Easy over Hard: A Simple Baseline for Test Failures Causes Prediction

Zhipeng Gao
Zhejiang University
China
zhipeng.gao@zju.edu.cn

Zhipeng Xue
Zhejiang University
China
zhipengxue@zju.edu.cn

Xing Hu*
Zhejiang University
China
xinghu@zju.edu.cn

Weiyi Shang
University of Waterloo
Canada
wshang@uwaterloo.ca

Xin Xia
Huawei
China
xin.xia@acm.org

ABSTRACT

The test failure causes analysis is critical since it determines the subsequent way of handling different types of bugs, which is the prerequisite to get the bugs properly analyzed and fixed. After a test case fails, software testers have to inspect the test execution logs line by line to identify its root cause. However, manual root cause determination is often tedious and time-consuming, which can cost 30-40% of the time needed to fix a problem. Therefore, there is a need for automatically predicting the test failure causes to lighten the burden of software testers. In this paper, we present a simple but hard-to-beat approach, named NCChecker (Naive Failure Cause Checker), to automatically identify the failure causes for failed test logs. Our approach can help developers efficiently identify the test failure causes, and flag the most probable log lines of indicating the root causes for investigation. Our approach has three main stages: log abstraction, lookup table construction, and failure causes prediction. We first perform log abstraction to parse the unstructured log messages into structured log events. NCChecker then automatically maintains and updates a lookup table via employing our heuristic rules, which record the matching score between different log events and test failure causes. When it comes to the failure cause prediction stage, for a newly generated failed test log, NCChecker can easily infer its failed reason by checking out the associated log events' scores from the lookup table. We have developed a prototype and evaluated our tool on a real-world industrial dataset with more than 10K test logs. The extensive experiments show the promising performance of our model over a set of benchmarks. Moreover, our approach is highly efficient and memory-saving, and can successfully handle the data imbalance problem. Considering the effectiveness and simplicity of our approach, we recommend relevant practitioners to adopt our approach as a baseline to beat in the future.

*This is the corresponding author

FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663850

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Log Analysis, Test Failure Causes, Test Logs, Root Causes Analysis

ACM Reference Format:
Zhipeng Gao, Zhipeng Xue, Xing Hu, Weiyi Shang, and Xin Xia. 2024. Easy over Hard: A Simple Baseline for Test Failures Causes Prediction. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3663529.3663850

1 INTRODUCTION

As modern software has become much larger and more complex, software defects and bugs are unavoidable in such systems. These software defects can lead to the system failures and its degraded quality (e.g., performance, reliability and/or security) [14, 28, 32, 34, 44]. To minimize the number of delivered errors and mitigate the risk of system failures, developers, and/or testers usually resort to software testing by running test cases. The risk of software failures may be considered low with passing all test cases. By contrast, the risk of software failures is significantly high if the test case fails. Once a test case fails, it is necessary for developers and/or testers to further investigate the reasons for its failure by analyzing the test execution results, which are typically stored in log files.

Developers use logs to record valuable runtime information (e.g., important events, program variables values, trace execution, runtime statistics, and even human-readable messages). The rich and detailed information recorded in log files are considered as the most important and useful resources to help developers and/or testers understand system failures and identify potential failure causes [2, 9, 23, 29, 33, 42]. Moreover, because logs are often the only available data that reports the software runtime information, logs are referred to as the most common accessible resources for diagnosing system failures.

There are many causes that can lead to test failures (e.g., environmental condition problem, source code problem, and software version problem). Different types of failures have their own corresponding action to perform (e.g., submitting bug reports to developers, rerunning test scripts, submitting exception messages to

software maintainers). Therefore, it is essential to identify the failure causes in a timely manner such that the corresponding experts can be assigned shortly in order to review, analyze and fix the bugs without further delay. On the other hand, the modern software systems grow rapidly and become more mature. Different software, hardware and services are tightly integrated, leading to ever higher difficulty in diagnosing test failures. Prior work reports that more than 100 billion US dollars has been spent on failure diagnosis process and manual determination of a test failure root causes can consume half of the total time for fixing a software issue [43, 44]. In practice, it is thus preferable to have toolkits that can automatically diagnose the cause of test failures. However, making such a tool is difficult due to the following challenges:

(1) ***Dealing with information overload.*** Information overload is a common challenge for different software engineering tasks [10, 11, 22, 31, 37, 39, 40]. To find out the causes of the test failures, software testers have to read and digest the test logs carefully. However, these test logs are often too large to examine manually. In a large software company, thousands of test failures are reported daily resulting into a huge amount of logs, with each log containing hundreds of test steps and thousands of log lines [2, 17, 38]. For example, in our study, we collected more than 10K logs from our industrial partner. A test log file consists of 3-4K log lines on average and the largest log file contains more than 550K log lines. This huge amount of information goes far beyond the level that testers can handle, and it is extremely difficult and inefficient for testers to manually figure out the failure causes [8, 26]. To alleviate such problems, practitioners have crafted extensive regular expressions to analyze test failure causes. However, the complexity of the runtime behaviors during testing makes the definition and maintenance of such regular expressions a time-consuming and error-prone task; while the performance is still far from satisfactory [16, 47].
(2) ***Dealing with imbalanced datasets.*** As mentioned above, the test failures are caused by various number of reasons. However, not all failure causes happen equally. For example, in our study, the vast majority of failure causes are bug-related issues (i.e., 63%) and environmental problems (i.e., 23%), the third-party library issue only accounts for a very small proportion of all failure cases (i.e., 1.4%). Despite previous study [18] has proposed a similarity-based approach to predict the multiple failure causes, the performance of the similarity-based approaches will decrease dramatically on such a highly imbalanced dataset, especially for these minority failure classes. It is thus valuable to have an approach that focuses on few-shot samples and prevents the majority of samples from excessively affecting the learning process.
(3) ***Dealing with the rapidly increased latency and memory.*** Considering the large-scale software systems run on 24x7 basis, the generated logs are typically huge [15, 28]. Analyzing the archived logs in such a huge volume brings challenge to the latency and memory usages. For example, the approaches proposed by previous studies [2] need to compare logs against a library of historical referenced logs to identify the test failure causes. If the library increases with the velocity of the rapidly accumulated logs (e.g., 50 gigabytes per hour in Google system), simply reading these logs into memory for comparison purposes and retrieving relevant logs can cost significant time. Therefore, dealing with the rapidly increased logs and ever-increasing computation resources is a major challenge.

To overcome the aforementioned challenges from practice, in this work, we propose NCChecker (**N**aive Failure **C**ause **Checker**), a heuristic rule-based approach for failure causes analysis that learns from the large volumes of test logs. NCChecker is simple and contains three stages: log abstraction, lookup table construction and failure causes prediction. In the first stage, we perform log abstraction to convert each unstructured log into structured log events. The structured information contains essential of log lines without any noisy details and can then be used as input for the downstream tasks. In the second stage, we create the failure reason lookup table by using four simple but effective heuristic rules. The rows of the lookup table are different log events abstracted from the first stage, while the columns of the lookup table are different failure causes (i.e., environmental issues, bug related issues, test script issues, and third-party library issues). The cell of the lookup table contains the relevant scores estimated by NCChecker by using our heuristic rules. For a given log event and a failure reason, the higher the score, the more relevant the log event is associated with the particular failure reason. When it comes to the last stage of failure causes prediction, for a newly reported failing test case, we parse the test log into a sequence of existing log events. Afterwards, for each log event, we check out the scores from the above lookup table for different failure reasons. For a given failure reason, we sum up the checked out scores from all the possible log events for this specific failure reason. A failure reason with the maximum value will be selected as the final prediction failure cause.

