2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC) | 978-1-6654-2463-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/COMPSAC51774.2021.00121

Trace-based Intelligent Fault Diagnosis for Microservices with Deep Learning

Hao Chen[1], Kegang Wei[1,3], An Li[1], Tao Wang[1,2], Wenbo Zhang[1,2]
1. Institute of Software Chinese Academy of Sciences
2. State Key Laboratory of Computer Science, Institute of Software Chinese Academy of Sciences
3. University of Chinese Academy of Sciences
Beijing 100190, China
{chenhao, weikegang, lian, wangtao, zhangwenbo}@otcaix.iscas.ac.cn

Abstract*—Due to the scalability, fault tolerance, and high availability, distributed microservice-based applications gradually replace traditional monolithic applications as one of the main forms of Internet applications. However, current fault diagnosis methods for distributed applications have drawbacks in coarse-grained fault location and inaccurate root-cause analysis. To address the above issues, this paper proposes a trace-based intelligent fault diagnosis approach for microservices with deep learning. First, we build a request weighted directed graph and a request string to characterize the behaviors of microservices with collected historical traces. Then, we build a normal trace dataset in normal status and a faulty dataset by injecting faults, and then calculate the expected intervals of microservices' response time and the call sequences. After that, we train the fault diagnosis model based on the deep neural network with the trace datasets to diagnose faulty microservices. Finally, we have deployed a typical open-source microservice-based application TrainTicket to validate our approach by injecting various typical faults. The results show that our approach can effectively characterize the behavior of microservices when processing requests and effectively detect faults. For fault detection, our approach achieves 91.5% accuracy in detecting faults, and has the accuracy of 85.2% in locating root causes.

***Keywords*—microservice, fault diagnosis, deep learning, distributed tracing**

I. Introduction

With the rapid changes in the users' requirements of Internet applications, the scale and complexity of services has grown exponentially, and thus the traditional monolithic software architecture can no longer meet the requirements of the complex and changeable design, development, and deployment of Internet applications nowadays. Consequently, the microservice architecture is proposed, which is constructed around high coupling business. Each microservice divided by business has a single responsibility, which can be independently developed, tested, and managed. The microservice architecture can well satisfy the requirements of Internet applications for high scalability and high fault tolerance. However, the huge scale, complex structure, and dynamic interaction of microservices brings difficulties to operation and maintenance. So accurately detecting faults and analyzing root causes is difficult. Thus, how to characterize the interaction behaviors, effectively detect the latent faults, and accurately locate the root causes of microservice-based applications has become an urgent issue.

Existing fault diagnosis methods for distributed software systems can be classified into three main categories that are analyzing metrics, mining log files, and tracing systems' execution. First, the methods based on metric analysis collect the metrics of software components, and dynamically set the metric thresholds. When the specific metrics exceed the thresholds, the abnormal alarms are raised and sent to operators [1]-[3]. Since this method does not require operators to understand the structure and interactions of the system, it can be easily implemented in practice. However, a microservice-based application always has large-scale independent microservices with various characteristics, operators cannot set suitable thresholds for the fluctuation ranges of metrics in different microservices. Second, the methods based on mining log files collect discrete log files, parse the information of metadata from them, build indexes to store a massive amount of scattered incidents and generated information during operation, and retrieve error reports to locate faults [4]-[7]. The methods can extract the critical information of faults from logs, but they cannot locate faults in real time owing to the large number of log files with various contents. Furthermore, microservices are often developed by various service providers with different logging rules, so the methods cannot find out some faults, when the logs are recorded with unexpected formats or information. Third, the methods based on tracing systems' execution collect the information of processing requests, and analyze the deviation of runtime traces from baseline ones to detect the anomalies [8]-[10]. However, since dynamic complex interactions between microservices cause a large scale of traces, comparing target traces and finding root causes are difficult in practice.

To address the above issues, this paper proposes a trace-based intelligent fault diagnosis approach for microservices with deep learning. First, we model the behaviors of processing requests in microservice applications with a request weighted directed graph (RWDG) and a request string (RSS), which are employed to characterize the response time and the call sequence of a microservice, respectively. Then, we collect the traces of processing requests with tracing tools to build a normal trace dataset in normal status, and then calculate the expected intervals of microservices' response time and the expected processing sequences. Meanwhile, we collect the traces of processing requests to build a faulty trace data set by injecting faults. Finally, we train the fault diagnosis model based on the deep neural network with the built normal and faulty trace data sets. The input vector of the model is converted from the RWDG and the RSS, and the output vector represents the faulty microservices. The contributions of this article are as follows:

•This paper models the behaviors of processing requests in microservice applications with a request weighted directed graph (RWDG) and a request string (RSS). The method can not only characterize the interactions between microservices, but also describe the features of processing requests in terms of response time and response sequence. The method can well present the status of complex and changeable microservice-based applications more flexibly and accurately.

978-1-6654-2463-9/21/$31.00 ©2021 IEEE
DOI 10.1109/COMPSAC51774.2021.00121

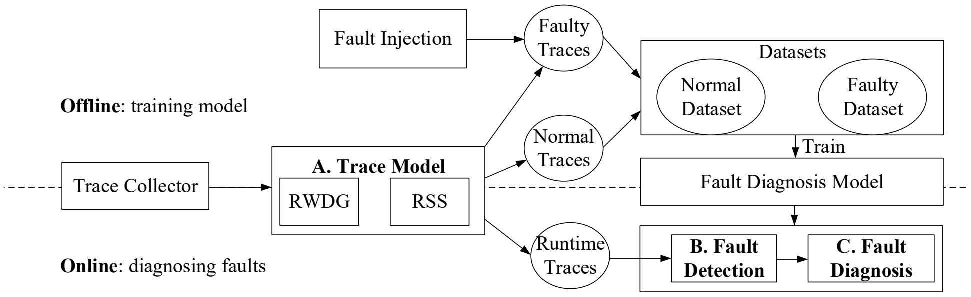

Fig. 1 Approach Overview

•This paper collects the traces of processing requests to build a faulty trace data set by injecting faults, and thus we can record the changes of interaction behaviors in faulty status. So when similar faults in the production environment occur again, we can determine the causes of the faults quickly and accurately with the recorded behaviors and performance of the injected faults.

