure or the tests fail in unexpected ways, developers are required to manually analyze/edit code. Among the 174 faults injected into the two applications, 71.84% are injected automatically without human efforts; 28.16% involve human efforts.

For each application, we randomly divide the trace instances equally into 5 subsets and perform 5-fold cross-validation to evaluate the accuracy. In every cross-validation, we use 4 subsets as training data and the remaining subset as testing data. For the prediction of latent errors, we evaluate the recall, precision, and F1-measure (the harmonic mean of precision and recall) of identifying trace instances that have latent errors. As the number of successful trace instances is much larger than that of erroneous ones, we also evaluate the false positive rate (FPR), i.e., the rate of error-free traces that are predicted to be erroneous among all the successful traces. For the prediction of faulty microservices, we evaluate the Top-k (with k being 1, 3, 5) accuracy of identifying a faulty microservice from an erroneous trace instance, i.e., the probability that the faulty microservice is included in the Top-k prediction results. For the prediction of fault types, we evaluate the recall, precision, and F1-measure of the predicted fault types of erroneous trace instances.

To our best knowledge, there are no existing approaches of trace-log-based error prediction or fault localization for microservice applications. Therefore, we compare MEPFL with a state-of-the-art approach [40] for trace-based failure diagnosis in distributed systems. The approach uses fault injection to populate the database of failures for a target distributed system and locate the root causes of reported failures by matching against the failures in the database. In this study, we use the trace instances in the training data as the failure database for the approach to identify the faulty microservices and fault types. Note that this state-of-the-art approach cannot identify latent errors, as it assumes that failures can be reported from the production environment. For MEPFL, we evaluate three versions, each with a different machine learning technique, i.e., MEPFL-RF, MEPFL-KNN, and MEPFL-MLP.

Table 4 shows the accuracy of intra-application error prediction and fault localization. It can be seen that for both applications MEPFL achieves very high accuracy in the prediction of latent errors, faulty microservices, and fault types. In particular, MEPFL-MLP performs the best. MEPFL-MLP predicts latent errors with a recall of 0.982 (0.896), a precision of 0.998 (0.995), and a false positive

Table 4: Accuracy of Intra-Application Error Prediction and Fault Localization

Methods	Sock Shop										TrainTicket									
	Latent Error				Faulty Microservice			Fault Type			Latent Error				Faulty Microservice			Fault Type		
	Recall	Precision	F1	FPR	Top1	Top3	Top5	Recall	Precision	F1	Recall	Precision	F1	FPR	Top1	Top3	Top5	Recall	Precision	F1
MEPFL-RF	0.949	0.997	0.973	0.015	0.864	0.943	1.000	0.926	0.863	0.893	0.801	0.979	0.881	0.041	0.924	0.934	0.938	0.862	0.986	0.921
MEPFL-KNN	0.961	0.997	0.978	0.013	0.891	0.965	1.000	0.967	0.925	0.946	0.865	0.993	0.924	0.030	0.923	0.938	0.940	0.892	0.978	0.933
MEPFL-MLP	0.982	0.998	0.990	0.009	0.933	0.972	1.000	0.952	0.983	0.967	0.896	0.995	0.943	0.014	0.937	0.939	0.939	0.930	0.986	0.957
Approach in [40]	N/A	N/A	N/A	N/A	0.340	0.748	1.000	0.618	0.375	0.467	N/A	N/A	N/A	N/A	0.117	0.189	0.263	0.614	0.794	0.692

rate 0.009 (0.014), faulty microservices with a Top-1 accuracy of 0.933 (0.937), and fault types with a recall of 0.952 (0.930) and a precision of 0.983 (0.986) for Sock Shop (TrainTicket). The overall prediction accuracy of Sock Shop is slightly higher than that of TrainTicket. This result may be explained by the fact that Sock Shop has much fewer microservices and trace types. For both applications, MEPFL substantially outperforms the state-of-the-art approach [40] in predicting faulty microservices and fault types.

