Fault Diagnosis for Test Alarms in Microservices through Multi-source Data

Shenglin Zhang*
Nankai University & HL-IT
Tianjin, China

Jun Zhu
Nankai University
Tianjin, China

Bowen Hao
Nankai University
Tianjin, China

Yongqian Sun†
Nankai University & TKL-SEHCI
Tianjin, China

Xiaohui Nie†
CNIC, CAS
Beijing, China

Jingwen Zhu
Nankai University
Tianjin, China

Xilin Liu
Huawei Cloud
Shenzhen, China

Xiaoqian Li
Huawei Cloud
Shenzhen, China

Yuchi Ma
Huawei Cloud
Shenzhen, China

Dan Pei§
Tsinghua University & BNRist
Beijing, China

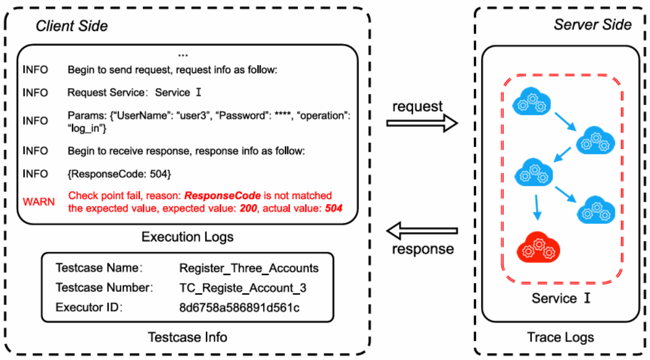
***Abstract*—Nowadays, the testing of large-scale microservices could produce an enormous number of test alarms daily. Manually diagnosing these alarms is time-consuming and laborious for the testers. Automatic fault diagnosis with fault classification and localization can help testers efficiently handle the increasing volume of failed test cases. However, the current methods for diagnosing test alarms struggle to deal with the complex and frequently updated microservices. In this paper, we introduce *SynthoDiag*, a novel fault diagnosis framework for test alarms in microservices through multi-source logs (execution logs, trace logs, and test case information) organized with a knowledge graph. An Entity Fault Association and Position Value (EFA-PV) algorithm is proposed to localize the fault-indicative log entries. Additionally, an efficient block-based differentiation approach is used to filter out fault-irrelevant entries in the test cases, significantly improving the overall performance of fault diagnosis. At last, *SynthoDiag*is systematically evaluated with a large-scale real-world dataset from a top-tier global cloud service provider, Huawei Cloud, which provides services for more than three million users. The results show the Micro-F1 and Macro-F1 scores improvement of *SynthoDiag*over baseline methods in fault classification are 21% and 30%, respectively, and its top-5 accuracy of fault localization is 81.9%, significantly surpassing the previous methods.**

***Index Terms*—Microservice, Test Case, Fault Diagnosis, Execution Logs, Trace Logs**

I. Introduction

Microservices represent an architectural paradigm that decomposes the functional aspects of software applications into a collection of small, lightweight services [23]. This approach enables the division of the business functionality into independent segments that can be concurrently developed, meeting the demand for frequent iterations and rapid updates [22]. To guarantee the reliability of service functionality, rigorous testing is conducted before launching or updating any business features [7]. Many research works have focused on developing test cases to thoroughly assess each component in a microservice[6,18].

During the execution of test cases, several factors can lead to failed test cases and raise test alarms, including environment issues, code errors, incorrect testing procedures, etc. Testers are responsible for addressing these test alarms by analyzing massive log files (as shown in Figure 1) scattered across different service modules, classifying the fault category, and diagnosing the root cause of each failed test case [1,11]. In this way, they can take the right remedial actions, such as resetting environmental configurations, reporting exception messages to developers, or rectifying test scripts[1,11].



```
                    ...
INFO   Begin to send request, request info as follow:
INFO   Request Service:  Service Ⅰ
INFO   Params: {"UserName": "user3", "Password": ****, "operation":
       "log_in"}
INFO   Begin to receive response, response info as follow:
INFO   {ResponseCode: 504}
WARN   Check point fail, reason: ResponseCode is not matched
       the expected value, expected value: 200, actual value: 504
```

Testcase Name: Register_Three_Accounts
Testcase Number: TC_Registe_Account_3
Executor ID: 8d6758a586891d561c

Fig. 1: **Multi-source logs in a failed test case. The execution logs are collected fromtheexecutor(clientside), andthetrace logs are collected from the microservice system (server side), and the testcase info is predefined by testers.**

A thorough classification of failed test cases and a detailed localization of fault-indicative logs is crucial for testers handling test alarms [11]. However, manually analyzing these logs is time consuming and laborious[1,8]. For example, we conducted a study at a top-tier global cloud service provider, Huawei Cloud, which provides services for over three million users. Thousands of test cases are executed daily, and even if only around 5% of these test cases result in faults, hundreds of test cases make it impractical for manual fault diagnosis. Previous studies [1, 2, 5, 11] classify failed test cases by their fault category or pinpoint the fault-indicative logs within the execution logs of test alarms. Nevertheless, they only process the execution logs while neglecting the trace logs, which could result in suboptimal fault diagnosis performance in the microservice system (see Section 2.2.1). Hence, it is urgently needed to design a new automatic fault diagnosis framework, including fault classification and localization for test alarms, by analyzing multi-source logs in microservices. Yet, it faces the following three challenges (see section 2.2.3 for more details):

Challenge 1: Underutilized multi-source logs. The disparate formats of multi-source logs render previous methods ineffective in leveraging them, particularly in integrating execution logs (semi-structured texts) with trace logs (tree-structured texts with span).

Challenge 2: Inefficient fault-irrelevant logs filtering. Not all logs in failed test cases are fault-related. Existing methods cannot effectively filter irrelevant entries while preserving relevant ones.

Challenge 3: Inaccurate localization for new types of logs. Frequent updates cause fault log content variations, making historical references inadequate for new failures.

This paper proposes SynthoDiag, a novel fault diagnosis frame work for test alarms in microservice systems through multi-source logs. (1) To address challenge 1, recognizing the capacity of knowl edge graphs to establish correlations among data in varying for mats [4, 12, 21], we propose the utilization of knowledge graphs for diagnosing failed test cases that involve multi-source logs; (2) To address challenge 2, we propose an effective block-based differenti ation strategy to eliminate fault-irrelevant logs while preserving the contextual information of the remaining logs that are relevant to faults; (3) To address challenge 3, we propose an Entity Fault As sociation score and Position Value (EFA-PV) algorithm to determine the significance of each log entry by capitalizing on the content of the logs as well as their relationships in multi-source logs. The contributions of this paper are summarized as follows:

- As far as our knowledge extends, we are among the first to simultaneously conduct fault classification and localization for test alarms in microservice systems, and SynthoDiag is intro duced as the premier framework to employ multi-source logs, encompassing execution logs, trace logs, and test case informa tion, for the diagnosis of failed test cases.
- Wepropose a simple yet effective block-based differentiation strategy to enhance data quality, thereby addressing the chal lenge introduced by the fault-irrelevant logs in the training data.
- Wepropose EFA-PV to assess the significance of log entries, leveraging both the content of log entries and the relationships between log entries.
- Tosubstantiate the effectiveness of SynthoDiag, a comprehen sive evaluation is conducted using a real-world test case dataset obtained from Huawei Cloud. The improvements demonstrated by SynthoDiag over baseline methods in fault classification are remarkable, with Micro-F1 scores and Macro-F1 scores experi encing a notable increase of 21% and 30%, respectively. Further more, it achieves a top-5 accuracy of 0.819 in fault localization.