•As for the rare faults in the system, the method proposed by this paper will compare them with known faults, and suggest possible similar faults through the analysis of known fault causes. The operation and maintenance personnel can use this as an aid to quickly find the real fault in the system.

II. Approach

As shown in Fig. 1., our approach employs the traces of a running application to model the procedure of processing requests with a request weighted directed graph and a request string, and then we diagnose the faults with a deep neural networks model. Faults could cause microservices to process requests that deviate from the normal processing pattern of learning. Such deviations usually manifest as prominent fluctuations in response times or differences in workflow processing of requests. By building trace models, we are able to extract the request processing flows recorded in the trace logs. Therefore, we focus on the faults that occur in a particular request as described by trace logs, and then we propose the fault detection and diagnosis methods in this section.

A. Trace Model

Tracing is a microservice monitoring method. trace logs record every call between microservices. We use the call data from Trace logs to build Trace models that can describe the response time and call sequence.

1) Collecting Traces

We use a distributed tracing tool Jaeger [25] abiding by the OpenTracing specification to collect trace logs. OpenTracing specification is a unified standard, which defines a language-neutral data model and a backend-independent API guidelines. We collect traces in normal status and faulty status to analyze different faults that cause different performance anomalies and workflow anomalies can have on the application's ability to process requests. Then, the normal tracing data is generated from the applications processing requests in a fault-free environment, and the faulty tracing data is generated from the applications injected with faults in processing requests. We use the fault injection technology of chaos engineering to analyze the possible changes in application after faults are injected. Chaos engineering is a type of engineering that investigates applications by triggering system failures [17]. However, faults may not be recorded in log files. We need to further design modeling methods to rebuild the trace model, so that we can uncover the connections between faults and microservice interactions.

2) Characterizing Response Time

In a distributed software architecture, the application processing requests usually invoke the corresponding multiple microservices. By collecting the call information between microservices, we can restore the request processing flow and find the latent faults. Then, we will introduce how to construct a RWDG model for requests, which focuses on response time and service dependencies.

We use the structure of RWDG to describe the invocation relationships and performance recorded by the microservice trace *T*. The structure is *D={V,E}*, where each vertex *S* corresponds to a microservice in the application, and all vertices *S* together form an array *V*; for any two microservices, such as microservice A and B, if microservice A generates a processing call to microservice B, then in the RWDG, the Directed edges $\overrightarrow{ab}$ are established between the vertices *a* and *b* corresponding to the call between microservice A and microservice B. In addition, since microservice A may invoke microservice B several times to process the request, its weight is expressed as the average of the time *t* consumed for the request to be accepted, and the adjacency matrix *E* records all directed edges of the directed graph with weights; depending on the content of the request and the functionality of the application, multiple calls of the same microservice may result in different invocation relationships, so the trace *T* of

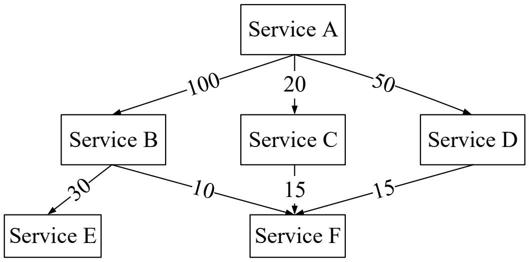

Fig. 2 Request example

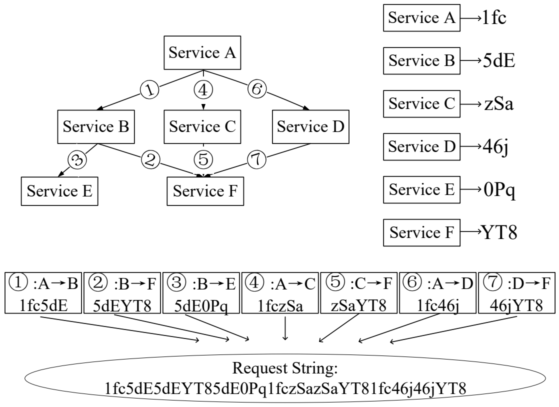

Fig. 3 Generation of a Request String

the request and the corresponding RWDG *D* may not be unique.

For each vertex *S* of the RWDGs, we describe the information related to the microservices' requests as:

Si = (ServiceName, spansNum, status, details)

where *ServiceName* is the service name of the microservice; *spansNum* indicates the number of times the microservice was called in this request; *status* indicates whether the microservice returned a fault response code; *details*={s_1, s_2, ... ,s_n} records the details of each call for the vertex S_i, s_i is represented as:

si = (startTime, duration, statusCode, callerName)

where *startTime* is the timestamp of the microservice being called; *duration* is the time consumed by the execution of the microservice; *statusCode* is the HTTP status code of the response, set to 0 if it does not exist; *callerName* is the name of the caller microservice.

In Fig. 2, three different microservices B, C, and D are called by microservice A. Microservice B continues to call microservices E and F to process requests, and microservice D calls microservice F. The same microservice names denote the same microservices, and there is a call-to-call relationship between the microservices. For the RWDG, the weights of directed edges between vertices indicate the time consumed by the service responses.

3) Characterizing Call Sequences

Microservice applications have the characteristic that each business module is independent of each other. Thus, microservice applications need to call each microservice in turn to process its business requests, then each microservice further refines and processes the requests, and finally returns the results. The changes in the order of microservice calls may affect the success of business processing or not. The documentation and analysis of the microservice call sequence will help us analyze microservice's workflow anomalies. Thus, we will introduce how to build RSSs for the traces, and the RSSs will focus on the order of microservice calls.

For a trace *T*, each span can be converted into a fixed-length string representing the operation by using a hash function. By stitching these strings together in order of call, we can get the RSS *C* representing the trace *T*. The RSS *C* is essentially a record of the order of the call. It reflects the dependencies and the call orders among microservices in an efficient way.