To evaluate NCChecker, we collect more than 10K test logs from our industrial partner, which is a leading information and communication technology company. The experimental results show that NCChecker outperforms several state-of-the-art approaches by a large margin. Moreover, NCChecker performs well with respect to the imbalanced dataset and can successfully identify the failure causes in minority. In addition, NCChecker is efficient and consumes low memory for log analysis. In summary, this study makes the following contributions:

(1) We propose a simple but hard-to-beat approach to address the challenges of test failure analysis. NCChecker can assist developers and/or testers to correctly and efficiently diagnose test failures.
(2) We construct a dataset with more than 10K test logs to evaluate and verify the effectiveness of our model. The dateset involves more than 7K failed logs and 3K passed logs in total. The failure causes of these test logs are manually verified by testers.
(3) We conduct comprehensive experiments to investigate the effectiveness of our approach. The experimental results show that our approach is effective and efficient.
(4) Considering the effectiveness and simplicity of our approach, we recommend developers to apply our approach in practice

Table 1: An Overview of the Collected Log Datasets

Log Type	Measurement	Value
Failed Logs	# Logs	7,159
	Avg. Log Lines	3,905
	Max. Log Lines	550,732
	Total File size	3.2G
Passed Logs	# Logs	3,286
	Avg. Log Lines	4,564
	Max. Log Lines	270,108
	Total File size	1.7G

and researchers to adopt our approach as a baseline to beat in the future.

2 PRELIMINARY

In this section, we first present an overview of our log datasets. Afterward, we introduce four key insights that guide the design of the heuristic rules of our approach.

2.1 Data Overview

In this study, we collected 10,445 test logs (including 7,159 failed test logs and 3,286 passed test logs) from our industry partner. We counted the log lines and file size of each test log, and the overall data statistics of the dataset are summarized in Table 1. In our work, each failed test log is manually labeled with a specific test failure cause. There are four types of failure causes with respect to our collected test logs. Different types of failure causes are expected to be handled by different kinds of solutions. We now describe the details of different failure causes as follows:

- **Bug related issues (C1):** The bug related issues are concerned with general software system bugs due to coding mistakes, compatibility problems, and security vulnerabilities etc,. When the bug related issues occur, it is necessary to notify the software developers to identify, reproduce and fix the corresponding bugs.
- **Environmental issues (C2):** The environmental issues are related to the problems of the network, CPU, memory, operating system, etc. When the environmental issues occur, software testers are responsible to diagnose the system environment.
- **Test script issues (C3):** The test script issues related to the defects within the test scripts (e.g., expressions, arguments, statements). When test script issues occur, software testers are responsible to diagnose and debug the test scripts.
- **Third party library issues (C4):** The third-party library issues are associated with defects or incompatible problems in the third-party libraries, e.g., there are problems regarding the automatic logging system. When third-party library problems appear, it is necessary to ask developers to diagnose the third-party library software.

The distribution of the above four types of test failure causes are summarized in Table 2. From the table, we can see that there is an unequal distribution of different failure causes among test

Table 2: Different Types of Failure Causes

ID	Failure Causes	Count	Percentage
C1	**bug related issues**	4, 559	63.7%
C2	**Environmental issues**	1, 664	23.2%
C3	**Test script issues**	835	11.7%
C4	**Third party library issues**	101	1.4%
Sum	-	7, 159	100.0%

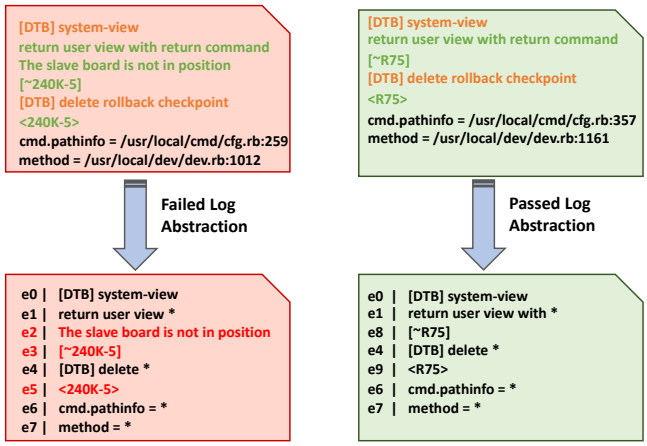


Figure 1: Log abstraction of failed/passed logs

logs. For example, the vast majority of the failure causes are the **C1** (*bug related issues*) and **C2** (*environmental issues*), which make up a very large proportion (i.e., over 86%) of all the failed test cases. The number of failed tests caused by **C3** (test script issues) and **C4** (third-party library issues) only account for a relatively small number of the failed test cases. Especially regarding **C4**, only 101 test failures are caused by the *third-party library issues*, which comprises only 1.4% among all failed test cases.

2.2 Key Insights

Our approach has been inspired by the following four key insights, which lead to our solution for this task.

Key Insight 1: Logs are often too large and too unstructured to analyze manually. First of all, the log files are often very large. For example, as shown in Table 1, the failed test log contains 3,905 log lines on average, with the largest log file containing over 550K log lines. Testers have to go through the entire log file to identify the log lines that correspond to the test failure. The sheer amount of log data makes its analysis a time-consuming and challenging task. Moreover, the log files are highly complex and unstructured. We present two snippets of failed/passed logs in Fig. 1. As shown in Fig. 1, a test log often involve a sequence of test steps (e.g., *system-view* and *delete rollback checkpoint* shown in orange text), each test step is followed by multiple lines of echo messages, which contains the state of the object (e.g., ⟨*240K-5*⟩ and ⟨*R75*⟩), environment variables (e.g., *cmd.pathinfo*), exception messages (e.g., *The slave board is not in position*), etc.

To overcome this challenge, we adopt a log abstraction technique in our approach. Log abstraction can significantly reduce the number of unique lines in a log. The raw log lines often contain dynamic run specific information (e.g., file path, IP address) that can hinder the automatic analysis, log abstraction can extract the log event by abstracting away dynamic parameters. For example, in Fig. 1, the second last log line of the failed log ("`cmd.pathinfo=/usr/local/cmd/cfg.rb:259`") and the passed log ("`cmd.pathinfo=/usr/local/cmd/cfg.rb:357`") can be abstracted with the same log event $\mathbf{e_6}$ (`cmd.pathinfo=*`). After removing the superfluous information, the abstract log line only contains the essence of each log line without noisy details. As a result, each log will be abstracted a sequence of log events which record the system status and operations. For example, the failed log snippet is abstracted into eight log events (i.e., $\mathbf{e_0}$ - $\mathbf{e_7}$) after the log abstraction process.