The subsequent sections of this paper are structured as follows: Preliminary knowledge is expounded upon in section 2, while the frame work of SynthoDiag is delineated in section 3. To demonstrate effective ness, the experiment setup and results are elaborated upon in section 4. section 7offers anoverview of related works, culminating in a conclusion in section 8.

II. PRELIMINARY AND MOTIVATION

A. *Background*

The structure of a test case is intricately designed, incorporating a sequence of operations along with their respective checkpoints. This design is pivotal for evaluating the functionality of services during the System and Integration Testing (SIT) phase [11].

Test Alarm Analysis. In the context of SIT, the occurrence of a failure within a test case is signified by the triggering of an alarm. This alarm indicates a discrepancy between the actual outcome of an operation and its anticipated result, which may arise due to various factors such as an unstable network environment, incorrect execution of test steps, or underlying issues within the service itself. To effectively address these alarms, testers are required to implement tailored solutions that are contingent upon the specific category of the fault identified.

TABLE I: **Cause and solution for different fault categories**

Fault Category	**Cause**	**Solutions**
Service Issue	Issues within the service	Submitting bug reports to the developers
Environment Issue	Issue external to the service and unrelated to the test steps	Re-executing test case or setting the environment
Script Defect	Wrong test steps	Correcting the test script
Tool Defect	Third-party test tools defect	Asking help from supporters of the third-party tools

Fault Category. Test case faults are typically categorized into the following four categories in Table 1. Among these, Environment Issue and Script Defect emerge as the most prevalent categories. Environment Issue pertains to failures induced by external environ mental factors, whereas Script Defect relates to failures caused by inaccuracies within the test scripts. The complexity of microser vices environments and the dynamic nature of test script updates often contribute to the frequency of these issues. Conversely, Tool Defect represents a highly impactful category, as a single flaw in the testing tool can lead to multiple test case failures. Meanwhile, Service Issue stands out as the most critical category, given its role in uncovering potential service problems and facilitating preemptive measures to avert severe incidents. To accurately determine the category of a failed test case, testers are reliant on the analysis of logs from multiple sources.

It is worth noting that in ourscenario, there is already a historical fault case category library, where many cases have been marked in these four categories by experienced testers, without the need to build it from zero.

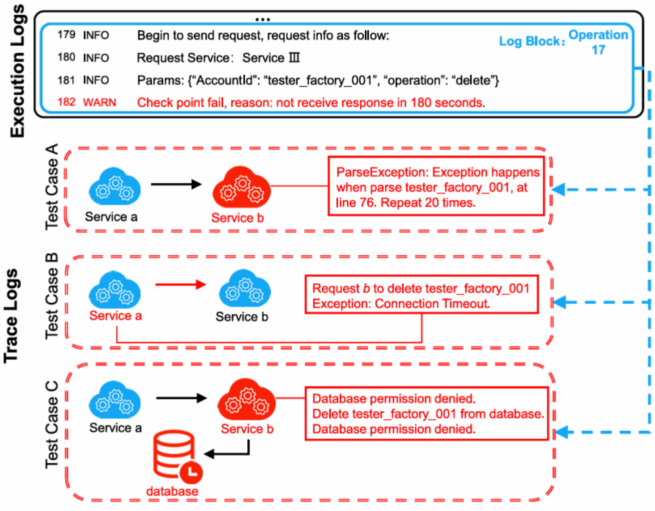
Multi-source Logs. Log plays a pivotal role in diagnosing faults due to their semantic information of services and operations [11, 21, 26] and the source of the log determines the focus of the content. As shown in Figure 5, each test case generates three types of logs, execution logs originating from the client side, trace logs from the server side, and testcase info pre-written by the tester:

- *Execution Logs:* The execution log is mainly composed of test information for manytest operations. There is a clear beginning and end (e.g., Begin to send request and Check point fail entry in Figure 1) for each operation. Specific testing content was recorded between them, including testing preparations, testing operations, request parameters, response messages, checkpoint outcomes, error messages etc. These messages can help us clas sify the test case into which category, often requires a combi nation of trace logs to achieve more accurate classification (see Section 2.2.1). Note that an operation in the execution log corresponds to a set of trace logs on the server side.
- *Trace Logs:* These logs are generated on the server side, and encompass invocation flow traces among services and logs gen erated by each service. There are multiple groups of trace logs in a test case, which is decided by the request counts in the execution logs, and each group of trace logs can be related to an operation in the execution logs. As such, trace logs meticu lously capture operational intricacies of microservice instances and messages about faults arising from service issues and envi ronmental issues.
- *Testcase Information.* It is written by the tester before testing and records the basic information about the test case, including the purpose, id, executors, etc., of the test case. This information can be used to understand the test case and find the passed test cases with the same purpose in history, which can help us to make the comparison and understand the cause of the fault more quickly.

Fault Diagnosis. Fault diagnosis in this paper refers to the classification and localization of faults for failed test cases. Fault classification is to determine which existing fault category a new case belongs to, while fault localization is to identify and highlight which specific log entries are most likely to be related to the root cause, i.e., fault-indicative log entries (e.g., the Request b to delete tester_factory_001 Exception: Connection Timeout in Figure 2). Fault diagnosis can assist testers in quickly identifying and resolving faults.

B. Motivation

Analyzingfaults throughlogsisalabor-intensive task, necessitating meticulous scrutiny of individual log entries by testers to accurately identify the underlying causes of faults. Given the sheer volume of failed test cases and the extensive number of logs, manual inspec tion of each test case becomes impractical. Therefore, providing the tester with the fault category and fault-indicative log entries can significantly enhance testing efficiency. These objectives align with the overarchinggoalsoftest casefault diagnosis[1, 11]. Nonetheless, when diagnosing failed test cases within a microservice system, we confront several challenges, including underutilized multi-source logs, inefficient fault-irrelevant logs filtering, and inaccurate local ization for new template logs.



```
179 INFO    Begin to send request, request info as follow:
180 INFO    Request Service:  Service Ⅲ
181 INFO    Params: {"AccountId": "tester_factory_001", "operation": "delete"}
182 WARN    Check point fail, reason: not receive response in 180 seconds.
```

```
ParseException: Exception happens
when parse tester_factory_001, at
line 76. Repeat 20 times.
```

```
Request b to delete tester_factory_001
Exception: Connection Timeout.
```

```
Database permission denied.
Delete tester_factory_001 from database.
Database permission denied.
```

Fig. 2: **The same error logs in the execution logs can be caused by diverse reasons as indicated in the trace logs.**

2.2.1 Underutilized Multi-source Logs. The inherent complexity of microservice systems necessitates the analysis of multi-source logs to accurately diagnose fault test cases.