Fig. 3 shows an example of how the above instance is translated into a RSS. In this example, microservice A first calls microservice B to process the request, and microservice B continues to call microservices E and F in turn. Then, microservice A calls microservice C, which calls microservice F to process the request. Finally, microservice A calls microservice D, which calls microservice F again to process. The same microservice name represents the same microservice. We use a fixed hash function to map the names of the microservices to fixed-length strings that reflect the microservices. All strings are then stitched together in turn based on the chronological order in which the calls are generated (as in the order of the markers in the figure). The resulting strings reflect the order of microservice invocations in that trace.

B. Fault Detection

Based on whether faults are injected in the application, we classify RWDGs and RSSs into two categories: normal trace model and faulty trace models. In this subsection we will use these classification models to detect faults.

As we know, faults are often accompanied by changes in the response times of microservices and in the invocation relationships, which corresponds to changes of RWDGs' adjacency matrixes and changes of RSSs' character orders. Thus, we use the value ranges of elements in RWDGs' adjacency matrixes and the string edit distance of RSSs to characterize those faults. Accordingly, we design the methods of anomalies detection and faults location.

1) Performance Anomaly Detection

The RWDG models focuses on microservice response time, which is an important metric for us to detect microservice performance anomalies. We next describe the role of RWDGs in detecting microservice performance anomalies.

In RWDGs, the response times of the same type of requests between vertices are close to a normal distribution due to the randomness factors such as network latency. Thus, we can take an interval so that the response times that falls outside the interval can be considered as an exception. Therefore, we take all RWDGs generated by the normal trace log as a sample and calculate the bounds for the normal response interval of the requests:

$$t_{max} = \bar{t} + z_\alpha * \frac{\sigma}{\sqrt{n}}$$

$$t_{min} = \bar{t} - z_\alpha * \frac{\sigma}{\sqrt{n}}$$

where t_{max} is the upper bound of the normal response time; t_{min} is the lower bound of the normal response time; $\bar{t}$ is the average response time of the service request; σ is the standard deviation of the response time of the microservices; *n* is the number of sample responses; z_α is the alpha quantile of the normal distribution, and a 95% confidence probability is used in this paper. It is taken as 1.96 according to the reference [11].

For each request, we can get the corresponding RWDG. The following algorithmic ideas are proposed to handle the complex call information and detect whether the application has performance anomalies:

a) For the RWDG *Du* corresponding to the current request, retrieve the vertex S_1 with the earliest start time as the starting vertex, which indicates the first microservice called in the current request. Then, we use the hash function to transform vertex S_1's service name N_1 into a fixed-length string C_1.

b) Get the weight dt_1 of the directed edge between this vertex and the next vertex S_2.

c) Go to the next vertex S_2. according to the call relationships. Then, use same mapping function as in step (1) to transform vertex S_2's service name N_2 into a fixed-length string C_2.

d) Query the normal weight interval of the directed edge between C_1 and C_2. If dt_1 is within the normal interval, judge that the call from S_1 to S_2 is a normal call and take the vertex S_2 as the current vertex; otherwise, this call is an abnormal call.

e) Repeat steps (a), (b), (c) and (d) for each vertex *T* until every call of the current request is detected. If there is an abnormality, return the RWDG status as 1; otherwise, return 0.

2) *Workflow Anomaly Detection*

In the real industrial environment, requests may still be processed in normal response time interval even if potential workflow anomalies exist in the current microservice application. Therefore, we propose an algorithm using edit distances between RSSs to detect workflow anomalies.

For the call order, we transform the trace *T* into the corresponding RSS C. The RSS C records a call sequence of a processed request. After recording the large number of correctly processed request, we can assume that all possible invocation relationships known to occur during normal responses have been collected. Therefore, for each request to be detected, it is queried by transforming its call sequence into a RSS, and when no matching RSS *C* appears, it can be considered that there is a workflow anomaly exist in this request, which means that a fault has occurred.

For each request, we can get the corresponding RSS. The following algorithmic ideas are proposed to determine if the call sequence is abnormal and to calculate the minimum edit distance from the normal RSSs:

a) For the current request, retrieve the span s with no parent span as the first span in its Trace logs, which indicates the first microservice called in the current request. request was issued as the starting service. Then, we use the hash function to transform caller microservice S_1's service name N_1 into a fixed-length string C_1.

b) Get the microservice S_2 called by microservice S_1. Then, use same mapping function as in step (1) to transform service S_2's service name N_2 into a fixed-length string C_2.

c) Continue to find the next span based on its dependencies, taking the next span as s.

d) Repeat steps (a), (b), and (c) until all spans of the current request are detected. Calculate the minimum edit distance *d* with all normal RSSs and return *d*.

Based on the above algorithm ideas, we propose the following algorithm for calculating the minimum edit distance of RSSs:

Algorithm 1 RSSs' Minimum edit distance calculation algorithm

Input：*Tr* trace logs, C the set of all normal RSSs
Output：*d* minimum edit distance

```
 1: function Judgement(Du)
 2:   Cr,Ck,C1,C2 ← " "
 3:   s ← Tr.root
 4:   S1,S2 ← null
 5:   d ← Max_Int
 6:   while s.next is not null do
 7:     S1 ← s.GetCaller()
 8:     S2 ← s.GetCalled()
 9:     C1 ← Hash(S1.name)
10:     C2 ← Hash(S2.name)
11:     Cr ← Cr + C1 + C2
12:     s ← s.next
13:   end while
14:   for Ck in C do
15:     if EditDis(Ck,C) < d:
16:       d ← EditDis(Ck,C)
17:     end if
18:   end for
19:   return d
20: end function
```

Algorithm 1 can determine whether there is a sequence exception in the RSS by using all normal RSSs as the basis. The inputs are the trace log sequence *Tr* that has been sorted according to the request dependencies , and the set of all normal RSSs *C*. The algorithm gets the first microservice's service name from the first span in *Tr* and then transforms the name into a fixed-length string (line 3 to 10). Then, it splices the string according to the invocation relationship (line 11) until all spans are detected (line 6 to 13). After that, it calculates the minimum edit distance between the RSS generated by the current request and all normal RSSs (line 14 to 18) as the return value (line 19). If the value of d is 0, here is no anomalies in RSS; if d is greater than 0, there is a workflow anomaly exist in RSS. The main purpose of Algorithm 1 is to discover traces that may reflect the presence of potential workflow anomalies and to mark such RSSs.