We also evaluate the accuracy of MEPFL for cross-application latent error prediction and fault localization. In this evaluation, we use the trace instances of an application (Sock Shop or TrainTicket) as training data and the trace instances of the other application as testing data. Note that the state-of-the-art approach [40] cannot be used for cross-application prediction, as it relies on the trace instances of the same application for matching. For MEPFL, only the microservice-level prediction works for this prediction.

Table 5 shows the accuracy of cross-application error prediction and fault localization. It can be seen that the accuracy is much lower than that of intra-application prediction. For example, for MEPFL-MLP, the recall/precision of latent error prediction drop from 0.982/0.998 and 0.896/0.995 to 0.451/0.562 and 0.461/0.418. Using TrainTicket data to predict Sock Shop has a higher accuracy than predicting the other way around. This result may be explained by the fact that TrainTicket has much more microservices and trace types than Sock Shop.

6.2 Influence of Trace Type Coverage (RQ2)

Fault injection and trace collection cost time and effort. Therefore, a practical problem is how to decide whether enough training data have been collected, i.e., whether the coverage of trace types in the training data substantially affects the prediction results.

In this evaluation, we use the best machine learning technique for MEPFL (i.e., MEPFL-MLP) and TrainTicket as the target application as it is much larger than Sock Shop. We randomly divide the trace types of TrainTicket into 10 subsets and use a part of the trace types for training and the others for testing. We start with one subset for training and gradually increase the number of subsets for training. For each setting, we calculate the accuracy of latent error prediction and fault localization along with the rate that trace-level prediction is adopted (T-Rate), and report the training time (TT) and prediction time (PT).

Table 6 shows the results of the evaluation. The general trend is that the prediction accuracy substantially increases with the increase of trace type coverage. For example, when 40% of the trace types are covered by the training data, the recall and precision of latent error prediction, the top-1 accuracy of faulty microservice prediction, and the recall and precision of fault type prediction are all higher than 80%. With the increase of trace type coverage, the overall prediction more and more relies on the trace-level prediction. When the trace type coverage reaches 30%, the trace-level prediction dominates the overall prediction. When the coverage reaches 90%, 81.3% of the prediction is made by the trace-level prediction. A possible explanation for the trend is that with the increase of trace type coverage, it is more likely to find similar trace instances in the training data. With the increase of trace type coverage, the training time linearly increases from 8m37s to 35m35s, while the prediction time keeps stable.

6.3 Effectiveness for Real Fault Cases (RQ3)

The purpose of this evaluation is to assess the effectiveness of MEPFL for real-world microservice faults. To this end, we evaluate the accuracy of MEPFL for the fault cases [10] provided by the TrainTicket benchmark. These fault cases are replicated from industrial fault cases by transferring the fault mechanisms from the original applications to the benchmark application. From 22 fault cases, we choose the 10 fault cases that belong to the three fault types for the evaluation.

We use the trace instances produced with injected faults to train the prediction models using MEPFL-MLP, and apply the models on the 10 fault cases. For each fault case, we apply the patch into the application to produce a faulty version and then deploy the version. Five student volunteers play the role of users and manually execute the scenarios that may involve the target microservice. For each execution, we manually check the system logs and execution results to confirm whether it involves a latent error and whether the error is caused by the fault case.

We collect all the produced trace instances and apply the prediction models on each trace instance. The evaluation results are shown in Table 7. For each fault case, we provide the fault type (Multi-Instance, Configuration, or Asynchronous Interaction), the numbers of trace instances (#TR) and erroneous ones (#ET), and the accuracy metrics. It can be seen that MEPFL accurately predicts the latent errors caused by these fault cases with a recall of 0.647-0.983 (0.821 on average), a precision of 0.586-0.984 (0.891 on average), and a false positive rate of 0.016-0.042 (0.030 on average), the faulty microservices with a Top-1 accuracy of 0.291-1.000 (0.788 on average), and the fault type with a recall of 0.676-0.975 (0.828 on average) and a precision of 0.648-0.977 (0.820 on average).