However, prior research has predominantly concentrated on fea ture extraction from execution logs to diagnose fault categories, because execution logs contain comprehensive information about the test case in the traditional system. This is no longer applicable in the scenarios of a microservice system, because the intricate environment introduces complexity, whereby the same log in exe cution logs may stem from various underlying causes. For instance, Figure 2 show three distinct test cases exhibit an log entry of iden tical error message. Based solely on the execution logs, one might assume that they belong to the same fault category. However, a closer examination of trace logs reveals disparate fault causes for each case. Test case A encounters fault due to an infinite loop in the service progress, classified as service issue. In contrast, test case B's fault results from network interference which lead to the loss of the response message, categorized as environment issue. Lastly, test case C's fault is caused by permission denied from the database and is also characterized as service issue due to the wrong configuration of service. Consequently, these diagnosing methods of the test case in traditional, which only focus on the execution logs, are no longer adapted to the scenes of the microservice system, and multi-source logs must be used.

Challenge 1: Fault diagnosis within microservices cannot solely depend on execution logs and there is a need for analyzing multi source logs.

2.2.2 Inefficient Fault-irrelevant Logs Filtering. Leveraging multi source logs for fault diagnosis typically involves analyzing the entirety of the log data. However, most of the logs are irrelevant to the failure, and often only a small fraction of the logs is critical to helping testers diagnose the failed test case. Remarkably, after carefully investigating thousands of failed test cases collected from Huawei Cloud,we find that less than 10% of thes elog entries pertain to faults. As is shown in Section4.3.1,this presence of irrelevant lo gentries poses challenges to characterizing failed test cases accurately, which leads to the final performance of the model dropping by more than 10%.

The prior research, such as LFF [1], attempts to address this issue by using a template-based differentiation technique. This method filters out the logs in the failed test case that share the same template with the logs in the historically successful test cases. On the one hand, the template is only the format of the output and does not determine whether it is related to the exception, which means that the fault-relevant logs may have the same template as the successful logs will also be filtered. On the other hand, the relationship between log entries, especially the context information, will be lost with this method. In this way, as is shown in Section 4.3.1, these incorrectly removed fault-relevant logs and lost messages even cause performance worse than without filtering.Consequently, we will propose the concept of log block in (Section 3.2, which divides logs according to test operations, which can more effectively filter irrelevant logs while retaining contextual information.

Challenge 2: Fault-irrelevant logs in the microservices will con fuse the diagnostic model and previous methods cannot filter out them effectively.

2.2.3 Inaccurate Localization for New Template Logs. Upon pre senting testers with fault categories, it becomes essential to provide further details to enhance the interpretability of faults, which helps testers to comprehend faults better and verify the accuracy of diag nostic results through the localization of fault-indicative logs.

The strategy proposed by LFF [1] is based on the premise that logs associated with templates frequently observed in historical failed test cases are more likely to indicate the underlying faults. Accordingly, this approach prioritizes logs in new failed test cases by quantifying the presence of their corresponding templates in past failures. Nevertheless, due to the independent and frequent deployment of services, microservice system test cases undergo rapid updates, leading to the generation of the new template [7]. On the one hand, these newly generated templates will not appear in the historical failed test cases. On the other hand, the log with these newly generated templates can be the real fault-indicative log. As is shown in Section 4.2, these methods achieve a bad performance on the Top-5 accuracy. Consequently, fault-indicative log entries in new test cases can not be localized by the existing method.

Challenge 3:The updateoftest cases leads to a mismatchbetween new logs and old logs, which makes previous methods no longer applicable to localize the historical fault-indicative log entries.

2.2.4 Summary. To diagnose failed test cases in microservices ef fectively, it is imperative to utilize both trace logs and execution logs. Furthermore, addressing the disparity in the formats of these logs from various sources is essential. Additionally, in light of the detrimental effects of irrelevant log entries within the training data, a more robust filtering mechanism must be devised. Moreover, to enhance the interpretability of diagnostic results, there is a need to propose a localization method that adapts to the scenes of the microservice system.

III. **DESIGN**

A. The Framework of SynthoDiag

The SynthoDiag framework comprises three main components: log f iltering, case embedding, and fault diagnosing, as illustrated in Figure 3. When parsing the logs of a failed test case, SynthoDiag f ilters the irrelevant log entries in logs with log-block Section 3.2 (ad dressing challenge 2). It then builds the knowledge graph of the failed test cases based on the remaining fault-relevant log entries Section 3.3 (addressing challenge 1). Next, SynthoDiag uses KGE with semantic information to embed each failed test case into a vector Section 3.4. Finally, the historical failed test cases and labels will be stored in a case category library. When a new failed test case needs to be diagnosed, SynthoDiag gets the case vector and knowledge graph of this test case with the same steps, and then it will output the fault category and localize the fault-indicative log entries in the test case for interoperability Section 3.5 (addressing challenge 3).

B. Fault-Irrelevant Log Filtering

As previously elucidated in Section 2.1, a test case comprises a single set of execution logs and multiple sets of associated trace logs. The execution logs of each operation can be related to the trace logs and the removal of irrelevant log entries from execution logs allows for the elimination of redundant trace logs. Consequently, SynthoDiag focuses exclusively on eliminating irrelevant log entries from execution logs with two steps: log block division and irrelevant log blocks removment.

Log block division. As mentioned at the end of Section 2.2.2, we involve the concept of log block to divide execution logs according to test operations for retaining contextual information. What is a log block? We know that the execution log consists of test operations' information. A test operation is the smallest unit for a test. Consequently, we define a log block corresponding to a test operation, which contains all test information related to that specific operation.

How to divide log blocks? As described in Section 2.1, there is a clear beginning and end for each test operation, which is pre-defined by testers. The testers have devised highly consistent beginning and end rules in our scenario, which ensures that the deviation of log blocks can be executed flawlessly by using a straightforward regular expression matching approach.

We will continue to utilize the examples presented in Figure 2 for illustrative purposes. The blue box marks the log block of Operation 17. Operation 17 starts with the log entry that marks the test operation is going to send a request (Begin to send request) and ends with the log entry that marks the

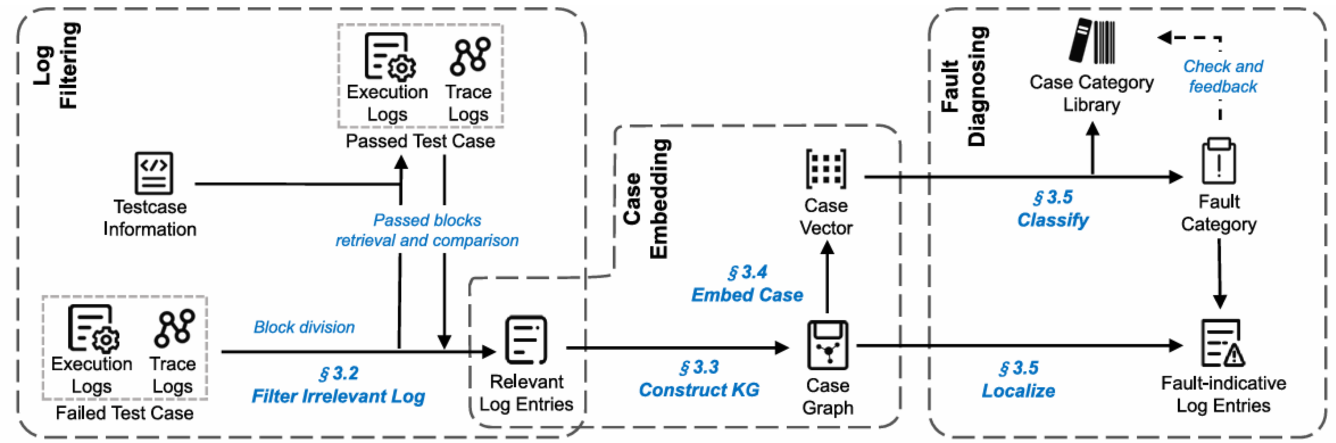


Fig. 3: **The framework of SynthoDiag**

response of this operation as unsuccessful (Check point fail). And the other logs of Operation 17 record the information of this request, such as the request object (Service III), and the request parameters (tester_factory_001, delete). Consequently, each log block can comprehensively document the details of a test operation on the client side and can be related to a group of specific trace logs on the server side.