We can quantitatively evaluate its anomaly degree e_w by using the ratio *r* of computing the minimum edit distance *d* of the RSS from all normal RSSs to the RSS length l_C. The magnitude of the anomaly degree e_w reflects the proportion of the overall call sequence accounted for by the suspicious call sequence in the request processing workflow. Anomaly degree e_w is the evaluation metric for call sequence anomalies in our approach.

C. *Fault Diagnosis*

In the previous two subsections, we got metrics to evaluate whether a RWDG is anomalous or not, and the degree of workflow anomalies based on the minimum RSS edit distance.

We next describe how to use the deep neural networks for fault localization.

First, we design the input vector X of the model as:

$$X=\left(E, status, e_c, S, l_C, e_w\right)$$

where E is the adjacency matrix of the RWDG, and each element t_{mn} represents the average response time of the call from a microservice numbered m to a microservice numbered n. Due to the constraints of the input vector, we split E in row vectors and combine sequentially. *Status* is the identifier of whether there is an exception in the corresponding RWDG and takes the value of 0 or 1; e_c is whether the request packets contain the http.error field, if there is http.error and the value is true, it is set to 1 and vice versa; S is the matrix of call times, each element s_{pq}'s value is the number of times the substring generated by the call initiated by the microservice numbered p to the microservice numbered q appears in the corresponding RSS. Due to the constraints of the input vector, we split S in row vectors and combine sequentially as:

$$S=\begin{pmatrix} s_{11} & \cdots & s_{1n} \\ \vdots & \ddots & \vdots \\ s_{n1} & \cdots & s_{nn} \end{pmatrix} \rightarrow (s_{11},\ldots,\ s_{1n}, s_{21},\ldots, s_{2n},\ldots, s_{n1},\ldots, s_{nn})$$

l_C is the length of the corresponding RSS; e_w is workflow anomaly degree, which takes values in the range [0,1]. We normalize the elements x_i contained in E, S, and l_C in the input vector X :

$$x_i' = \frac{x_i - \bar{x}}{x_{\max} - x_{\min}}$$

where x_i is the current element value; $\bar{x}$ is the average value of x_i; x_{max} is the maximum value of x_i; x_{min} is the minimum value of x_i. The elements within the normalized input vector X and their value ranges are as follow: E is the adjacency matrix of RWDG and each of E's elements t_{mn} takes values ranging from 0 to 1; e_c indicates whether the http.error response code in the request appears, which still takes the value of 0 or 1; *status* indicates whether the request has performance anomalies , which also still takes the value of 0 or 1; S is the matrix of call times and each of S's elements s_{pq} takes values ranging from 0 to 1; l_C is the length of the corresponding RSS, which takes values ranging from 0 to 1; e_w is workflow anomaly degree, which takes values ranging from 0 to 1.

Correspondingly, we design the output vector to represent the model of detecting the microservice affected by the potential faults with above input parameters. Each y in vector Y can take in the set of all microservices and the value of y means the microservice that is most likely to be affected by the current fault. The process of finding the vector Y is a multi-classification problem.

Then, we use the deep neural networks, or DNNs, to implement the classifier model. DNNs is a supervised learning model. By giving the inputs and the outputs, a nonlinear function classification approximation can be performed. DNNs can have multiple hidden layers between the input and output layers. With the backpropagation process, the DNNs can update these hidden layers. Thus, DNNs has self-learning and self-adaptive capabilities. We can calculate and find the computational faults of the output network by the feed-forward process. Then, we were able to optimize the weights of the neural network using gradient descent through a backpropagation process. After multiple adjustments, the DNNs model is effective to reflect the nonlinear mapping relationship between the microservices affected by the fault and the trace models. Thus, we use the DNNs model based on back propagation algorithm in our approach to locate faults. We use the Adam algorithm for the feedforward transfer process. The Adam algorithm can maintain a learning rate for each parameter by correcting the exponential moving averages of the gradients and the squares gradients. Therefore, the Adam algorithm is able to obtain results faster compared to other feedforward algorithms. Besides, we choose the tanh function as the activation function:

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The tanh function solves the non-zero mean problem faced by Sigmoid function and can reduce the gradient vanishing problems. In addition, in order to achieve the labeling of the actual value in the training set in supervised learning, we adopt the following method: when the impact of the injection fault in the microservice application is a large fluctuation in the response time of a microservice, all the requests involving the service are labeled as having an exception. The output values of fault requests are the affected microservices and the rest are "No fault". When the impact of the injection fault in the microservice application is a change in the processing order of microservice requests, all the requests containing the changed part are marked as an exception. The output values of fault requests are the affected microservices, and the rest are "No fault".

III. Evaluation

A. Experimental Environment

We use an open source microservice application TrainTicket [21] to start our experiments. In our experiments, we access it by simulating HTTP requests from normal users. TrainTicket is an application with complex interactions, the main function is to simulate a train ticket reservation system, and contains a total of 41 microservices. The microservices included in the application are developed in multiple programming languages. Their interaction modes include synchronous invocation, asynchronous invocation, message queue, etc., which are in line with the characteristics of microservice applications having a single responsibility, being independently deployable, scalable, testable, and interacting via messages [22]. To collect and process the trace logs of TrainTicket, we used Kubernetes [23] and istio [24] for container instance orchestration management and inter-microservices traffic monitoring, respectively, and used Jaeger, an open source distributed trace system based on the OpenTracing specification, to collect the raw trace logs, which are stored in the Elasticsearch [26] database. We have released the project of our work in the github. It contains two subprojects: Tracelogstotraining [30] and TTFI [31]. The main function of the Tracelogstotraining project is to transform the raw trace logs into the dataset required for the training model and to develop fault discovery rules. The TTFI project contains the dataset used in this experiment.

B. Steps

Zhou et al. listed 22 representative industrial fault cases [20] and classified the root causes of these 22 faults into four major categories: Monolithic faults, Multi-Instance faults, Configuration faults and Asynchronous Interaction faults, which were reproduced in the TrainTicket application by fault injection. Their study comprehensively analyzed the causes of faults and proposes of faults reflects the challenges that may be faced in the current production environment of microservice applications, which facilitates us to study the characteristics and behaviors of these faults.