Key Insight 2: The failed log events are more important than the passed log events for identifying failures causes. We make the following observations: test failure locations should be contained in the lines of a failed log. On the contrary, a passed log should not contain the lines related to a failure. Therefore, log lines that appear in both failed test logs and passed test logs are unlikely to link to a fault. For a failed test log, most of the log lines record normal operations, only a small percentage of log lines are problematic and indicate problems. For example, as shown in Fig. 1, the failed log contains a log event $\mathbf{e_1}$ (i.e., `return user view *`), However, the passed log also contains this log event, so it is unlikely that the test failure is related to the log event $\mathbf{e_1}$. In contrast, the log event $\mathbf{e_2}$ (`the slave board is not in position`) occurs only in the failed log, indicating the potential cause for this failure. Based on our assumption, we should focus on the log events that are only covered by failed test logs.

Key Insight 3: The single-problem log events are more important than the multi-problem log events for identifying failure causes. As mentioned above, a log event can occur in both failed test logs and passed test logs. Similarly, a log event can be relevant to a single failure cause or multiple failure causes. Based on the number of failure causes, we further divide the failed log event into single-problem log events and multi-problem log events. That is, if a log event is only relevant to one particular type of failure cause, we then consider this log event as a single-problem log event. Otherwise, if a log event is relevant to two or more types of failure causes, we consider this log event as a multi-problem log event. For example, as shown in Fig. 2, the log event $\mathbf{e_2}$ is associated with four types of failures (i.e., C1-C4), which is a multi-problem log event. The log event $\mathbf{e_3}$ is only relevant to C2 (*environmental issues*) in history, which is a single-problem log event. In this study, we assume that multi-problem log events are less helpful in identifying the root cause of a specific test failure. In contrast, the single-problem log event that only happens with respect to a particular failure, is more likely to be useful in failure causes prediction.

Key Insight 4: The minority-class log events are more important than the majority-class log events for identifying failure causes. Our last insight comes from the challenge of the imbalanced dataset. As discussed in Section 2, each failed test log is manually labeled with a failure cause verified by testers. However, most of the failed logs are associated with the C1 (*bug related issues*) and C2 (*environmental issues*) (more than 86%). Only a small percentage of failed logs are caused by C3 (*test script issues*) and C4 (*third party library issues*). In this study, regarding the single-problem log events, if the log event is associated with C1 or C2, we consider this log event as a majority-problem log event; if the log event is associated with C3 or C4, we consider this log event as a minority-problem log event. For example, as shown in Fig. 2, the log event $\mathbf{e_3}$ is a majority-problem log event (associated with C1 only), while the log event $\mathbf{e_8}$ is a minority-problem log event (associated with C4 only). When predicting the potential failures, even though $\mathbf{e_3}$ and $\mathbf{e_8}$ both occur twice in history, considering the C4 (*third party library issues*) is comparatively rare than the C1 (*bug related issues*), the minority-problem event log $\mathbf{e_8}$ clearly has greater predictive power for identifying failures causes.

3 OUR APPROACH

We first define the task of identifying test failure causes for our study. We then present the details of our proposed approach. The overall framework of our approach is illustrated in Fig. 2.

3.1 Task Definition

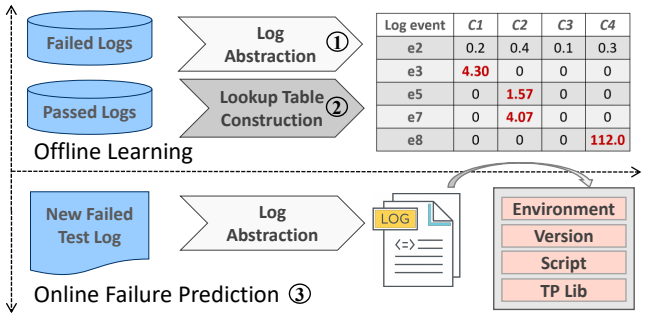
The goal of our work is to automatically predict the failure cause of a test based on analyzing the test logs. In particular, given the historical passed logs $\mathbf{P}$, the historical failed logs $\mathbf{F}$ and the associated failure causes $\mathbf{C}$, the failure cause prediction problem is to predict $\mathbf{C_{new}}$ for the newly arisen test failures via analyzing the unseen failed log $\mathbf{F_{new}}$ with the help of $\langle \mathbf{P}, \mathbf{F}, \mathbf{C} \rangle$. We formulate this task as a multi-class classification problem. More formally, our task is to find a function `Predict` so that:

$$\mathbf{Predict}(\mathbf{F_{new}} | \langle \mathbf{P}, \mathbf{F}, \mathbf{C} \rangle) = \mathbf{C_{new}} \tag{1}$$

3.2 Approach Details

In this paper, we propose NCChecker, whose overall framework is presented in Fig. 2. NCChecker consists of three stages: log abstraction, lookup table construction and failure causes prediction. In short, the unstructured raw logs are parsed into a sequence of log events by log abstraction. Then based on our previous insights and observations, we propose heuristic rules to make a lookup table. The lookup table records the predictive power scores for different log events regarding different failure causes. Finally, during the failure cause prediction stage, for a newly failed test log, NCChecker can easily infer the root cause by checking from the lookup table. More details are presented in the following sub-sections.

3.2.1 Log Abstraction. As discussed as **Key Insight 1**, the raw logs are too large and unstructured to analyze manually. To overcome the unstructured nature of test logs and reduce the large amount of log messages, we first use log abstraction techniques to parse unstructured, free-text log messages into structured log events. The common way of log abstraction in industry is to write regular expression according to the logging statements in the source code. However, due to the rapidly growing log data and frequently software updates, it is too costly and time-consuming to manually construct regex. Therefore, automatic log parsing without source code is necessary. Previous studies have proposed different log abstraction techniques [5, 7, 12, 25, 27], in this study, we adopt the widely used log parser, Drain [13], for our log abstraction tasks.



Log event	C1	C2	C3	C4
e2	0.2	0.4	0.1	0.3
e3	4.30	0	0	0
e5	0	1.57	0	0
e7	0	4.07	0	0
e8	0	0	0	112.0

Figure 2: Workflow of our approach

In general, each log line contains two parts: the static part and the dynamic part. The static part describe the semantic meaning of a program event, while the dynamic part includes parameters (e.g., file name) that record system attributes. The goal of automatic log abstraction is to separate the static part from the dynamic part. In particular, the log parser can abstract a group of similar log messages into a unique log event, and mask the dynamic part with a placeholder (usually by an asterisk). For example, by applying the Drain log parser, the log message *"Took 10 seconds to build instances"* can be abstracted with the log event "`Took * seconds to build instances`", where the static part represents the common part of similar log messages and the asterisk represents the dynamic part.

After log abstraction, each log line is transformed into a unique log event, and a log file is transformed into a sequence of log events. As a result, for a given passed log $\mathbf{p}$, it will be parsed into a sequence of log events, i.e., $\mathbf{E_p} = \{\mathbf{e}_{p_1}, \mathbf{e}_{p_2}, ..., \mathbf{e}_{p_m}\}$. Similarly, a failed log $\mathbf{f}$ will be parsed into $\mathbf{E_f} = \{\mathbf{e}_{f_1}, \mathbf{e}_{f_2}, ..., \mathbf{e}_{f_n}\}$. For example, as shown in Fig 1, the failed log snippet is sequentially represented by its corresponding log events sequence $\{\mathbf{e_0}, \mathbf{e_1}, ..., \mathbf{e_7}\}$.