Irrelevant log blocks removement. In this part, we want to remove the normal operation log blocks from the fault case log. Specifically, we check whether each log block has appeared in the passed cases to determine whether to delete it. This is because we have many passed test cases, all of which are operations that have successfully passed the test. The specific removement process for a log block is divided into the following three steps: Step 1: Find the same case in passed test cases. We check the testcase information of the passed cases to find the case with the same name. Since the same case usually involves the same testing operation, this step helps us narrow down the scope and quickly find the same pass log block.

Step 2: Find the same log block from the same passed case. Due to the request part (i.e., the request operation and parameters) representing the content of the operation and the response part (i.e., the checkpoint result) representing the test result, we can compare the test log block's request part and response part with the passed blocks'. If these two parts are the same, it means they are the same test operation and have the same results. Then this test block should be removed.

Naturally, if we cannot find the same passed case in Step 1:, we need to spend more time searching for the same log block in different passed cases. Furthermore, if the same log block cannot be found, we can only temporarily consider it a fault test and keep it.

Step 3: Removement. If the same passed log block is found, not only should the test block of the execution log, but the set of trace logs corresponding to the test operation should also be deleted, for the whole operation testing is successful. In this way, irrelevant log blocks both in the execution log and trace log are removed, and relevant log entries are retained.

C. Knowledge Graph Construction

SynthoDiag performs the steps of entity extraction, entity alignment and graph construction to build a knowledge graph that captures essential information of the failed test case from multi source logs. This subsection outlines the steps involved in knowledge graph construction.

At the entity extraction step, SynthoDiag extracts three types of entities from the original log entries.

- *Attribute Entity.* The attribute entity consists of structured data within the log entry, representing specific attributes of the log entry (e.g., INFO, WARN, and ERROR in the execution log entries that represent the level of the log entry or the service nameofthetracelogentry).Thus,we extract the attribute entity directly from the original log entries using regular expressions, which are predefined with the testers according to the features of logs.
- *Parameter Entity.* The parameter entity represents the vari able component within the unstructured data. Consequently, parameters contain highly specific information about each test case. SynthoDiag utilizes the Drain log parsing method [8] to extract these parameters, which can separate the template and the parameters (e.g. in Figure 2, the tester_factory_001 and delete in the execution logs and tester_factory_001, 76 and 20 in the trace logs of test case A).
- *LogEntity.* The log entity represents the unstructured original log content, which is the most salient aspect of the original log entry that testers prioritize. With the log entity, we can build the relationship between the attribute entities and the parameter entities. We extract the log entity from the original log entries by excluding structured data (e.g. In Figure 2, Request Service: Service III in the execution logs or Connection Timeout in the trace logs of test case B).

SynthoDiag takes entity alignment step to merge the parameter entities with similar semantic representations. On the one hand, the same parameter may be recorded in different formats by multi source logs, on the other hand, there are still deviations in the log parsing method when they parse similar log content into different parts, which will be regarded as different entities. To ensure the robustness of the framework, SynthoDiag converts each parameter entity into vectors using a pre-trained language model BERT [20]. In this way, the parameter entities with similar semantics can be represented with vectors that are close to each other. Then, Syntho Diag aggregates these close parameter entities together and takes the parameter entity in the center of the cluster to represent the other parameter entities.

Finally, SynthoDiag takes the graph construction step with these three types of entities. As shown in Figure 4, when building the graph of the failed test case, SynthoDiag builds the connection between the log entity and the corresponding attribute entities with the relationship at. The connection between the log entity and the extracted parameter entities can be with the relationship has. Besides, due to the feature of trace logs, SynthoDiag also builds the connection between the attribute entities, that represent the name of service, to reflect the request relationship between service with request. In this way, the log entities from the same source can be connected by attribute entities and parameter entities, and the log entities from different sources can be connected by the merged parameter entities.

This integrated approach to knowledge graph construction en ables SynthoDiag to represent and connect information from multi source logs effectively, facilitating more comprehensive classifica tion and localization.

D. *Case Embedding*

Utilizing the prior modules, we can construct a graph that focuses on the relationship between logs, and standardize the format of logs from different sources.

To facilitate a more effective comparison of similarities between different test cases, SynthoDiag converts the failed test case into a vector representation with its knowledge graph and the filtered fault relevant log entries. On the one hand, the knowledge graph of the failed test case records the relationship among log entries, enabling the distinction of various failed test cases that exhibit similar logs but differ in structure. On the other hand, the seman tic information contained in the fault relevant log entries is also essential to identifying test cases with identical failed reasons. As discussed in prior research[3], SynthoDiag adopts MRotate [10], a knowledge graph embedding algorithm that embeds the knowledge graph into a continuous space by viewing relations as rotations in the complex vector space, where it models the interaction between entities and relations, to obtain the structural representation vector for each entity in the knowledge graph. This captures structural information among diverse entities, providing an overview of the relationships between log entries. Consequently, SynthoDiag uti lizes Sentence-BERT[20], an adaptation of the pre-trained BERT model specifically tailored for sentence embedding tasks, to derive the semantic representation vector for each log entry. Following this, we get the vector for each entity by concatenating its structural representation vector and the semantic representation vector of the corresponding log entry. Finally, we aggregate all entity vectors in each test case to compute an average vector, designated as the case vector.

E. *Fault Diagnosing*

At the fault diagnosing stage, SynthoDiag classify the test cases into a specific categories using case vectors we obtained above, and then proposes an measure mechanism of fault-indicative log entries for localization.

Fault Classification. Table 1 shows four fault categories that used in SynthoDiag. Noting that many historical cases have already been marked by the testers in these four categories (as mentioned in Section 2.1). In addition, this library has the capability to continuously expand as more diagnostic outcomes are validated and feedback is incorporated, as shown in Figure 3.

In this work, SynthoDiag utilizes the k-nearest neighbors (KNN) algorithm to classify the failed test case based on the category labeled test cases, and subsequently provides the corresponding fault category with a diagnostic report with an explanation of fault classification. The K-NN algorithm represents a non-parametric, supervised learning method that determines an entity's category based on the categories of its k closest neighbors. This approach is accessible to testers due to its straightforward conceptual basis and ease of interpretation.

Fault Localization. Additionally, to facilitate more efficient root cause localization, SynthoDiag identifies the most likely fault indicative log entry within the test case.This identification is carried out by using the extracted parameter entity, which records more specific details of a test case in a more fine-grained form. We intro duce the Entity Fault Association score (EFA) for individual entities to assess their relationship with a specific fault category, along with the Position Value (PV) to gauge the entity's significance within the current test case.