On this basis, typical faults faced with microservice applications are selected for replication in this paper, and the experimental faults are numbered as injection faults F1~F8. The selected test method can be better collected and analyzed by microservice tracing. We studied the interaction rules of microservices within the TrainTicket application and designed an automatic script. By running the script and heavily simulating user requests, a large number of traces documenting the microservice interaction behavior can be collected, which describe the real performance of the application at the time of faults. We can accordingly restore the time consumed for each request to be invoked, which is the main metric used to detect performance anomalies. The collected tracing information is classified by using the injected faults as labels, which can be used as a useful reference when faults with unknown causes occur. In addition, in the real environment, microservices also face faults caused by physical resources and container anomalies, which are often not caused by microservices themselves, and we have selected typical cases for analysis respectively. The faults injected in the experiments are shown in Table 1.

As shown in the table above, the fault generated by experiment F3 causes the microservices ts-admin-travel-microservice, ts-preserve-other-microservice, and ts-travel-plan-microservice that calls the ts-travel2-microservice present anomalies. However, considering that the fault injected by our experiment is ts-travel2-microservice, we should assume that the fault itself is not caused by the ts-admin-travel-microservice, ts-preserve-other-microservice, ts-travel- plan-microservice service, but by the ts-travel2-microservice service, which has a packet loss exception in the network, so we mark ts-travel2-microservice as the service that generates the fault in experiment F3.

Since the applications in real production environments tend to generate a large number of historical trace logs and we can easily verify that the requests logged by the monitoring system are often handled correctly (since few systems handle more requests in a fault situation than in a normal operating environment). Therefore, in the experiments, we first ensured the normal operation of the application under test and used Katalon [28] to write automated test scripts to simulate normal user requests and repeatedly record the trace logs generated by the application under normal operation. These trace logs serve as the basis for our analysis of normal requests, and are used to generate RWDG and RSS for normal requests. After the approach described before, we transformed them into RWDGs and RSSs as a way to describe the possible behavior of the system in a normal operating environment. Then, we use two algorithms to set the rules for determining performance anomalies and workflow anomalies based on these trace logs.

In real production environments, faults tend to occur randomly and last for some time, and disappear with spontaneous or human corrections. Therefore, for practical validation, we first ensure that the TrainTicket application runs in a normal environment for 10min and simulate user requests and generate trace data during this time. After this, we inject faults in the TrainTicket application and simulate user requests, keeping the application running for 5 min in the presence of faults. Then, we collect these trace logs containing faults, which are recorded and tested as data to be tested. Finally, we remove the fault from the application to simulate the operation of the application after the recovery from the faults and collect the trace logs for 5 min.

In our experiments, we first keep the TrainTicket application running normally in the experimental environment and monitor the application's normal request processing behavior by simulating user actions with an automated test script for 60 min. This part of the monitoring data is used as the system's historical normal operation data in order to build the normal requests' RWDGs and RSSs. Then, we start the fault injection experiments, and the experimental injection faults and implementation steps are described in subsection III.B of this paper. Each experiment is conducted for 20 min and the original trace log files are collected. Based on the type of injected faults, we divide the log files into 8 categories corresponding to injected faults F1~F8. After that, we use 90% of each log files as the training set to train the detection model

Table 1 Actual injected faults and fault descriptions

Injecting Faulty Objects	Fault numbers	Root causes	Details
TrainTicket Application	F1	OOM errors of JVM.	Causes the application's heap memory footprint to exceed the Xmx limit, triggering an overflow.
	F2	Network communication delayed between microservices.	Make the communication of ts-cancel-microservice incur a 200ms delay and timeout for partially sent cancellation requests.
	F3	Network communication lost and generates unexpected errors.	Make the communication of ts-travel2-microservice generate 70% packet loss, and some requests to send notifications are lost.
	F4	The microservice was accidentally deleted.	Delete the pod: ts-admin-user-microservice, which is used to provide user management operations.
	F5	The CPU of the Java process in the container is fully loaded.	CPU full for java process in ts-admin-route-microservice service container.
	F6	Unexpected increase in processing time for some requests due to latency in the method.	The createNewOrder method in the java process of the ts-order-microservice service injects 3s fault, which controls the generation of new orders.
Containers	F7	Triggers an error hang of the microservice container, resulting in unexpected errors in business processing.	Hang the container of ts-notification-microservice.
	F8	The CPU load in the container is 100%.	Increase the pod's CPU load of ts-config-microservice

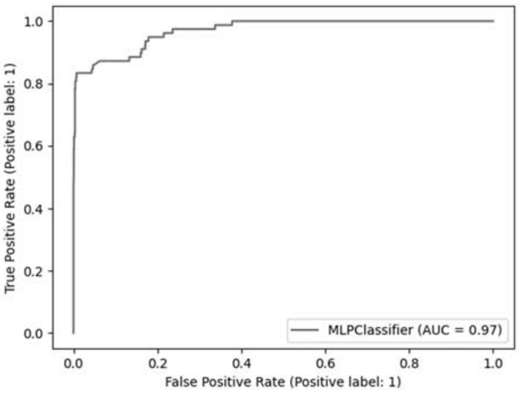

Fig. 4 ROC of Fault detection

and use the remaining 10% as the test set to verify the detection effectiveness of the model.

For faults generated by physical resources and containers, the resource release will make the faults disappear automatically and a part of Kubernetes pods will automatically restart the deleted containers. Therefore, in our experiments, when verifying this type of faults, we use a 10min-normal-5min-fault-5min-normal operation approach to simulate a system that is affected by the fault and recovers itself. We use a test script to simulate the normal operation of users to generate the original dataset. After data extraction and pre-processing of the raw dataset, we can transform it into the type of parameters needed for the experimental model.

In order to evaluate the method proposed in this paper, we will investigate the following three questions:

RQ1: For the faults in our experiments, how accurate is the proposed algorithm in terms of fault detection and fault location in this paper? How accurate is it in the face of unexpected faults in actual operation?

RQ2: For abnormal faults that are not directly caused by the application's own faults, such as physical resource faults and container faults, can the algorithm proposed in this paper discover, locate and diagnose the cause of the faults?