3.2.2 Lookup Table Constructing. After parsing all logs (including passed logs and failed logs) into log events, in this step, we construct a lookup table for representing the predictive power of each log event with respect to different failure causes. The lookup table is constructed by the following steps:

1. Diff with pass. Log events that both occur in passed log and failed log are unlikely to reveal a fault, which may introduce noise for failure causes prediction. Inspired by our **Key Insight 2**, we assume that the failed log events are more important than the passed log events for identifying failure causes. After log abstraction, each passed log is parsed into a set of log events, we collect all log events occur in passed logs to construct a passed log events pool, $\mathcal{P}$. Similarly, we collect all the log events occur in failed logs to construct a failed log events pool, $\mathcal{F}$. In this step, we perform *Diffwithpass* operation to remove all log events that appear in passed logs from failed logs. Specifically, for each log event in $\mathcal{F}$, if it also appears in $\mathcal{P}$, we then remove this log event from $\mathcal{F}$. After performing the *Diffwithpass* operation, all the log events in $\mathcal{F}$ occur only in failed logs, denoted as $\mathcal{F}'$. For example, as shown in Fig. 1, only the log events $\mathbf{e_2}$, $\mathbf{e_3}$, $\mathbf{e_5}$ (highlighted in red color) are retained for subsequent log analysis, while other log events (e.g., $\mathbf{e_0}$, $\mathbf{e_1}$) are removed.

2. Lookup table initialization. In this step, for each log event in $\mathcal{F}'$, we count its failed times with respect to different failure causes. Specifically, we initialize a lookup table $\mathbb{T}$, where the rows are log events in $\mathcal{F}'$, the columns are different failure causes $\mathbf{C_i}(1 \le i \le 4)$. The cell value c_{ij} represents the number of times that the log event e_i failed according to the failure cause $\mathbf{C_j}$.

In particular, the lookup table $\mathbb{T}$ initialization is conducted as follows: First, all the cell values in table $\mathbb{T}$ are initialized to zero. Given the failed test log $\mathbf{f_i}$ and its failure cause $\mathbf{C_i}$, for each log event $\mathbf{e_k}$ in $\mathcal{F}'$, if $\mathbf{e_k}$ occurs in $\mathbf{f_i}$, we then increment the count by 1 regarding the log event $\mathbf{e_k}$ and the failure cause $\mathbf{C_i}$. After the table initialization, the lookup table $\mathbb{T}$ records the failed frequency of all log events in $\mathcal{F}'$ regarding different failure causes. For example, as shown in Fig. 3, the log event $\mathbf{e_2}$ occurs 10 times in total, two of which are associated with *bug related issues* (C1), four times with *environmental issues* (C2), once with *test script issues* (C3) and three times with *third party library issues* (C4). The log event $\mathbf{e_{10}}$ occurs five times in total, all of which are associated with *environmental issues* (C2).

3. Lookup table updating with single/multi-problem log events. Inspired by our **Key Insight 3**, we assume the single-problem log events are more important than the multi-problem log events. Therefore in this step, we would like to increase the predictive power of single-problem log events and decrease the predictive power of multi-problem log events. To do this, we first normalize the multi-problem log event frequency values to the range of $(0, 1)$. In particular, regarding the multi-problem log event $\mathbf{e_i}$, the updated cell value c'_{ij} is calculated as follows:

$$c'_{ij} = \frac{c_{ij}}{\sum_{j=0}^{4} c_{ij}} \tag{2}$$

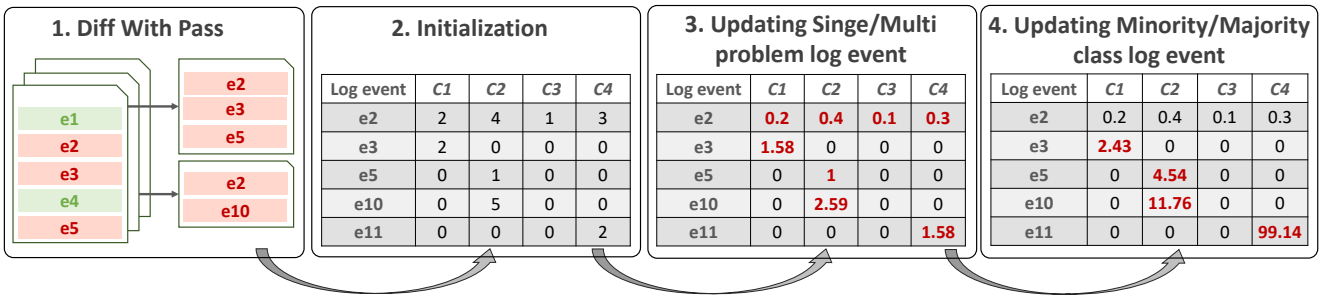
For example, the log event $\mathbf{e_2}$ is associated with several failure causes, which is a multi-problem log event. After updating the multi-problem log event, the original count value of e_2 (i.e., $[2, 4, 1, 3]$) will be normalized as $[0.2, 0.4, 0.1, 0.3]$, which represents the 20% of e_2 contributes to C1.

In contrast, regarding the single-problem log event $\mathbf{e_k}$, the updated cell value c'_{kj} is calculated as follows:

$$c'_{kj} = \begin{cases} 0.0 & \text{if} \quad c_{kj} = 0 \\ 1.0 & \text{if} \quad c_{kj} = 1 \\ log_2(1 + c_{kj}) & \text{if} \quad c_{kj} > 1 \end{cases} \tag{3}$$

For example, the log event $\mathbf{e_3}$ and $\mathbf{e_5}$ are only associated with C2, which are single-problem log events. The original count value will be updated.

4. Lookup table updating with minority/majority-class log events. Inspired by our **key insight 4**, we assume the minority-class log events are more important than the majority-class log events. Therefore we aim to increase the predictive power of the minority-class log events, and decrease the predictive power of the majority-class log events. To do this, we give different weights to different log events for different log events based on whether they belong to the majority or the minority classes. In particular, we use the ICF (inverse class frequency) to operationalize the importance of a log event to different class frequencies. Regarding a single log



Log event	C1	C2	C3	C4
e2	2	4	1	3
e3	2	0	0	0
e5	0	1	0	0
e10	0	5	0	0
e11	0	0	0	2

Log event	C1	C2	C3	C4
e2	0.2	0.4	0.1	0.3
e3	1.58	0	0	0
e5	0	1	0	0
e10	0	2.59	0	0
e11	0	0	0	1.58

Log event	C1	C2	C3	C4
e2	0.2	0.4	0.1	0.3
e3	2.43	0	0	0
e5	0	4.54	0	0
e10	0	11.76	0	0
e11	0	0	0	99.14

Figure 3: Lookup Table Construction of NCCHECKER

event $\mathbf{e_i}$, the updated cell value c''_{ij} is calculated as follows:

$$c''_{ij} = c'_{ij} \times ICF_{C_j} \tag{4}$$

$$ICF_{C_j} = \frac{N}{N_{C_j}} \tag{5}$$

where ICF_{C_j} represents the inverse class frequency value with respect to failure cause $\mathbf{C_j}$. N denotes the total number of failed test logs and N_{C_j} denotes the number of failed test logs that belong to $\mathbf{C_j}$. The intuition here is to assign a higher weight to the log events associated with minor classes. For example, the log event $\mathbf{e_{11}}$ is associated with failure cause $\mathbf{C_4}$, which is a minority-class log event. Therefore its value will be updated by multiplying ICF_{C_4} (i.e., 62.75). For the majority-class log event $\mathbf{e_3}$, its value will be updated by multiplying ICF_{C_1} (i.e., 1.54). After updating, the log event $\mathbf{e_{11}}$ has a greater predictive power than $\mathbf{e_3}$ due to rare class frequency.