Based on our observations, entities that predominantly manifest within a single fault category, with infrequent occurrences in other categories, often exhibit relevance to the primary cause of that specific fault category. Building upon this insight, we define the EFA for Entity e and fault Category i as Equation 1:

$$EFA_i^e = -mse(D_e) \cdot \frac{1}{\log(1 - \frac{n_i^e}{N_i})} \tag{1}$$

where D_e denotes the distribution of entity e in each fault category, N_i denotes the number of test cases in fault category i, and n_i^e denotes the number of test cases in fault category i where entity e appears in. A higher EFA_i^e value indicates a stronger link between entity e and fault category i.

In addition, for a parameter entity, the more log entities it relates to, the more important it is in the current use case

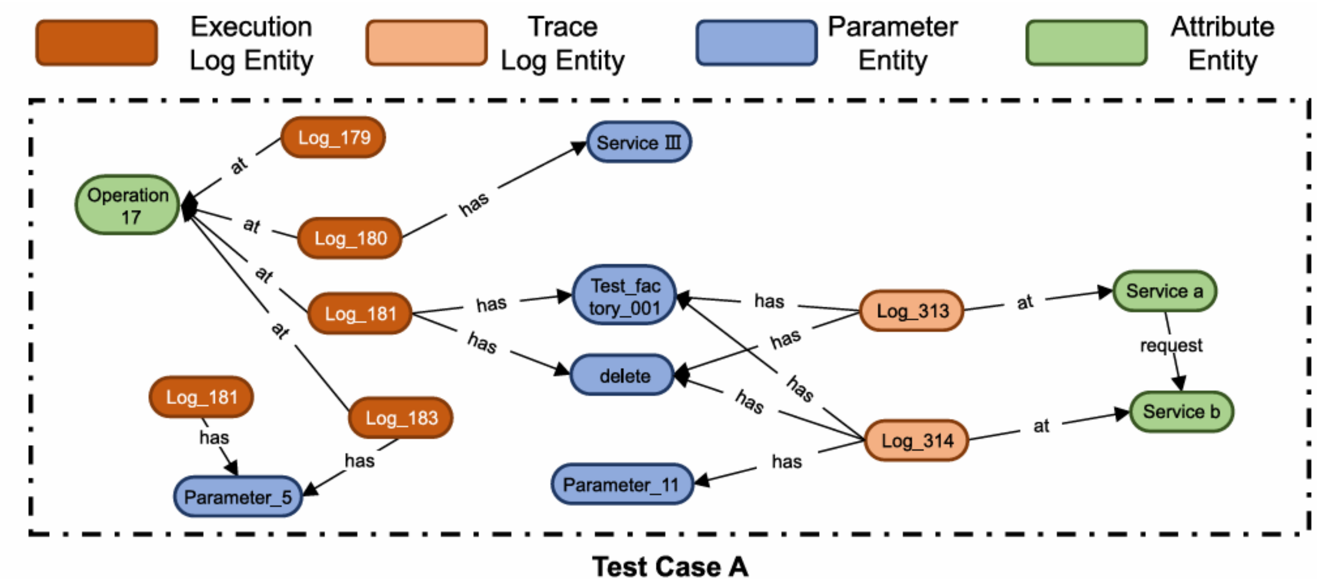


Fig. 4: **The knowledge graph of test case A.**

because this parameter entity appears in many fault-indicative log entries and can build the relationship between log entries. Thus, SynthoDiag searches the knowledge graph of the current case to calculate the PV of parameter entity e as Equation 2:

$$PV^e = (L^e_{exe} + 1) \cdot (L^e_{trace} + 1) \tag{2}$$

where L^e_{exe} is the number of connected execution log entries, and L^e_{trace} is the number of connected execution log entries. Then, we get the EFA-PV^e_i of entity e with fault category i by multiplying EFA^e_i and PV^e, as Equation 3:

$$EFA\text{-}PV^e_i = EFA^e_i \times PV^e \tag{3}$$

Next, we calculate the value score of each log entry by summing up the EFA-PV^e_i of the extracted parameter entities. Finally, we sort the log entries with their value score and localize the top-K log entries as the fault-indicative log entries.

IV. EVALUATION

In this section, we evaluate the performance of *SynthoDiag* using the datasets collected from a top-tier global Cloud service provider. We aim to answer the following research questions (RQs):

RQ1: How does *SynthoDiag* perform overall in log diagnosis compared to baseline models?

RQ2: Does each component of *SynthoDiag* have significant con tributions to SynthoDiag's performance?

RQ3: Specifically, discuss the necessity of KNN in classification and how its main hyperparameter ? affects the results. Noting that there are no other important hyperparameters to discuss in the whole framework besides ? here.

A. *Fault Diagnosing*

Dataset: We conduct experiments using a real-world Multi-source test case dataset collected from the production environment ofatop tier global Cloud service provider (***Huawei*** dataset). The Huawei dataset includes execution logs and trace logs from 1687 failed test cases. To ensure consistent classification standards, we assigned one tester to manually re-label all 1600+ failed test cases into four categories (”Service Issue”-9.0%, ”Environment Issue”-53.2%, ”Script Defect”-36.3%, ”Tool Defect”-1.5%), and take the re-labeled result as the ground truth of each test case. We adopt a cross-validation strategy to assess performance. This strategy involves dividing all failed test cases into five equal subsets and selecting one subset as the test set while using the remaining four as the training set in rotation. The final performance is determined by averaging the results from these five datasets.

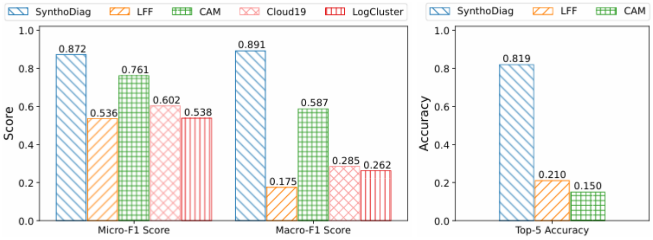
Baselines: We compare *SynthoDiag* with four fault classification algorithms: running-system classification algorithm Log Cluster (trace logs) [14], Cloud19 (trace logs) [25] and fault test classification algorithm LFF (execution logs) [1], CAM (execution logs) [11]. We also compare *SynthoDiag* with LFF and CAM for fault localization. The parameters of these methods are optimized for performance.

Evaluation Metrics: In this scenario, fault diagnosis can be viewed as a multi-classification task. To measure the performance, we take the Micro-F1 score and Macro-F1 score as the metrics [21], where the Micro-F1 score is the percentile of fault test cases classified into the correct category. And the Macro-F1 score is calculated as follows,

$$\text{Macro-F1} = \frac{\sum_{i=1}^{k} \frac{2\times\text{Recall}_i\times\text{Precision}_i}{\text{Recall}_i+\text{Precision}_i}}{k} \tag{4}$$

where k is the number of fault categories, $Recall_i$ is the percent age of fault cases with category i that are correctly classified into category i and $Precision_i$ is the percentage of fault cases classified into category i with correct category i. The Micro-F1 score reflects the overall performance on classification, and the Macro-F1 score reflects the average performance among the fault categories, which we focus more on. Besides, to measure the performance of fault localization, we use Top-5 Accuracy [19], which is the percentage of localization results that contain correct fault log entries within 5 candidates.