RQ3: What is the actual overhead of the proposed algorithm in this paper? How much impact will it have on the operation of the application?

C. *Prediction Accuracy(RQ1)*

To answer RQ1, we evaluate our approach in three aspects: fault detection, fault localization, and fault cause detection. We collected a total of 383863 trace logs in our experiments, which were processed to generate 14478 RWDGs and RSSs, of which 709 contained injection faults, accounting for 4.8% of the sample size. We will show the accuracies of our approach.

1) *Fault Detection Accuracy*

Fault detection is the basis of the whole fault diagnosis method. We will first check the effectiveness of the fault detection method we propose in this paper.

We implement fault detection algorithms that use the directed edge weight ranges of RWDGs and the character sequences of RSSs to detect anomalies in applications. Then, we will demonstrate the accuracy of the fault detection algorithm of our approach. The metrics of our experiments are: True Positive, or TP, is the request that our approach thinks there is a fault and the fault does exist in this request; False Positive, or FP, is the request that our approach thinks there is a fault and the fault does not exist in this request; True Negative, or TN, is the request that our approach thinks there is no fault and the fault does not exist in this request; False Negative, or FN, is the request that our approach thinks there is no fault and the fault does exist in this request. We use the precision, recall and F1-score for evaluation.

Table 2 shows the fault detection accuracy in the experiments of our approach. Fig. 4 shows the ROC curves of fault detection for the overall test set, which can measure the performance of the model.

The results show that our approach can effectively describe the behavior of the application, and the already trained model can effectively discover potential faults in the application. For the various types of faults injected, the average fault discovery accuracy after balanced is 0.915, the average recall rate is 0.890, and the average F1 value is 0.886. The results show that our approach can effectively discover whether there are potential faults in the application.

2) *Fault Diagnosis Accuracy*

In this section, we experiment with the fault detection algorithm that uses the directed edge weight ranges of RWDGs and the character sequences of RSSs as inputs to the DNNs to train the model and locate the faults in the application. Then, we show the accuracy of the fault diagnosis model of our approach.

Table 3 shows the fault diagnosis accuracy in the experiments of our approach. For the detection of fault-affected services, the proposed method achieves a good result. The average values of accuracy, recall, and F1 value are 0.852, 0.820, and 0.812, which shows that our approach can effectively fit the relationship between the change of service request processing flow and the fault-affected microservices. The results show that when unknown faults appear, the trace model of known faults can be used to match with the trace model of unknown cause faults to diagnose unknown faults.

We note that the experimental results for the injection faults F2, F3, and F5 have good performance in our experimental environment, which is because: whether the physical nodes or the containers themselves encounter resource-based faults, they will trigger significant changes in the response time of the corresponding services, and the proposed RWDG models in this paper itself can well represent the changes in the response time of requests between services. Besides, the loss of service communication leads to the absence of the corresponding call chain and trace log, which is reflected by the RSSs' character sequences, and therefore can be detected well. However, for the two types of faults F6 and F7, the effect of our approach proposed in this paper is not very satisfactory. This is because the generation of these two types of faults not only leads to changes in the response time of the microservice injected into the fault, but also leads to changes in the processing logic of subsequent requests. Our proposed model has difficulty in predicting the differences in microservices' performance between the

Table 2 Faults Detection Results

Fault No.	Precision	Recall	F1-value
F1	0.907	0.888	0.887
F2	0.979	0.979	0.979
F3	0.921	0.907	0.906
F4	0.953	0.949	0.949
F5	0.963	0.960	0.960
F6	0.832	0.747	0.729
F7	0.914	0.906	0.906
F8	0.847	0.780	0.768
Average	0.915	0.890	0.886

Table 3 Faults Diagnosis Results

Fault No.	Precision	Recall	F1-score
F1	0.843	0.772	0.759
F2	1.000	1.000	1.000
F3	1.000	1.000	1.000
F4	0.917	0.900	0.899
F5	0.977	0.976	0.976
F6	0.515	0.384	0.323
F7	0.649	0.629	0.638
F8	0.917	0.900	0.899
Average	0.852	0.820	0.812

Table 4 Fault Types Diagnosis Results

Fault No.	Precision	Recall	F1-value
F1	1.000	1.000	1.000
F2	1.000	1.000	1.000
F3	1.000	1.000	1.000
F4	0.875	0.833	0.829
F5	0.977	0.976	0.976
F6	0.516	0.390	0.333
F7	0.649	0.629	0.638
F8	0.917	0.900	0.899
Average	0.867	0.841	0.834

Table 5 Performance Overhead

Application	CPU	Average Response Time
TrainTicket	4.6%	1.21s
TrainTicket with Jaeger	5.1%	1.32s

changed abnormal processing logic and the similar normal logic. Therefore, the predict model may make wrong judgments, making the experimental results not idealized. This type of faults is the main direction of our future research.

3) Fault Type Diagnosis Accuracy

We try to redesign the classification represented by each element in the output vector of the fault discovery algorithm from the service affected by the fault to the type number of the injected fault (F1~F8). After re-tune, train and test our model, we can further predict the cause of the generated faults, and the results are shown in Table 4.

For fault type diagnosis, the average values of accuracy, recall, and F1 are 0.867, 0.841, and 0.834. In this case, the model is able to correlate the changed features with the injected fault types by learning the changes in RWDGs and RSSs so that the model can classify fault types. However, such fault types diagnosis method can only satisfy the diagnosis of the known fault types associated with the injected faults and cannot infer the general performance caused by such faults based on the injected known faults. Thus, it is hard to satisfy the need for fault localization when the faults of the same cause occur on different microservices. We will do further research on this fault localization method in the future works.