It can be seen that the prediction accuracy for configuration faults (F3, F4, F5, F7) is lower than the accuracy for multi-instance faults (F8, F11, F12) and asynchronous interaction faults (F1, F2, F13). This result may be due to that execution traces are less sensitive to latent errors caused by configuration faults. Multi-instance faults and asynchronous interaction faults have direct influences on execution traces, e.g., involved microservice instances, interaction sequences, or accesses of shared variables and cache. In contrast, configuration faults are usually related to environmental configurations and resource consumption, and have no direct influences on execution traces.

6.4 Threats to Validity

There are three major threats to the internal validity of the studies. The first one lies in the correctness of fault injection and trace

Table 5: Accuracy of Cross-Application Error Prediction and Fault Localization

Methods	TrainTicket Model Predicting Sock Shop										Sock Shop Model Predicting TrainTicket									
	LE				FM			FT			LE				FM			FT		
	Recall	Precision	F1	FPR	Top1	Top3	Top5	Recall	Precision	F1	Recall	Precision	F1	FPR	Top1	Top3	Top5	Recall	Precision	F1
MEPFL-RF	0.441	0.510	0.473	0.141	0.316	0.771	1.000	0.394	0.623	0.483	0.410	0.419	0.414	0.189	0.304	0.523	0.662	0.346	0.456	0.394
MEPFL-KNN	0.461	0.530	0.493	0.136	0.326	0.856	1.000	0.400	0.628	0.488	0.415	0.463	0.438	0.160	0.325	0.635	0.737	0.358	0.463	0.404
MEPFL-MLP	0.451	0.562	0.501	0.113	0.335	0.872	1.000	0.415	0.627	0.499	0.461	0.418	0.471	0.169	0.332	0.654	0.799	0.365	0.466	0.409

Table 6: Influence of Trace Type Coverage

Coverage	TT	PT	T-Rate	Latent Error				Faulty Microservice			Fault Type		
				Recall	Precision	F1	FPR	Top1	Top3	Top5	Recall	Precision	F1
10%	8min37s	12s	0.329	0.771	0.807	0.789	0.084	0.284	0.626	0.822	0.539	0.665	0.595
20%	11min58s	12s	0.403	0.795	0.846	0.820	0.061	0.299	0.627	0.827	0.676	0.890	0.768
30%	14min15s	12s	0.591	0.824	0.916	0.868	0.062	0.688	0.845	0.931	0.810	0.921	0.862
40%	16min24s	12s	0.655	0.829	0.945	0.884	0.059	0.813	0.905	0.935	0.803	0.960	0.874
50%	19min58s	12s	0.662	0.884	0.951	0.897	0.054	0.873	0.932	0.938	0.856	0.949	0.901
60%	23min39s	12s	0.766	0.858	0.950	0.902	0.044	0.920	0.941	0.944	0.844	0.937	0.888
70%	27min26s	12s	0.797	0.848	0.954	0.898	0.036	0.926	0.943	0.944	0.818	0.976	0.890
80%	30min42s	12s	0.830	0.857	0.958	0.905	0.032	0.907	0.939	0.943	0.851	0.995	0.917
90%	35min35s	12s	0.813	0.879	0.950	0.913	0.030	0.926	0.937	0.944	0.860	0.974	0.914