Experimental Setup: We conduct all the experiments on a Fusion Server G560 V5 server with Inter Xeon E5-2697 v4 CPU and 251GB of memory. We implement *SynthoDiag* and the other four baselines with Python 3.7.



(a) F1-Scores (b) Top-5 Accuracy

Fig. 5: **The effectiveness of different methods on *Huawei* dataset. F1-Score for classification and Top-5 Accuracy for localization.**

TABLE II: **Time for diagnosing one case**

	SynthoDiag	LFF	CAM	Cloud19	LogCluster
Time(s/case)	0.39	34	2.22	0.23	1.76

B. *The Overall Performance*

This section presents a comparative analysis of *SynthoDiag* and four baseline methods applied to ***Huawei*** dataset for evaluating the effectiveness of our approach.

Figure 5 presents a comprehensive comparison between Syntho Diag and the baseline methods in terms of F1-score, Top-5 accuracy, and Table 2 presents the execution time.

In summary, *SynthoDiag* outperforms all the baseline methods, with a Micro-F1 score of 0.872 and a Macro-F1 score of 0.891. These scores are 21% and 30% higher than the best baseline method, re spectively. To be specific, while CAM achieves a Micro-F1 score of 0.761, it has a lower Macro-F1 score of 0.587, and the other three baselines all perform poorly on Macro-F1 score, which shows that *SynthoDiag* has a better performance than these fault classification methods. And, *SynthoDiag* also shows the best performance on fault localization with a top-5 accuracy of 0.819, which shows that both template-based or string-based localization methods are not suitable for localizing fault log entries in failed test cases of Microservice System. While LFF and CAM can do both the classification and localization, the defects in their design make them not adapt to this scenario.

Furthermore, as shown in Table 2, *SynthoDiag* achieves prompt diagnosis of each failed test case, with a time requirement of just 0.39 seconds. This efficiency is over 100 times higher than previous manual diagnosis. According to testers' statistics, manual diagnosis of a case usually takes more than 5 minutes because it requires complex analysis, including reading the origin execution logs, re trieving the log entry that may be related to this fault, analyzing both the fault-relevant log entries and the trace logs etc.

C. *Ablation Study*

We conduct ablation experiments on ***Huawei*** dataset to evaluate the effectiveness of key components in SynthoDiag: fault-irrelevant log filtering, multi-source log fusion, case embedding, and EFA-PV.

4.3.1 Fault-Irrelevant Log Filtering. Firstly, *SynthoDiag* utilizes the block-based filtering strategy to filter out irrelevant log entries. We conduct experiments with the other two processing strategies: line-based filtering and no filtering. As depicted in Figure 6 (a), the block-based filtering strategy yields the best performance because it retains the most log entries relevant to faults. line-based filtering almost removes all the log entries and only leaves the log entries of the error message, which means that there is not enough valid information remaining to diagnose the fault. Conversely, the no filtering strategy retains all log entries, including irrelevant ones, which adversely affect the extraction of features from failed test cases.

4.3.2 Multi-source Log Fusion. In this work, we propose to use multi-source logs (execution logs from the client side and trace logs from the server side) to diagnose failed test cases. To verify the effectiveness of using multi-source logs, we only take execution logs (exe logs) and trace logs (trace logs) to build the failed test case with the same steps, and evaluate the performance on these three metrics.

As illustrated in Figure 6(b), the utilization of multi-source logs indeed aids in diagnosing the fault categories of failed test cases. While execution logs document test progress, trace logs from the server side capture finer-grained server-side details. Therefore, the amalgamation of execution logs and trace logs can enhance diagnostic performance. Service issues are responsible for only a small fraction of failed test cases; hence, the trace log contains limited valuable information.

4.3.3 Case Embedding. After building the knowledge graph of the failed test case, *SynthoDiag* embeds the case with structure and semantic vectors. We believe that this approach yields a case vector that encompasses a richer set of information about the test case. To evaluate the effectiveness of this module, we take different embedding strategies: we only use the structure vectors (w/o Bert) and the semantic vectors (w/o KGE), and we also take the traditional template-based embedding strategy on the origin logs (One-hot), which take the template of the origin log entry to represent the test case.

In Figure 6 (c), it is evident that the combination of structure representation and semantic vectors significantly enhances the per formance of *SynthoDiag*. Both w/o Bert and w/o KGE exhibit lower Micro-F1 and Macro-F1 scores in isolation. The exclusion of w/o Bert results in the neglect of crucial semantic representation within the log content, which is essential for identifying similarities among cases in the same fault category. w/o KGE cannot use the relation ship betweenlogs, whichisacrucial feature for distinguishing cases amongfault categories. *SynthoDiag* effectively combines both struc ture representation from KGE and semantic representation from Bert, allowing for a more detailed extraction of features from failed test cases. When the embedding method is replaced with one-hot, the performance deteriorates significantly, as the template-based approach fails to adapt to the varying recording styles of trace logs.

Consequently, it can be concluded that *SynthoDiag* achieves superior performance when both structure representation and semantic representation are incorporated.

4.3.4 EFA-PV. We introduce EFA-Pv as a method for identifying fault logs by computing the importance of parameter entities and arranging the original log entries based on their value scores. EFA-PV combines EFA to determine the correlation between parameter entities and fault categories, and PV to assess the significance of parameter entities within the current testcases. In contrast to EFA-PA, TF-IDF solely gauges the uniqueness of parameter entities without quantifying their relationship with fault categories. To evaluate the effectiveness of this module, we only use part of EFA-PV (EFA) and (PV) to localize the fault log entries. And we replace EFA-PV with TF-IDF to measure the importance of the log entry (TF-IDF).

As depicted in Figure6 (d), EFA-PV attains the highest accuracy, surpassing EFA and PV when used individually. This is because EFA aids inidentifying fault parameter entities, while PV gauges the significance of parameter entities in failed testcases. However, TF-IDF can solely identify unique log entries, which may not necessarily be associated with faults.

D. *Classification Algorithms*

We also discuss the effect of different classification algorithms. We conduct experiments with five different classification algorithms: *KNN (K=1), KNN (K=3), KNN (K=5), Random Forest*, Bayes, Gradient Boosting Classifier (*GBC*), Ada Boost Classifier (*ABC*). As depicted in Figure 7, both the KNN and Random Forest algorithms exhibit comparable performance. However, Random Forest is unable to provide a reasonable explanation for the output results. On the other hand, KNN can adapt to updates by incorporating the newly labeled cases into the case category library. Furthermore, with increasing values of *K*, the classification performance diminishes due to the imbalance in the fault category and the heightened interference as *K* increases.

V. IMPLEMENTATION

SynthoDiag has been implemented on the automated testing cloud platform of Huawei Cloud. As illustrated in Figure 8, this cloud platform is provided as a service (PaaS) for users to do the task of testing automatically, including test script generation, fault diagnosis, board of test, etc. When users want to access these services, they connect their test pipeline to the interface provided by the platform. The results of each test case will be uploaded and saved into the platform. When there are failed test cases, *SynthoDiag* will be called and the final analyzed failure reason and log entries of the root cause will be presented in the platform to help testers take the right remedial actions.