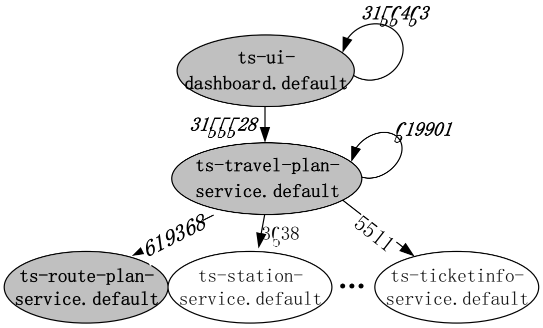
316463
ts-ui-dashboard.default
316628
619901
ts-travel-plan-service.default
619368
3638
5511
ts-route-plan-service.default
ts-station-service.default
...
ts-ticketinfo-service.default

Fig. 5 Corresponding RWDG

D. Case Study(RQ2)

Trace models are the fundamental objects of our analysis. We will verify that trace models can be used to find faults implied in trace logs. We take a representative example, which is generated by a typical performance fault. We extract the span content as *[traceID, callerName, parentSpanID, microserviceName, spanID. duration, perfError]*. *traceID* is a unique identifier for the trace log; *callerName* is the name of the caller microservice of the current microservice; *parentSpanID* is the parent span's id; *microserviceName* is the name of the current microservice; *spanID* is the current span's id; *duration* is response time; *perfError* indicates whether the http.error_code is returned. The RWDG of the case is shown in Fig. 5. The invocating response time between microservices ts-ui-dashboard.default, ts-travel-plan-microservice.default and ts-route-plan-microservice.default exceeded the threshold value, and there was http.error_code exist. Therefore, our approach predicts that these microservices are at risk of receiving potential fault effects. In fact, our approach also determines that the rest of the services are affected by the failure as well. And in this case, the performance failure shown does cause all the above services to be affected by the failure.

The faults of the above fault RWDGs themselves are not caused by the microservice application itself, but by the increase in microservice response time due to the resource occupation of the physical node where the application is located. And as the response time increases, faults such as http request timeout may occur and generate the http-error keyword. Therefore, the behavior of the system is described using the proposed RWDGs in our approach, which itself can reflect the impact of physical resource faults on the microservices.

E. Overhead of the Approach (RQ3)

The overhead of our approach consists of tracking log generation and collection, tracking log analysis, and model training. The generation and collection of trace logs are handled by a distributed trace tool, and a proxy is used to manage the network traffic to and from the service to generate trace logs that are stored permanently in the back-end database. We designed an experiment to examine the impact of trace log generation and collection on microservice applications. In the experiment we simulate a 1000 concurrent admin login process and compare the performance of the original TrainTicket application with the TrainTicket application monitored by the distributed tracing tool Jaeger. CPU utilization and average response time are two commonly used metrics to describe the performance of microservice applications, and we evaluate the experiments by examining the variation of these two experimental metrics. The results of the experiments are shown in Table 5.

According to the experiment results, the CPU usage increased from 4.6% to 5.1% and the average response time increased from 1.21 seconds to 1.32 seconds. Thus, we learn that the overhead of trace log generation and collection is small. For microservice applications, this overhead is acceptable. In fact, in a study by Song et al [27], it has been shown that such a trace log generation and collection approach have little impact on the performance of the original microservice application. The main overheads considered in the experiments are the trace log analysis overhead and the training overhead. Then, we analyze the complexity of the model. We generate the corresponding trace models for the 383863 trace logs and count the time spent. The average time spent under five experiments is 2813.26 seconds, which can process 136.4 trace log files per second on average. As for the training, the DNNs model has a high time complexity for each training. he time complexity of backpropagation is $O(n*m*h^{k}*o*i)$ [29], where *n* is the number of training samples; *m* is the number of features; *k* is the number of hidden layers; *h* is the number of neurons in each hidden layer; *o* is the number of output neurons; *i* is the number of iterations. And training the model requires a lot of feedforward and backpropagation processes. Such a time overhead is obviously large. However, our proposed approach achieves the decoupling of data collection and data analysis. We can deploy the trace log analysis and training part on another high-performance server, thus not affecting the performance of the original microservice application.

IV. Related Work

The methods detect faults by analyzing metrics. The method proposed by Márquez etc. [14] evaluates frameworks assemblies in microservices-based systems, by using imperfect information of non-functional requirements and framework that may be incomplete, inaccurate or changing. The method proposed by Sánchez etc. [15] builds a self-modeling framework based on SDN(Software-Defined Networking) and NFV(Network Functions Virtualization). The framework mainly includes three aspects of work: the definitions of multi-layered templates, a dynamically generated diagnosis model based on definitions, and a service-aware root-cause analysis module. And the method proposed by Pina etc. [17] registers all calls to and between microservices, as well as their responses, by logging the activities of Zuul [18] (the gateway of Netfiix). However, none of the models built by the above methods model or classify abnormal requests and normal requests separately. This paper uses the existing historical models to compare with the unknown trace models. We can detect the possible causes of the faults without modifying the existing models.

The following methods are used for fault diagnosis in terms of mining log files. The non-invasive method proposed by Cinque etc. [19] collects trace information and generated logs to help the operation and maintenance personnel troubleshoot related faults. It analyzes the HTTP headers in the form of capturing network data packets, to judge the types of requests and calculate trace information. However, it still has the defects of performance consumption and log file reduction under high load conditions. The method proposed by Zamfir etc. [12] builds a machine-learning monitoring framework based on big data database systems (e.g. Elasticsearch). The framework can't locate the location or the causes of the fault quickly as a result of unclassified known faults. We can use fault injection to obtain the behavior of the system with faults.

The following methods are used for fault diagnosis in terms of tracing systems' execution. The method proposed by Ma etc. [16] generates relationship dependency graphs by analyzing the chain of service calls. Specifically, it analyzes service dependencies to find potentially risky calls, and find anomalies in the target system. The method proposed by Cinque etc. [13] can mine fault indicators from traces, and can also find the faults without containing the wrong keyword directly. But it doesn't analyze the connection between the causes and the locations of the faults. We describe the relationship between fault behaviors and causes, by pre-recording the overview of the correct request processing and the behaviors of the system after injecting faults. When a request in an unknown state occurs, we compare the trace information of the current request instead of modifying the known trace models, which has less resource overhead.

V. Conclusion and Future Work

Distributed microservice-based applications have the advantages of scalability, fault tolerance, high availability, but they increase the difficulty of effectively detecting and locating faults of microservices. This paper proposes a trace based intelligent faults diagnosis with deep learning. We build request weighted directed graphs and request strings to characterize the behaviors of microservices, train a fault diagnosis model based on the deep neural network with the trace dataset built by injecting faults, and then detect and diagnose faults.