Table 7: Effectiveness for Real-World Fault Cases

Fault Cases	Type	#TR	#ET	LE				FM			FT		
				Recall	Precision	F1	FPR	Top1	Top3	Top5	Recall	Precision	F1
F1	AI	306	61	0.983	0.803	0.884	0.033	0.951	1.000	1.000	0.819	0.655	0.728
F2	AI	322	65	0.738	0.984	0.843	0.031	0.907	1.000	1.000	0.815	0.846	0.830
F3	C	310	48	0.667	0.979	0.793	0.042	0.291	1.000	1.000	0.750	0.771	0.760
F4	C	312	37	0.783	0.918	0.845	0.026	0.486	0.513	1.000	0.918	0.972	0.944
F5	C	328	89	0.685	0.977	0.805	0.034	0.977	0.977	1.000	0.943	0.977	0.960
F7	C	392	34	0.647	0.882	0.746	0.028	0.558	1.000	1.000	0.676	0.823	0.742
F8	MI	343	75	0.960	0.586	0.728	0.041	1.000	1.000	1.000	0.840	0.826	0.833
F11	MI	328	40	0.975	0.875	0.922	0.024	0.875	0.975	0.975	0.975	0.975	0.975
F12	MI	369	37	0.837	0.918	0.876	0.027	0.837	1.000	1.000	0.702	0.648	0.674
F13	AI	322	64	0.937	0.984	0.960	0.016	1.000	1.000	1.000	0.843	0.703	0.767
Average	-	333	55	0.821	0.891	0.854	0.030	0.788	0.947	0.998	0.828	0.820	0.824

labeling. The fault injection implemented by automatic code transformation may fail to introduce a fault as expected and the error status of a trace instance may be incorrectly labeled. The second one lies in unknown problems within the two benchmark applications. These problems may cause unexpected errors of the applications, thus influencing the error status of trace instances. The third one lies in the uncertainties of automatic deployment and runtime management.

There are two major threats to the external validity of the studies. The first one lies in the limitation of two target applications used in the studies. Sock Shop and TrainTicket are much smaller than complex industrial microservice applications. The second one lies in the limitation of the fault cases used in the studies. The TrainTicket fault cases may be simpler than industrial faults and represent only a limited part of different types of faults. Thus it is not clear whether the approach can be effectively applied for much larger industrial applications and more complex fault cases.

7 RELATED WORK

Traditional fault localization approaches include slice-based fault localization (e.g., [3, 4, 6, 7, 55]), spectrum-based fault localization (e.g., [2, 18, 21, 23, 29, 38, 45], fault localization based on analyzing program states (e.g., [58–60]), fault localization based on data mining (e.g., [15, 16, 39]), and model-based fault localization (e.g. [1, 9, 19, 35]). In recent years, machine learning techniques have been applied for fault localization in different systems [12, 14, 33, 54]. These approaches focus on traditional programs and thus are not applicable to microservices. Indeed these approaches can be potentially applied to application faults in microservice applications, whereas our approach focuses on microservice-specific faults.

Recently, multiple approaches on fault localization for distributed systems or cloud native systems have been reported. Whittaker *et al.* [53] presented an approach that enables the developers to reason about the causes of events in distributed systems. Leonardo *et al.* [34] proposed a lightweight approach of fault localization for cloud systems. Wang *et al.* [52] presented an online approach of incremental clustering for fault localization in web applications. Pham *et al.* [40] proposed an approach to automating failure diagnosis in distributed systems by combining fault injection and data analytics. Compared to our work, the preceding previous approaches are designed for traditional distributed systems or cloud systems, lacking consideration of complex environmental factors that are essential in microservice applications, e.g., auto-scaling of the instances of each service, tremendous asynchronous interactions.

There are previous approaches on fault analysis for cloud systems by considering the high complexity and dynamism of cloud computing environments. Lin *et al.* [32] proposed an approach for predicting the failure proneness of a node in a cloud service system based on historical data. Pitakrat *et al.* [41] proposed an approach named Hora, which combines component failure predictors with architectural knowledge, and predicts failures caused by three representative types of faults: memory leak, system overload, and sudden node crash. Ahmed *et al.* [5] proposed a machine learning approach based on detecting metric correlation stability violations (CSV) for automated localization of performance faults for datacenter services running under dynamic load conditions. Dean *et al.* [17] presented PerfCompass, an online debugging approach for performance anomaly faults. PerfCompass can quantify whether a production-run performance anomaly has a global impact or local impact. The preceding previous approaches share some similar ideas as our approach. However, these approaches do not support latent error prediction and they do not focus on microservices. Although Hora can be applied for microservice applications, it is used to predict violations of expected QoS levels caused by system level faults (e.g., memory leak, system overload, and sudden node crash). In contrast, our approach is tailored to microservices and predicts latent errors caused by three types of microservice specific faults in the implementation and configuration of microservices.