In the cloud platform, *SynthoDiag* works as a service for diagnosing failed test cases. When there is a test alarm during testing, the information about the test case will be sent to the *SynthoDiag*. Then *SynthoDiag* will take the steps in § 3 to preprocess data and diagnose the failed test cases. The fault category and root cause will be presented on the platform to the users with the fault logs.

Deployment effectiveness. *SynthoDiag* has been implemented over 7 months, and diagnoses thousands of fault cases per week. According to the feedback from testers,Testers have provided feedback that it achieves over 85% classification recall and over 80% localization accuracy, which is consistent with our evaluation. Meanwhile, it greatly improves the diagnostic efficiency of testers compared to manual labor before. The process of diagnosing failed test cases be comes more straightforward, rapidly and automatically for testers. Besides, they believe that *SynthoDiag* has strong generalization ability and plan to apply it to more testing scenarios.

VI. DISCUSSION

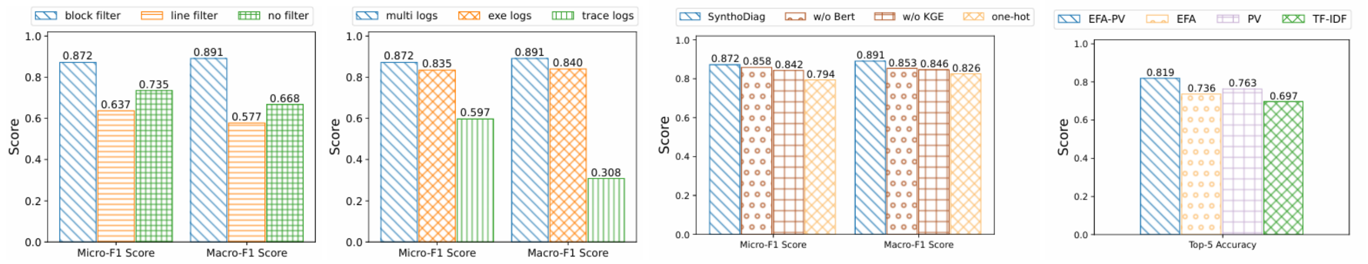
During the deployment, we also encountered some practical problems that affected the performance of the results.

A. *Data Quality*

SynthoDiag employs a knowledge graph to construct test cases by leveraging multi-source logs. The effectiveness of this construction process relies on the relationships between the various multi-source logs. A more distinct relationship between multi-source logs (e.g., shared content or explicit request parameters), enhances the effectiveness of joint analysis. In real-world testing scenarios, the relationships between multi-source logs need to be well recorded, which is a prerequisite for using multi-source logs for diagnosis. Therefore, before implementing *SynthoDiag*, it is crucial to under stand the mechanism of the tested system.

B. *Ambiguity of Fault Category*

SynthoDiag utilizes the KNN approach to identify the most similar historical test case and assigns the fault category of the matched test case to the current test case. Consequently, the accuracy of the historical fault category label directly impacts the effectiveness of categorization. Nevertheless, in real-world scenarios, there are often over a dozen testers responsible for analyzing system failures, and the criteria for fault categorization may differ among them. As a result, identical fault test cases may receive different categorizations, leading to



(a) Filtering Strategy (b) Source of Logs (c) Case Embedding Strategy (d) Localization Strategy

Fig. 6: **The effectiveness of each module**

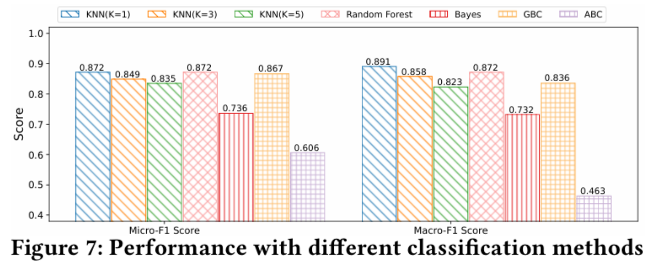


Fig. 7: **Performance with different classification methods**

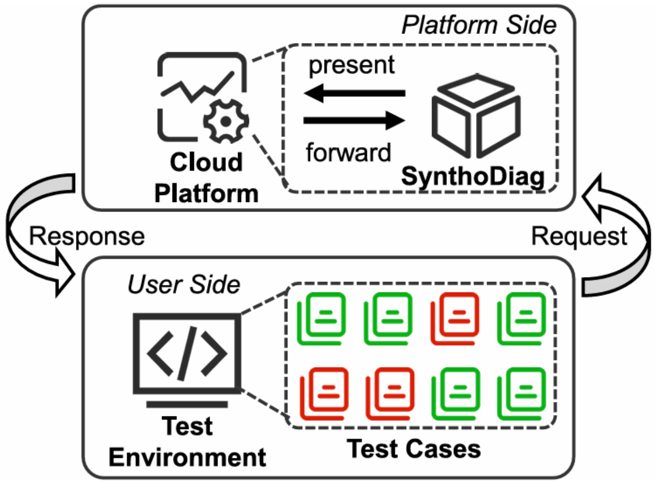


Fig. 8: **The deployment of SynthoDiag on the cloud platform.**

ambiguity that can hinder the accurate categorization of faults by the diagnostic model. This is a recurring issue in situations involving numerous participants. In our future work, we will explore a cost-effective method for label adjustment.

VII. Related Work

Numerous research approaches are dedicated to addressing the challenges of diagnosing or localizing faults, which focus on iden tifying faults and localizing fault information respectively. In our work, we address both of these critical tasks, with the overarching goal of reducing the workload on the tester.

A. *Fault Diagnosis*

Diagnosing before manually analyzing can help the tester comprehend faults. Lu et al. [16] classifies Spark faults with four distinct categories. Cloud19 [25] employs log associations with system tasks to classify faults within cloud systems. It's worth noting that these studies conducted their analyses within run-time environments, a context that doesn't precisely align with our specific scenario. Kim et al. [9] use association rule learning to identify false test alarms. CAM [11] exploits information retrieval techniques to classify test faults. Log-FaultFlagger (LFF) [1] distinguishes faults caused by product problems from all test faults by ameliorating CAM [11].

B. *Fault Localization*

The testers need to find the information about the fault to solve problems faster. However, as software systems have expanded, the volume of logs has surged, rendering manual log inspection an exceedingly time-consuming endeavor.

MicroRank[24] combines anomaly detection with PageRank scoring and extended spectrum analysis to effectively identify and rank potential root causes of latency issues in microservice systems. MicroHECL [15] identifies root causes by constructing a dynamic relation graph and employing finely tuned rules. AutoMap [17] employs a modified PCalgorithm to construct an anomaly behavior graph and propose potential root causes. LogCluster [14] utilizes clustering techniques to decrease the number of logs that should be examined. Onion [27] provides debugging clues by locating incident-indicating logs. However, it is important to highlight that these methods may not align with our specific scenario, as they are primarily designed for runtime microservice systems and offer debugging insights at the server level. While LFF [1], CAM [11], and Historian [13] use information retrieval techniques to flag fault log entries, they can not take advantage of multi-source logs and adapt the the demand of microservice systems.