We plan to improve our approach in the future work as follows. The injected faults cannot cover some rare and complex faults, and we will enrich the injected fault types. Furthermore, because the performance overhead of the diagnosis method is proportional to the number of recorded traces, we will further carefully select the number and types of injected faults, and select suitable injection points to achieve the high accuracy of fault diagnosis efficiently.

Acknowledgment

This work is supported in part by National Key R&D Program of China (No. 2017YFB1400804), National Natural Science Foundation of China (No. 61872344), and Youth Innovation Promotion Association of Chinese Academy of Sciences Fund (No. 2018144).

References

[1] S. Chouliaras and S. Sotiriadis, "Real-Time Anomaly Detection of NoSQL Systems Based on Resource Usage Monitoring," in IEEE Transactions on Industrial Informatics, vol.16, no. 9, pp. 6042-6049, 2020.

[2] S. Sotiriadis, N. Bessis, C. Amza and R. Buyya, "Elastic Load Balancing for Dynamic Virtual Machine Reconfiguration Based on Vertical and Horizontal Scaling," in IEEE Transactions on Services Computing, vol. 12, no. 2, pp. 319-334, 2019.

[3] J. Hochenbaum, O. S. Vallis and A. Kejariwal, "Automatic anomaly detection in the cloud via statistical learning", 2017, [online] Available: https://arxiv.org/abs/1704.07706.

[4] Y. Yuan, W. Shi, B. Liang and B. Qin, "An Approach to Cloud Execution Failure Diagnosis Based on Exception Logs in OpenStack," 2019 IEEE 12th International Conference on Cloud Computing, 2019, pp. 124-131.

[5] J. Xu, P. Chen, L. Yang, F. Meng and P. Wang, "LogDC: Problem Diagnosis for Declartively-Deployed Cloud Applications with Log," 2017 IEEE 14th International Conference on e-Business Engineering (ICEBE), Shanghai, 2017, pp. 282-287.

[6] T. Jia, Y. Li, C. Zhang, W. Xia, J. Jiang and Y. Liu, "Machine Deserves Better Logging: A Log Enhancement Approach for Automatic Fault Diagnosis," 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Memphis, TN, 2018, pp. 106-111.

[7] E. Chuah et al., "Enabling Dependability-Driven Resource Use and Message Log-Analysis for Cluster System Diagnosis," 24th International Conference on High Performance Computing, 2017, pp. 317-327.

[8] S. Zhang, Y. Wang, W. Li and X. Qiu, "Microservice failure diagnosis in microservice function chain," 2017 19th Asia-Pacific Network Operations and Management Symposium, 2017, pp. 70-75.

[9] N. M. Popa and A. Oprescu, "A Data-Centric Approach to Distributed Tracing," 2019 IEEE International Conference on Cloud Computing Technology and Science, Sydney, Australia, 2019, pp. 209-216.

[10] J. Mace, R. Roelke and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems", ACM Trans. Comput. Syst., 2018.

[11] Spiegel, M. R. Theory and Problems of Probability and Statistics. New York: McGraw-Hill, pp. 116-117, 1992.

[12] V. Zamfir, M. Carabas, and C. Carabas, "Systems Monitoring and Big Data Analysis Using the Elasticsearch System," International Conference on Control Systems and Computer Science, 2019, pp. 188-193.

[13] M. Cinque, R. Della Corte and A. Pecchia, "Discovering Hidden Errors from Application Log Traces with Process Mining," 2019 15th European Dependable Computing Conference, Naples, Italy, 2019, pp. 137-140.

[14] G. Márquez, Y. Lazo and H. Astudillo, "Evaluating Frameworks Assemblies In Microservices-based Systems Using Imperfect Information," IEEE International Conference on Software Architecture Companion (ICSA-C), Salvador, Brazil, 2020, pp. 250-257.

[15] J. M. Sánchez, I. G. Ben Yahia and N. Crespi, "Self-modeling based diagnosis of microservices over programmable networks," 2016 IEEE NetSoft Conference and Workshops (NetSoft), Seoul, 2016, pp. 277-285.

[16] S. Ma, C. Fan, Y. Chuang, W. Lee, S. Lee and N. Hsueh, "Using Microservice Dependency Graph to Analyze and Test Microservices," 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Tokyo, 2018, pp. 81-86.

[17] F. Pina, J. Correia, R. Filipe, F. Araujo and J. Cardroom, "Nonintrusive Monitoring of Microservice-Based Systems," 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA), Cambridge, MA, 2018, pp. 1-8.

[18] Zuul., https://github.com/Netflix/zuul.

[19] M. Cinque, R. Della Corte and A. Pecchia, "Microservices Monitoring with Event Logs and Black Box Execution Tracing," in IEEE Transactions on Services Computing, 2019.

[20] X. Zhou et al., "Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study," in IEEE Transactions on Software Engineering, 2020.

[21] Microservice System Benchmark. 2020. TrainTicket. Retrieved September 11, 2020 from https://github.com/FudanSELab/train-ticket/

[22] X. Larrucea, I. Santamaria, R. Colomo-Palacios and C. Ebert, "Microservices," in IEEE Software, vol. 35, no. 3, pp. 96-100, 2018.

[23] Kubernetes.Com. 2020. Kubernetes. https://kubernetes.io/

[24] Istio. 2020. Istio. https://istio.io/

[25] Jaeger. 2020. Jaeger. https://www.jaegertracing.io/

[26] Elasticsearch. 2020. Elastic. https://www.elastic.co/cn/elasticsearch/

[27] M. Song, Q. Liu and H. E., "A Microservice Tracing System Based on Istio and Kubernetes," 2019 IEEE 10th International Conference on Software Engineering and Microservice Science (ICSESS), Beijing, China, 2019, pp. 613-616.

[28] Katalon. 2021. https://www.katalon.com/

[29] "Efficient BackProp" Y. LeCun, L. Bottou, G. Orr, K. Müller - In Neural Networks: Tricks of the Trade 1998.

[30] Trace Analyzer. 2021. https://github.com/BIGXT/Tracelogstotraining.

[31] Experiment Dataset. 2021. https://github.com/BIGXT/TTFI.