8 CONCLUSION

In this paper, we have proposed MEPFL, an approach for latent error prediction and fault localization of microservice applications by learning from system trace logs. It supports three types of microservice application faults that are specifically relevant to microservice interactions and runtime environments, i.e., multi-instance faults, configuration faults, and asynchronous interaction faults. Based on a set of features defined on system trace logs, MEPFL trains prediction models at both the trace level and microservice level using the system trace logs collected from automatic executions of the target application and its faulty versions produced by fault injection. Our experimental studies have demonstrated the effectiveness of MEPFL with two open-source microservice benchmarks and real-world fault cases.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China under Grant No. 2018YFB1004803, and NSF under grants no. CNS-1513939, CNS-1564274, CCF-1816615, and a grant from Futurewei.

REFERENCES

[1] Rui Abreu and Arjan J. C. van Gemund. 2009. A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis. In *Proc. 8th Symposium on Abstraction, Reformulation, and Approximation.*

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proc. 12th IEEE Pacific Rim International Symposium on Dependable Computing*. 39–46.

[3] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. 1993. Debugging with Dynamic Slicing and Backtracking. *Softw., Pract. Exper.* 23, 6 (1993), 589–616.

[4] Hiralal Agrawal and Joseph Robert Horgan. 1990. Dynamic Program Slicing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation.* 246–256.

[5] Jawwad Ahmed, Andreas Johnsson, Farnaz Moradi, Rafael Pasquini, Christofer Flinta, and Rolf Stadler. 2017. Online Approach to Performance Fault Localization for Cloud and Datacenter Services. In *Proc. IFIP/IEEE Symposium on Integrated Network and Service Management.* 873–874.

[6] Zuhoor A. Al-Khanjari, Martin R. Woodward, Haider Ali Ramadhan, and Narayana Swamy Kutti. 2005. The Efficiency of Critical Slicing in Fault Localization. *Software Quality Journal* 13, 2 (2005), 129–153.

[7] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. 2011. Fault-Localization Using Dynamic Slicing and Change Impact Analysis. In *Proc. 26th IEEE/ACM International Conference on Automated Software Engineering.* 520–523.

[8] Hidenao Abe andShusaku Tsumoto. 2008. Analyzing Behavior of Objective Rule Evaluation Indices Based on a Correlation Coefficient. In *Proc. 12th International Conference on Knowledge-Based Intelligent Information and Engineering Systems.* 758–765.

[9] George K. Baah, Andy Podgurski, and Mary Jean Harrold. 2010. The Probabilistic Program Dependence Graph and Its Application to Fault Diagnosis. *IEEE Trans. Software Eng.* 36, 4 (2010), 528–545.

[10] TrainTicket Benchmark. 2018. Fault Cases. Retrieved December 25, 2018 from https://github.com/FudanSELab/train-ticket-fault-replicate

[11] Dhruba Borthakur. 2019. HDFS Architecture Guide. Retrieved January 08, 2019 from https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[12] Lionel C. Briand, Yvan Labiche, and Xuetao Liu. 2007. Using Machine Learning to Support Debugging with Tarantula. In *Proc. 18th IEEE International Symposium on Software Reliability Engineering.* 137–146.

[13] Antonio Brogi, Luca Rinaldi, and Jacopo Soldani. 2018. TosKer: A Synergy between TOSCA and Docker for Orchestrating Multicomponent Applications. *Softw., Pract. Exper.* 48, 11 (2018), 2061–2079.

[14] Yuriy Brun and Michael D. Ernst. 2004. Finding Latent Code Errors via Machine Learning over Program Executions. In *Proc. 26th International Conference on Software Engineering.* 480–490.

[15] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. 2008. Formal Concept Analysis Enhances Fault Localization in Software. In *Proc. 6th International Conference on Formal Concept Analysis.* 273–288.

[16] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. 2011. Multiple Fault Localization with Data Mining. In *Proc. 23rd International Conference on Software Engineering & Knowledge Engineering.* 238–243.