VIII. Conclusion

After careful investigations on thousands of failed test cases col lected from a top-tier global cloud service provider, Huawei Cloud, which provides services for millions of users, we identify the im portance of fusing multi-source logs for test case fault diagnosis in microservices. Consequently, we propose a framework, SynthoDiag, for diagnosing failed test cases by fusing the information from execution logs, test information, and trace logs with the knowledge graph technique. We present a log-block-based differentiation ap proach to reduce the impact of fault-irrelevant log entries. To tackle the challenge introduced by the newly generated log templates, we introduce EFA-PV, which can effectively represent the relevance of a log entry to a given fault. Finally, we conduct comprehensive ex periments on a real-world test case dataset collected from Huawei Cloud, demonstrating that the proposed SynthoDiag outperforms all state-of-the-art methods. SynthoDiag takes the first step towards fusing multi-source logs for diagnosing failed test cases in microser vices. In the future, we will evaluate SynthoDiag in more scenarios to demonstrate its performance.

Acknowledgment

This work is supported by the Advanced Research Project of China (No. 31511010501), and the National Natural Science Foundation of China (62272249, 62302244, 62072264).

References

[1] Anunay Amar and Peter C Rigby. 2019. Mining historical test logs to predict bugs and software faults in the test logs. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 140–151.

[2] Zhichao Chen, Junjie Chen, Weijing Wang, Jianyi Zhou, Meng Wang, Xiang Chen, Shan Zhou, and Jianmin Wang. 2023. Exploring better black-box test case prioritization via log analysis. ACM Transactions on Software Engineering and Methodology 32, 3 (2023), 1–32.

[3] Yuanfei Dai, Shiping Wang, Neal N Xiong, and Wenzhong Guo. 2020. A survey on knowledge graph embedding: Approaches, applications and benchmarks. Electronics 9, 5 (2020), 750.

[4] Cheng Deng, Yuting Jia, Hui Xu, Chong Zhang, Jingyao Tang, Luoyi Fu, Weinan Zhang, Haisong Zhang, Xinbing Wang, and Chenghu Zhou. 2021. Gakg: A multimodal geoscience academic knowledge graph. In Proceedings of the 30th ACM International Conference on Information & Knowledge Management. 4445–4454.

[5] Wojciech Dobrowolski, Maciej Nikodem, and Olgierd Unold. 2023. Software Failure Log Analysis of Engines. Electronics 12, 10 (2023), 2260.

[6] Luca Gazzola, Maayan Goldstein, Leonardo Mariani, Marco Mobilio, Itai Segall, Alessandro Tundo, and Luca Ussi. 2023. ExVivoMicroTest: ExVivo testing of microserivces. Journal of Software: Evolution and Process 35, 4 (2023), e2452.

[7] Israr Ghani, Wan MN Wan-Kadir, Ahmad Mustafa, and Muhammad Imran Babir. 2019. Microservice testing approaches: A systematic literature review. International Journal of Integrated Engineering 11, 8 (2019), 65–80.

[8] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In 2017 IEEE international conference on web services (ICWS). IEEE, 33–40.

[9] Kim Herzig and Nachiappan Nagappan. 2015. Empirically detecting false test alarms using association rules. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, 39–48.

[10] Xuqian Huang, Jiuying Tang, Zhen Tan, Weixin Zeng, Ji Wang, and Xiang Zhao. 2021. Knowledge graph embedding by relational and entity rotation. Knowledge-Based Systems 229 (2021), 107310.

[11] Ba Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. 2017. What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 712–723.

[12] Amar Viswanathan Kannan, Dmitriy Radkin, Ioannis Akrotirianakis, Tugba Kulahcioglu, Arquimedes Canedo, Aditi Fadke, John-Yuan Yu, Malawade Arnav, and Muhammad Abdullah Al Faruque. 2020. Multimodal knowledge graph for deep learning papers and code. In Proceedings of the 29th ACM International Conference on Information and Knowledge Management. 3417–3420.

[13] Jinhan Kim, Valeriy Savchenko, Kihyuck Shin, Konstantin Sorokin, Hyunseok Jeon, Georgiy Pankratenko, Sergey Markov, and Chul-Joo Kim. 2020. Automatic abnormal log detection by analyzing log history for providing debugging insight. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice. 71–80.

[14] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In Proceedings of the 38th International Conference on Software Engineering Companion, 102-111.

[15] Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshen Wu. 2021. Microhel: High-efficient root cause localization in large-scale microservice systems. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 338–347.

[16] Siyang Lu, BingBing Rao, Xiang Wei, Byungchul Tak, Long Wang, and Liqiang Wang. 2017. Log-based abnormal task detection and root cause analysis for spark. In 2017 IEEE International Conference on Web Services (ICWS). IEEE, 389–396.

[17] Meng Ma, Jingmin Xu, Yuan Wang, Pengfei Chen, Zonghua Zhang, and Ping Wang. 2020. Autoproc: Diagnose your microservice-based web applications automatically. In Proceedings of The Web Conference 2020. 246–258.

[18] Shang-Pin Ma, Chen-Yuan Fan, Yen Chuang, Wen-Tin Lee, Shin-Jie Lee, and Nien-Lin Hsueh. 2018. Using service dependency graph to analyze and test microservices. In 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Vol. 2. IEEE, 81–86.

[19] Suman Ravuri and Oriol Vinyals. 2019. Classification accuracy score for conditional generative models. Advances in neural information processing systems 32 (2019).

[20] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. arXiv preprint arXiv:1908.10084 (2019).

[21] Yichen Sui, Yuzhe Zhang, Jianjun Sun, Ting Xu, Shenglin Zhang, Zhengdan Li, Yongqian Sun, Fangrui Guo, Junyu Shen, Yuzhi Zhang, et al. 2023. LogKG: Log Failure Diagnosis through Knowledge Graph. IEEE Transactions on Services Computing (2023).

[22] Muhammad Waseem, Peng Liang, Mojtaba Shahn, Amleto Di Salle, and Gastón Márquez. 2021. Design, monitoring, and testing of microservices systems: The practitioners' perspective. Journal of Systems and Software 182 (2021), 111061.

[23] Zhe Xie, Haowen Xu, Wenxiao Chen, Wanxue Li, Hua Jiang, Liangfei Su, Hanzhang Wang, and Dan Pei. 2023. Unsupervised Anomaly Detection on Microservice Traces through Graph VAE. In Proceedings of the ACM Web Conference 2023. 2874–2884.

[24] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. Microrain: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In Proceedings of the Web Conference 2021. 3087–3098.

[25] Yue Yuan, Wenchang Shi, Bin Liang, and Bo Qin. 2019. An approach to cloud execution failure diagnosis based on exception logs. In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). IEEE, 124–131.

[26] Shenglin Zhang, Pengxiang Jin, Zihan Lin, Yongqian Sun, Bicheng Zhang, Sibo Xia, Zhengdan Li, Zhenyu Zhou, Minghua Ma, Wu Jin, et al. 2023. Robust Failure Diagnosis of Microservice System through Multimodal Data. arXiv preprint arXiv:2302.10512 (2023).

[27] Xu Zhang, Yong Xu, Si Qin, Shilin He, Bo Qiao, Ze Li, Hongyu Zhang, Xukun Li, Yngnong Dang, Qingwei Lin, et al. 2021. Onion: identifying incident-indicating logs for cloud systems. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 1253–1263.