After the initialization and updating process, the lookup table $\mathbb{T}$ represents the predictive power of different log events for different failure causes. The higher the value, the stronger the indicator the log event is associated with the failure cause.

3.2.3 Failure cause prediction. After constructing the lookup table $\mathbb{T}$, we can easily check out the matching score between a failed log event and a specific failure cause. When it comes to the failure cause prediction stage, for a given newly failed test log, we first perform the log abstraction to parse the log into a sequence of log events. After that, for each log event in the sequence, if the log event appears in the lookup table $\mathbb{T}$, we check out the matching score of this log event with respect to different failure causes. Finally, we sum up all the associated row values, the failure cause with the highest score will be reported as the final prediction result. The log events with the maximum values will be highlighted for investigation.

4 EMPIRICAL EVALUATION

In this section, we first present the baselines, the evaluation metrics and our experiment settings. We then describe the results of our automatic evaluation results.

4.1 Baselines

To demonstrate the effectiveness of our proposed model, NNCHECKER, we compared it with the following chosen baselines:

- **Random Guess (RG).** Regarding the test failure cause prediction, the RG model randomly determine a failure cause for a given failed test log according to the data distribution. In particular, we predict the failure cause randomly and we repeat the random prediction 100 times to get a median performance.
- **The Majority Class Classifier (MCC).** Considering the data imbalance problem in our dataset, e.g., 63% of the test failures are caused by bug related issues, it is reasonable to use the majority class classifier as a baseline for comparison purposes. In particular, the majority classifier model predicts the most frequent class label (i.e., bug related problem in our study) for all test samples.
- **CAM.** Jiang et al. [18] proposed a a novel approach CAM (Cause Analysis Model) that infers test failure causes by analyzing test logs. CAM runs TF-IDF across the logs to determine which terms had the highest importance, it then construct attribute vectors based on test log terms. When a new test alarm occurs, CAM calculates the log similarity between the new test log and the historical test logs. Finally, the unseen failed test log is categorized by examining the categories of the K nearest neighbours.
- **LogFaultFlagger.** Amar et al. [2] recently proposed an approach named LogFaultFlagger to predict bugs and localize faults in test logs via mining historical test logs. LogFaultFlagger vectorizes the logs with line-IDF metrics and uses EKNN to identify which logs are likely to lead to product faults and which lines are the most probable indication of the failure. We adapt the LogFaultFlagger for our task of failure cause prediction.

4.2 Evaluation Metrics

Since our task is a multi-class classification problem, we adopted the widely-accepted evaluation metrics, i.e., Precision, Recall and F1-score to evaluate the performance of NCCHECKER and baseline methods. We define the following statistics with respect to our task: (i) TP_k (True Positives): TP_k is the number of test logs assigned correctly to class k. (ii) FP_k (False Positives): FP_k is the number of test logs that do not belong to class k but assigned to class k incorrectly by classifier. (iii) FN_k (False Negatives): FN_k is the number of test logs that do not assigned to class k by classifier but

which actually belong to class k. Our evaluation metrics are defined as follows:

- **Precision:** Precision is the fraction of true positive samples divided by the total number of positively predicted samples (column sum). The precision metric for class k, namely p_k, is defined as follows:

$$p_k = \frac{TP_k}{TP_k + FP_k} \tag{6}$$

The precision over all K categories (macro average precision) is defined as follows:

$$Precision = \frac{\sum_{k=1}^{K} p_k}{K} \tag{7}$$

- **Recall:** Recall is the fraction of true positive samples divided by the total number of positively classified samples (row sum). Similarly, the recall metric for class k, namely r_k, is defined as follows:

$$r_k = \frac{TP_k}{TP_k + FN_k} \tag{8}$$

The recall over all categories (macro average recall) is defined as follows:

$$Recall = \frac{\sum_{k=1}^{K} r_k}{K} \tag{9}$$

- **F1-score:** F1-score is the harmonic mean of precision and recall, which combines both of the two metrics above. It evaluates if an increase in precision (or recall) outweighs a reduction in recall (or precision), respectively. The F1-score metric for class k, namely $f1_k$, is defined as follows:

$$f1_k = \frac{2 \times p_k \times r_k}{p_k + r_k} \tag{10}$$

The F1-score over all categories (macro average f1-score) is defined as follows:

$$F1 = \frac{\sum_{k=1}^{K} f1_k}{K} \tag{11}$$

The higher an evaluation metric, the better a method performs. It is worth mentioning that we did not consider accuracy due to data imbalance.

4.3 Experimental Settings

We divide our dataset into training set and testing set, the training set is used for learning and building the lookup table, and the testing set is hold out to evaluate the performance of our approach (we did not use validation in this study since there are no hyper parameters of our model for fine tuning). In particular, we randomly sampled 10% of failed test logs for testing and kept the rest for training. The details of the training and testing samples as well as the log events are summarized in Table 3. All experimental results are conducted over a server equipped with Intel(R) Core(R) CPU i7-4790 at 3.60GHZ on 32GB RAM, four cores, and 64-bit Windows 10 operating system.

Table 3: Statistics of our training/testing Datasets

Type	Measurement	Count
C1. Bug related	Train Logs	2,600
	Test Log	289
	Train Log events	91,237
	Test Log events	11,916
C2. Environment	Train Logs	885
	Test Log	99
	Train Log events	26,388
	Test Log events	5,112
C2. Test Script	Train Logs	470
	Test Log	47
	Train Log events	18,724
	Test Log events	5,062
C2. TP Lib	Train Logs	65
	Test Log	8
	Train Log events	5,166
	Test Log events	1,563

4.4 Quantitative Analysis

4.4.1 RQ1: The Effectiveness Evaluation. To evaluate the effectiveness of our proposed model, i.e., NCChecker, we compare NCChecker with the baseline methods on our testing set in terms of overall Precision, Recall and F1-score. The evaluation results is shown in Table 4. From the table, we have the following observations:

- The currently state-of-the-art models, e.g., CAM and LFF performs suboptimal on our evaluation set. The CAM creates TF-IDF vectors based on the raw log messages and ranks the logs using cosine similarity. While the LFF parses the raw logs into log events and creates the TF-Line IDF vectors based on the parsed log events. Considering the constructed TF-IDF vectors are relatively sparse, the retrieval-based models (i.e., CAM and LFF) can easily ignore the critical log events, which contain the key information for indicating the failure root cause correctly.
- Our model, i.e., NCChecker, outperforms all the baseline methods by a large margin in terms of all evaluation metrics. We attribute this to the following reasons: First, it uses the log abstraction techniques to parse the unstructured log lines into structured log events, instead of recording the overwhelming amount of log messages, NCChecker transforms a log file into a sequence of log events, which can extract useful patterns from raw log messages. Second, compared with the information retrieval based models, which rely on the test log from existing database, our model predicts the failure causes by using finer-granularity log event elements. We designed heuristic rules to boost the predictive power of the critical log events, when two raw logs are dissimilar but share a number of critical log events, our model can easily make the correct predictions by checking these critical log events.