[17] Daniel Joseph Dean, Hiep Nguyen, Peipei Wang, Xiaohui Gu, Anca Sailer, and Andrzej Kochut. 2016. PerfCompass: Online Performance Anomaly Fault Localization and Inference in Infrastructure-as-a-Service Clouds. *IEEE Trans. Parallel Distrib. Syst.* 27, 6 (2016), 1742–1755.

[18] Vidroha Debroy, W. Eric Wong, Xiaofeng Xu, and Byoungju Choi. 2010. A Grouping-Based Strategy to Improve the Effectiveness of Fault Localization Techniques. In *Proc. 10th International Conference on Quality Software.* 13–22.

[19] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. 1997. Failure and Fault Analysis for Software Debugging. In *Proc. 21st International Computer Software and Applications Conference.* 515–521.

[20] Andrei Furda, Colin J. Fidge, Olaf Zimmermann, Wayne Kelly, and Alistair Barros. 2018. Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency. *IEEE Software* 35, 3 (2018), 63–72.

[21] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. 2006. Error Explanation with Distance Metrics. *International Journal on Software Tools for Technology Transfer* 8, 3 (2006), 229–247.

[22] Hadoop.Com. 2018. Hadoop. Retrieved February 21, 2018 from http://hadoop.apache.org/

[23] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. 2000. An Empirical Investigation of the Relationship Between Spectra Differences and Regression Faults. *Softw. Test., Verif. Reliab.* 10, 3 (2000), 171–194.

[24] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. 2016. Experience Report: System Log Analysis for Anomaly Detection. In *Proc. 27th IEEE International Symposium on Software Reliability Engineering.* 207–218.

[25] Istio. 2018. Istio. Retrieved February 21, 2018 from https://istio.io/

[26] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35, 3 (2018), 24–35.

[27] Javaparser.Com. 2018. Javaparser. Retrieved February 21, 2018 from https://javaparser.org/

[28] Kafka.Com. 2018. Kafka. Retrieved February 21, 2018 from http://kafka.apache.org/

[29] Bogdan Korel. 1988. PELAS - Program Error-Locating Assistant System. *IEEE Trans. Software Eng.* 14, 9 (1988), 1253–1260.

[30] Kubernetes.Com. 2018. Kubernetes. Retrieved February 21, 2018 from https://kubernetes.io/

[31] JinJin Lin, Pengfei Chen, and Zibin Zheng. 2018. Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-service Environments. In *Proc. 16th International Conference on Service-Oriented Computing.* 3–20.

[32] Qingwei Lin, Ken Hsieh, Yingnong Dang, Hongyu Zhang, Kaixin Sui, Yong Xu, Jian-Guang Lou, Chenggang Li, Youjiang Wu, Randolph Yao, Murali Chintalapati, and Dongmei Zhang. 2018. Predicting Node Failure in Cloud Service Systems. In *Proc. ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 480–490.

[33] Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jin Song Dong. 2018. Break the Dead End of Dynamic Slicing: Localizing Data and Control Omission Bug. In *Proc. 33rd ACM/IEEE International Conference on Automated Software Engineering.* 509–519.

[34] Leonardo Mariani, Cristina Monni, Mauro Pezzè, Oliviero Riganelli, and Rui Xin. 2018. Localizing Faults in Cloud Systems. In *Proc. 11th IEEE International Conference on Software Testing, Verification and Validation.* 262–273.

[35] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. 2000. Modeling Java Programs for Diagnosis. In *Proc. 14th European Conference on Artificial Intelligence.* 171–175.

[36] Microservice.System.Benchmark. 2018. TrainTicket. Retrieved December 25, 2018 from https://github.com/FudanSELab/train-ticket/

[37] William Morgan. 2017. What's a Service Mesh? And Why Do I Need One? Retrieved February 21, 2018 from https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/

[38] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A Model for Spectra-Based Software Diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3 (2011), 11:1–11:32.