Table 4: Overall Effectiveness Evaluation

Measure	Precision	Recall	F1
RG	25.3%	25.0%	19.7%
MCC	25.0%	16.4%	19.8%
CAM	53.1%	61.8%	55.0%
LFF	37.2%	48.3%	37.8%
NCCHECKER	**72.0%**	72.4%	**71.9%**

Answer to RQ-1: How effective is our approach for test failure causes prediction? – We conclude that our approach is highly effective for identifying the root causes of failed test logs.

4.4.2 RQ2: Class-Wise Evaluation. The test failures are caused by a various of failure causes. Accurately identifying the failure causes are important because different actions need to be taken subsequently. One of the key challenges with respect to our task is that the class imbalance problem, i.e., the majority classes are more frequent occurring than the minority classes. For example, the **C1** (bug related issues) accounts for more than 63% of all failed test logs. To verify the effectiveness of our model for identifying failure causes with respect to different failure cause classes, we conduct a class-wise evaluation in this research question. In particular, we evaluate the performance of NCCHECKER and baselines regarding different types of failure causes. The evaluation results are shown in Table 5. It can be seen that:

- It is obvious that all the approaches (excluding ours) achieve a better performance on majority classes (e.g., **C1** and **C2**) than the minority classes (e.g., **C3** and **C4**). This is because the majority class have more examples, the models can learn meaningful patterns from these abundant training samples, while it is more difficult to explore the minority class patterns given its smaller sample size and skewed class distribution.
- The RG and MCC model achieve the worst performance with respect to the minority class failure causes. This is reasonable because the prediction results of RG and MCC model simply rely on the proportions of the majority classes. The likelihood of assigning a test failure into the minority classes is very small.
- The CAM and LFF model have their advantage as compared to the RF and MCC model. It is notable that CAM can achieve a comparable (i.e., **C1**) or better (i.e., **C2**) performance than our approach, but its predictive performance on minority classes are still suboptimal. This is because that the CAM and LFF are information retrieval based models. Their performance heavily rely on whether similar test logs can be found and how similar the test logs are. Considering the limited number of test logs belong to the minority classes, similar logs are hard to find in the training set.
- Regarding the minority class, it is obvious that our model, NCCHECKER, outperforms all other baselines. Rather than checking the similar test logs in history, NCCHECKER maintains a lookup table by estimating the predictive power of the historical failed log events, we designed heuristic rules to increase the predictive power of the minority-class log events. As a result, if the minority-class log events appear in the minority-class test logs, our model are more likely to infer correct failure causes regardless of the size of the minority-classes.
- There is still a large room for our approach to improve regarding minority-classes. For example, the precision of our model for **C3** is 54.3%, which means around half of the test script failure test logs are wrongly assigned. The reason may be that there are log events only available in the testing set and not in the training set, our model is unable to handle the out-of-table log events.

Table 5: Class-wise Effectiveness Evaluation

Type	Approach	Precision	Recall	F1
C1. Bug	RG	26.7%	65.2%	37.9%
	MCC	65.6%	**100.0%**	79.2%
	CAM	83.3%	81.4%	82.3%
	LFF	80.7%	71.9%	76.0%
	NCChecker	**85.4%**	80.9%	**83.1%**
C2. Env	RG	26.5%	23.4%	24.9%
	MCC	0.0%	0.0%	0.0%
	CAM	**69.4%**	**73.9%**	**71.5%**
	LFF	35.3%	41.1%	38.0%
	NCChecker	60.2%	67.0%	63.4%
C3. Test	RG	19.6%	9.4%	12.7%
	MCC	0.0%	0.0%	0.0%
	CAM	45.6%	42.0%	43.7%
	LFF	28.6%	30.3%	29.4%
	NCChecker	**56.5%**	**66.7%**	**61.2%**
C4. Lib	RG	2.9%	1.8%	3.3%
	MCC	0.0%	0.0%	0.0%
	CAM	14.3%	50.0%	22.2%
	LFF	4.1%	50.0%	7.7%
	NCChecker	**85.7%**	**74.9%**	**80.0%**

Answer to RQ-2: How effective is our approach for predicting different types of failure causes? – We conclude that our approach is effective for failure causes prediction with respect to different failure types and can successfully handle the minority classes.

4.4.3 RQ3: Ablation Evaluation. The key to our test failure cause prediction task is how effectively the lookup table can capture the relationship between different log events and failure causes. We propose three key heuristic rules guiding us to build the lookup table. Specifically, we construct the lookup table by following four key steps: step1 (diff with pass), step2 (lookup table initialization), step3 (lookup table updating with single/multi-problem log events), step4 (lookup table updating with minority/majority-class log events). To study the effectiveness of our three heuristic rules, we conduct an

Table 6: Ablation Evaluation

Measure	Precision	Recall	F1
Drop 1	39.7%	58.3%	39.1%
Drop 2	44.9%	36.7%	17.2%
Drop 3	48.2%	78.1%	55.3%
NCChecker	72.0%	72.4%	71.9%

ablation analysis to evaluate their effectiveness and contributions one by one. We compare NCChecker with three of its incomplete versions:

- **Drop 1:** We drop step1 (diff with pass) in this version. The lookup table is constructed by step2 (lookup table initialization), step3 (lookup table updating with single/multi-problem log events) and step4 (lookup table updating with minority/majority class) only.
- **Drop 2:** In this version, we drop step3 (lookup table updating with single/multi-problem log events). The lookup table is constructed by step1 (diff with pass), step2 (lookup table initialization), and step4 (lookup table updating with minority/majority-class) only.
- **Drop 3:** In this version, we drop step4 (lookup table updating with minority/majority-class) and construct the lookup table only with step1 (diff with pass), step2 (lookup table initialization), step3 (lookup table updating with single/multi-problem log events).
- **NCChecker:** Our model which considers all the steps to construct the lookup table.

The experimental results are shown in Table 6. From the table, several points stand out:

- **No matter which step we remove, the overall performance of our model decreases.** This shows the importance and usefulness of our key insights. All three assumptions provide valuable information to construct the lookup table respectively.
- **Drop 2 achieves the worst performance.** It is clear that there is a significant drop overall in every evaluation measure after removing step3. This signals that the step3 (i.e., lookup table updating with single/multi problem log events) is the most important of all the steps for constructing the lookup table and has major contributions to the overall performance.

Answer to RQ-3: How effective is our use of three heuristic rules for constructing lookup table? – We conclude that all the three heuristic rules are effective and helpful to enhance the performance of our model.

4.4.4 RQ4: Computation Resources Evaluation. Due to the huge number of test logs, the rapidly increasing computation resources (e.g., computation time and memory usage) are key challenges for model design. In this research question, we analysis the time consumption as well as memory consumption of our model.

Regarding the time consumption, we record the training time and testing time of NCChecker. The time consumption of NCChecker on training is mostly for the log abstraction. It takes three hours to parse all the test logs (including failed test logs and passed test logs) using Drain. However, we argue that the log abstraction process is a one-time cost. The subsequent operations of making the lookup table is highly efficient. According to the Equations defined in Section 3, the lookup table can be constructed by going through all the log events only once, which is dependent on the sizes of the training logs and generated log events.

The time cost advantage of NCChecker is more obvious in terms of the testing procedures. Regarding the evaluation, the time complexity of IR based models are $O(N)$, while the time complexity of our model is $O(1)$. This is because for a given test log, IR based models calculate the similarity score across all training logs. For our approach, the failed reason can be predicted by checking out the associated log events scores from the lookup table directly. NCChecker takes 8.75 seconds for analyzing the 443 test logs, which means it costs only 20ms on average to check each log.

Regarding the memory consumption, the raw log files are often very large. In particular, it cost over 3GB to store the raw log files. After log parsing, all the log events and their frequency are recorded and the memory consumption is significantly dropped (from 3GB to 6M). By employing our approach, we only need manage and maintain a relatively small size lookup table (i.e., 8.9KB), which records all relevant scores between different log events and failure causes, for diagnosing test failures.

Answer to RQ-4: How effective is our approach in terms of the computation resources? – We conclude that our approach is efficient and memory saving.

5 RELATED WORK

5.1 Log-Based Root Cause Analysis

Root cause analysis, also known as failure diagnosis, aims to identify the underlying causes leading to a test failure that has affected end users. Root cause analysis is a crucial step for effectively resolving the software problems, which is extremely expensive and inefficient [14, 41, 44].

As modern software system grows rapidly and becomes more mature, test failures are more and more difficult to analysis and diagnose [4, 20, 46]. For example, Jiang et al. [19] reported that problem debugging is time-consuming and challenging which can be improved by using logs. They suggested developers to automate failure diagnosis process to speed up the problem fixing time. Zhou et al. [45] studied the failure debugging process with respect to microservice systems. They concluded that proper tracing and visualization techniques can improve failure diagnosis, which shows the necessity for intelligent log analysis tools.

Researchers have developed different techniques for automating the log-based failure cause diagnosis [2, 18, 36]. The works most similar to ours are the retrieval-based root cause analysis methods. In particular, retrieval-based methods retrieve similar failures in history for better diagnosing newly-occurred failures. Shang et al. [36] focused on diagnosing applications in Hadoop system by injecting failures manually and analyzing the logs. Nagaraj et al. [30] investigated the system behaviors in good or bad performance.

They adopted machine learning techniques to automatically infer failures by analyzing the correlations between performance and system components. Jiang et al. [18] proposed CAM to failure cause analysis for test alarm in system and integration testing. For a given test alarm, they searched the test logs of historical test alarms that may have the same failure cause with the new test log. In particular, the similar matching is conducted by using K nearest neighbors (KNN) algorithm between log vectors, where log vectors are built on test log terms extracted by term frequency-inverse document frequency (TF-IDF). Following that, Amar et al. [2] extended CAM by removing log lines that passed the test while keeping log lines only occurred in failed test logs. Then the historical logs were vectorized by a modified Line-IDF metrics. The vectors were utilized to utilized to train an EKNN model to identify most probable log lines that led to the test failures. Even though the retrieval-based methods are proposed to predict the test failure causes, they performed relatively poor with respect to the minority test failure causes, because they heavily rely on the sizes of the similar test logs, our approach based on heuristic rules can effectively handle the minority failure causes.

5.2 Log-Based Failure Prediction

Different from the root cause analysis which diagnoses the causes after the test fails, failure prediction aims to proactively predict the failure before it happens. Failure prediction is essential for predictive maintenance due to its ability to prevent failure occurrences and maintenance costs [1].

A common practice of failure prediction is analyzing the system logs, which record the system status, changes in configuration, operational maintenance, etc. The source of failure can be divided into two categories, homogeneous systems, and heterogeneous systems, and the mainstreaming failure prediction approaches of different categories are different [14].

In homogeneous systems (e.g., large-scale supercomputers), failure prediction approaches mainly focus on modeling sequential information. Sahoo et al. [35] collected the system log of components' health status and leveraged several time series models to predict the health of each node in the system through indication metrics, such as the percentage of system utilization, usage of network IO, and system idle time. Klinkenberg et al. [21] trained a binary classification model from the system log and detected the potential node failure given a time sequence of monitoring data collected from each node. Das et al. [6] adopted deep learning technology and proposed Desh to predict the failure of each node. It firstly recognized the log events chain leading to node failure, then it trained the log events chain recognition with expected lead times to node failure. Finally, Desh can predict the lead time of specific node failure.

In heterogeneous systems (e.g., cloud systems), failure prediction approaches mainly focus on modeling relationships among multiple components. Chen et al. [3] proposed AirAlert to find the dependence between the alerting signal extracted from system logs by the Bayesian network. Then AirAlert predicted failure based on a gradient boosting tree. Lin et al. [24] designed MING which combined the LSTM and Random Forest model to o find the relationship between logs and the failure from temporal and spatial features.

6 THREATS TO VALIDITY

Threats to internal validity are related to potential errors in the code implementation and experimental settings. To reduce the errors in automatic evaluation, we have double checked the code of our approach and baselines. Regarding the experiment results, we have carefully tuned the parameters of baseline approaches and used them in their highest performing settings for comparison.

Threats to external validity are related to the generalizability of the our experimental results. The generalizability of the root cause prediction algorithm in NCChecker should be further explored, since our algorithm may be sensitive to datasets. To alleviate this threat, we evaluate our approach over industry datasets with more than 10K test logs. The ground truth of our dataset are of high quality because they are manually labelled by software developers/testers.

Threats to construct validity relate to suitability of our evaluation metric selection. We use the widely-accepted evaluation metrics (i.e., Precision/Recall/F1-score) to evaluate the effectiveness of our approach and baselines in our experiments. Since our dataset is highly imbalanced with respect to different types of causes, we use macro Precision/Recall/F1 metrics to estimate the overall performance. In addition, there are other software artifacts such as source/test code we did not use in this study, the performance of our approach may be further improved by leveraging more software artifacts.

7 CONCLUSION

This research aims to automatically predict test failure causes for failed test logs. To address this task, we first collected more than 10K test logs from our industry partner and manually labeled the failed reason for each failed test log. We propose an approach named NCChecker (Naive Failure Cause Checker) by leveraging log parsing and heuristic rules. Extensive experiments have demonstrated its effectiveness and promising performance for test failure causes prediction. Considering the effectiveness and simplicity of our approach, we recommend relevant practitioners to adopt our approach as a baseline for the failure causes prediction task.