[39] Syeda Nessa, Muhammad Abedin, W. Eric Wong, Latifur Khan, and Yu Qi. 2008. Software Fault Localization Using N-gram Analysis. In *Proc. 3rd International Conference on Wireless Algorithms, Systems, and Applications.* 548–559.

[40] Cuong Pham, Long Wang, Byung-Chul Tak, Salman Baset, Chunqiang Tang, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2017. Failure Diagnosis for Distributed Systems Using Targeted Fault Injection. *IEEE Trans. Parallel Distrib. Syst.* 28, 2 (2017), 503–516.

[41] Teerat Pitakrat, Dusan Okanovic, André van Hoorn, and Lars Grunske. 2018. Hora: Architecture-Aware Online Failure Prediction. *Journal of Systems and Software* 137 (2018), 669–685.

[42] The CentOS Project. 2019. CentOS Linux. Retrieved January 08, 2019 from https://www.centos.org/about/

[43] Peter-Christian Quint and Nane Kratzke. 2018. Towards a Lightweight Multi-Cloud DSL for Elastic and Transferable Cloud-native Applications. In *Proc. 8th International Conference on Cloud Computing and Services Science.* 400–408.

[44] Redis.Io. 2016. redis.io. Retrieved January 16, 2018 from https://redis.io/

[45] Manos Renieris and Steven P. Reiss. 2003. Fault Localization with Nearest Neighbor Queries. In *Proc. 18th IEEE International Conference on Automated Software Engineering.* 30–39.

[46] Replication.Package. 2019. Latent Error Prediction and Fault Localization for Microservice Applications. Retrieved June 30, 2019 from https://fudanselab.github.io/Research-ESEC-FSE2019-AIOPS

[47] ScikitLearn.Com. 2018. ScikitLearn. Retrieved February 21, 2018 from https://scikit-learn.org/

[48] Sockshop.Com. 2018. Sockshop. Retrieved March 8, 2018 from https://github.com/microservices-demo/microservices-demo

[49] Spark.Com. 2018. Spark. Retrieved February 21, 2018 from http://spark.apache.org/

[50] SpringBoot.Com. 2018. Spring Boot. Retrieved February 21, 2018 from http://projects.spring.io/spring-boot/

[51] Testng.Com. 2018. Testng. Retrieved February 21, 2018 from https://testng.org/doc/index.html

[52] Tao Wang, Wenbo Zhang, Chunyang Ye, Jun Wei, Hua Zhong, and Tao Huang. 2016. FD4C: Automatic Fault Diagnosis Framework for Web Applications in Cloud Computing. *IEEE Trans. Systems, Man, and Cybernetics: Systems* 46, 1 (2016), 61–75.

[53] Michael Whittaker, Cristina Teodoropol, Peter Alvaro, and Joseph M. Hellerstein. 2018. Debugging Distributed Systems with Why-Across-Time Provenance. In *Proc. ACM Symposium on Cloud Computing.* 333–346.

[54] W. Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani M. Thuraisingham. 2012. Effective Software Fault Localization Using an RBF Neural Network. *IEEE Trans. Reliability* 61, 1 (2012), 149–169.

[55] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016),

707–740.

[56] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing Logging Practices in Open-Source Software. In *Proc. 34th International Conference on Software Engineering*. 102–112.

[57] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving Software Diagnosability via Log Enhancement. *ACM Trans. Comput. Syst.* 30, 1 (2012), 4:1–4:28.

[58] Andreas Zeller. 2002. Isolating Cause-Effect Chains from Computer Programs. In *Proc. 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. 1–10.

[59] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.

[60] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating Faults through Automated Predicate Switching. In *Proc. 28th International Conference on Software Engineering*. 272–281.

[61] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2887384

[62] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2019. Delta Debugging Microservice Systems with Parallel Optimization. *IEEE Transactions on Services Computing* (2019). https://doi.org/10.1109/TSC.2019.2919823

[63] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Wenhai Li, Chao Ji, and Dan Ding. 2018. Delta Debugging Microservice Systems. In *Proc. 33rd ACM/IEEE International Conference on Automated Software Engineering*. 802–807.