ACKNOWLEDGMENT

This research is supported by the Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study, Grant No. SN-ZJU-SIAS-001. This research is partially supported by the Shanghai Sailing Program (23YF1446900) and the National Science Foundation of China (No. 62202341). This research is partially supported by the Ningbo Natural Science Foundation (No. 2023J292). This research was also supported by the advanced computing resources provided by the Supercomputing Center of Hangzhou City University. The authors would like to thank the reviewers for their insightful and constructive feedback.

REFERENCES

[1] A Abu-Samah, MK Shahzad, E Zamai, and A Ben Said. 2015. Failure prediction methodology for improved proactive maintenance using Bayesian approach. *IFAC-PapersOnLine* 48, 21 (2015), 844–851.

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

[3] Yujun Chen, Xian Yang, Qingwei Lin, Hongyu Zhang, Feng Gao, Zhangwei Xu, Yingnong Dang, Dongmei Zhang, Hang Dong, Yong Xu, et al. 2019. Outage prediction and diagnosis for cloud service systems. In *The World Wide Web Conference*. 2659–2665.

[4] Edward Chuah, Shyh-hao Kuo, Paul Hiew, William-Chandra Tjhi, Gary Lee, John Hammond, Marek T Michalewicz, Terence Hung, and James C Browne. 2010. Diagnosing the root-causes of failures from cluster log files. In *2010 International Conference on High Performance Computing*. IEEE, 1–10.

[5] Hetong Dai, Heng Li, Che Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2020. Logram: Efficient log parsing using n-gram dictionaries. *IEEE Transactions on Software Engineering* (2020).

[6] Anwesha Das, Frank Mueller, Charles Siegel, and Abhinav Vishnu. 2018. Desh: deep learning for system health prediction of lead times to failure in hpc. In *Proceedings of the 27th international symposium on high-performance parallel and distributed computing*. 40–51.

[7] Min Du and Feifei Li. 2016. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 859–864.

[8] Diana El-Masri, Fabio Petrillo, Yann-Gaël Guéhéneuc, Abdelwahab Hamou-Lhadj, and Anas Bouziane. 2020. A systematic literature review on automated log abstraction techniques. *Information and Software Technology* 122 (2020), 106276.

[9] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*. 24–33.

[10] Zhipeng Gao, Xin Xia, David Lo, and John Grundy. 2020. Technical Q8A Site Answer Recommendation via Question Boosting. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2020), 1–34.

[11] Zhipeng Gao, Xin Xia, David Lo, John Grundy, Xindong Zhang, and Zhenchang Xing. 2023. I know what you are searching for: Code snippet recommendation from stack overflow posts. *ACM Transactions on Software Engineering and Methodology* 32, 3 (2023), 1–42.

[12] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. Logmine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 1573–1582.

[13] 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.

[14] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. 2021. A survey on automated log analysis for reliability engineering. *ACM Computing Surveys (CSUR)* 54, 6 (2021), 1–37.

[15] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2018. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 60–70.

[16] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2016. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 207–218.

[17] 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.

[18] He 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.

[19] Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. 2009. Understanding customer problem troubleshooting from storage system logs. In *Proccedings of the 7th conference on File and storage technologies*. 43–56.

[20] Soila P Kavulya, Kaustubh Joshi, Felicita Di Giandomenico, and Priya Narasimhan. 2012. Failure diagnosis of complex systems. In *Resilience assessment and evaluation of computing systems*. Springer, 239–261.

[21] Jannis Klinkenberg, Christian Terboven, Stefan Lankes, and Matthias S Müller. 2017. Data mining-based analysis of HPC center operations. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 766–773.

[22] Zhenhao Li, An Ran Chen, Xing Hu, Xin Xia, Tse-Hsun Chen, and Weiyi Shang. 2023. Are They All Good? Studying Practitioners' Expectations on the Readability of Log Messages. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 129–140.

[23] Zhenhao Li, Chuan Luo, Tse-Hsun Chen, Weiyi Shang, Shilin He, Qingwei Lin, and Dongmei Zhang. 2023. Did we miss something important? studying and exploring variable-aware log abstraction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 830–842.

[24] Qingwei Lin, Ken Hsieh, Yingnong Dang, Hongyu Zhang, Kaixin Sui, Yong Xu, Jian-Guang Lou, Chenggang Li, Youjiang Wu, Randolph Yao, et al. 2018. Predicting node failure in cloud service systems. In *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 480–490.

[25] Adetokunbo Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. 2011. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering* 24, 11 (2011), 1921–1936.

[26] Leonardo Mariani and Fabrizio Pastore. 2008. Automated identification of failure causes in system logs. In *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 117–126.

[27] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. 2018. A search-based approach for accurate identification of log message formats. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 167–16710.

[28] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, and Hua Cai. 2013. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems* 24, 6 (2013), 1245–1255.

[29] Andriy Miranskyy, Abdelwahab Hamou-Lhadj, Enzo Cialini, and Alf Larsson. 2016. Operational-log analysis for big data systems: Challenges and solutions. *IEEE Software* 33, 2 (2016), 52–59.

[30] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 353–366.

[31] Liqiang Nie, Xiaochi Wei, Dongxiang Zhang, Xiang Wang, Zhipeng Gao, and Yi Yang. 2017. Data-driven answer selection in community QA systems. *IEEE transactions on knowledge and data engineering* 29, 6 (2017), 1186–1198.

[32] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and challenges in log analysis. *Commun. ACM* 55, 2 (2012), 55–61.

[33] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry practices and event logging: Assessment of a critical software development process. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 169–178.

[34] Fangcheng Qiu, Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Xinyu Wang. 2021. Deep Just-In-Time Defect Localization. *IEEE Transactions on Software Engineering* (2021).

[35] Ramendra K Sahoo, Adam J Oliner, Irina Rish, Manish Gupta, José E Moreira, Sheng Ma, Ricardo Vilalta, and Anand Sivasubramaniam. 2003. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. 426–435.

[36] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Brain Adams, Ahmed E Hassan, and Patrick Martin. 2013. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 402–411.

[37] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. 2017. AnswerBot: Automated generation of answer summary to developers' technical questions. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 706–716.

[38] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 117–132.

[39] Zhipeng Xue, Zhipeng Gao, Xing Hu, and Shanping Li. 2023. ACWRecommender: A Tool for Validating Actionable Warnings with Weak Supervision. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1876–1880.

[40] Yanming Yang, Xing Hu, Zhipeng Gao, Jinfu Chen, Chao Ni, Xin Xia, and David Lo. 2024. Federated Learning for Software Engineering: A Case Study of Code Clone Detection and Defect Prediction. *IEEE Transactions on Software Engineering* (2024).

[41] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. 2014. Comprehending performance from real-world execution traces: A device-driver case. In *Proceedings of the 19th international conference on architectural support for programming languages and operating systems*. 193–206.

[42] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)* 30, 1 (2012), 1–28.

[43] Hamzeh Zawawy, Kostas Kontogiannis, and John Mylopoulos. 2010. Log filtering and interpretation for root cause analysis. In *2010 IEEE International Conference on Software Maintenance*. IEEE, 1–5.

[44] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. 2019. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 131–146.

[45] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering* 47, 2 (2018), 243–260.

[46] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 683–694.

[47] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. 2019. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 121–130.

Received 2024-02-08; accepted 2024-04-18