Just-in-Time Flaky Test Detection via Abstracted Failure Symptom Matching

1st Gabin An
KAIST
Daejeon, Korea
gabin.an@kaist.ac.kr

2nd Juyeon Yoon
KAIST
Daejeon, Korea
juyeon.yoon@kaist.ac.kr

3rd Thomas Bach
SAP
Waldorf, Germany
thomas.bach03@sap.com

4th Jingun Hong
SAP Labs Korea
Seoul, Korea
jingun.hong@sap.com

5th Shin Yoo
KAIST
Daejeon, Korea
shin.yoo@kaist.ac.kr

***Abstract*—We report our experience of using failure symptoms, such as error messages or stack traces, to identify flaky test failures in a Continuous Integration (CI) pipeline for a large industrial software system, SAP HANA. Although failure symptoms are commonly used to identify similar failures, they have not previously been employed to detect flaky test failures. Our hypothesis is that flaky failures will exhibit symptoms distinct from those of non-flaky failures. Consequently, we can identify recurring flaky failures, without rerunning the tests, by matching the failure symptoms to those of historical flaky runs. This can significantly reduce the need for test reruns, ultimately resulting in faster delivery of test results to developers. To facilitate the process of matching flaky failures across different execution instances, we abstract newer test failure symptoms before matching them to the known patterns of flaky failures, inspired by previous research in the fields of failure deduplication and log analysis. We evaluate our symptom-based flakiness detection method using actual failure symptoms gathered from CI data of SAP HANA during a six-month period. Our method shows the potential of using failure symptoms to identify recurring flaky failures, achieving a precision of at least 96%, while saving approximately 58% of the machine time compared to the traditional rerun strategy. Analysis of the false positives and the feedback from developers underscore the importance of having descriptive and informative failure symptoms for both the effective deployment of this symptom-based approach and the debugging of flaky tests.**

***Index Terms*—flaky test, continuous integration, error message, failure symptoms**

I. Introduction

SAP HANA is an in-memory Database Management System (DBMS) that is used by many of the largest enterprises around the world, offering both an on-premise installation and a database-as-a-service solution. A commercial database service like SAP HANA requires tremendous effort in testing because the cost incurred by production errors is prohibitively expensive. Therefore, SAP HANA is tested systematically against every incoming code change to detect bugs as early as possible in its Continuous Integration (CI) environment [1].

As pointed out in previous work [1], one of the main challenges of testing SAP HANA is *flaky tests* [2], [3] that both fail and pass against the same version of the source code, making their results less actionable. The flakiness of tests not only diminishes the reliability of test results [4], but also increases the cost of testing in SAP HANA. This is because failed test cases are often re-executed multiple times to determine if they consistently fail or not, in the so-called *rerun strategy* [5], [6]. Our investigation on the pre-submit testing data of SAP HANA shows that 87% of test failures are discovered to be flaky, and a significant portion of the total testing time per test run, with a maximum of 67% and an average of 10%, is spent on rerunning the failed tests. Such increase in testing time means that developers have to wait longer for test results, diminishing their productivity [7], [4].

To address these challenges, researchers have proposed various flaky test detection techniques that do not require reruns. Some techniques rely on only static features, such as the vocabulary of source code [8], [9], [10] or test smells [11]. While these approaches are useful for identifying source code patterns that are indicative of potential flakiness in a test suite, they cannot accurately handle a single test case that can result in both flaky and non-flaky failures. Note that *flaky failure* refers to a failure that is not consistently reproduced against the same version of the program. Dynamic techniques, on the other hand, focus on failing test executions. DeFlaker [7], for example, determines the flakiness of test failure based on whether it executes the recently changed code or not. However, by design, this technique is unable to detect the flaky failures whose coverage overlaps with the changed code. Moreover, it requires precise code coverage, the cost of which makes it less practical for use in extensive testing of large-scale projects.

Meanwhile, there has been a significant amount of research on failure deduplication [12], [13], [14], [15], [16], [17], which uses easily obtainable failure symptoms such as stack traces or error messages to identify and group failures that have the same root cause. These studies have demonstrated that failure symptoms can be valuable information sources for identifying duplicate failures. In the context of flaky tests, it is also assumed that failure symptoms contain information about the cause of flakiness. For example, Flakes, a flaky test management system offered by CloudBuild [18], uses failure symptoms in the bug report assignment process, linking multiple flaky tests with similar error messages to the same bug report [19]. Additionally, FlakeRepro [20], a flaky test reproduction technique, uses error messages to determine whether a flaky failure has been successfully reproduced. However, to the best of our knowledge, failure symptoms have not yet been explicitly used to detect recurring flaky failures in CI systems, despite their usefulness in failure deduplication.

In this paper, we propose a lightweight and black-box approach to detect recurring flaky tests in a CI environment using failure symptoms and information from historical test executions. Our approach is based on the idea that failure symptoms can be linked to the root cause of the flakiness and therefore can be used as an indicator of the flakiness. During the CI cycle, we gather the symptoms of the flaky failures, which are discovered by the rerun strategy, and subsequently use them to predict whether a new failure is flaky or not. If the symptoms of a new failure have been frequently observed in previous flaky failures, the new failure is regarded as flaky without re-executions of tests. To increase the chances of matching symptoms from failures with the same root cause across different execution instances, we abstract the symptoms to include only the most relevant information related to the failure. When evaluated with historical CI data of SAP HANA, our approach achieves a 96% precision and a 76% recall in detecting flaky failures. The ablation study outcomes reveal that the abstraction of symptoms has a great impact on the performance, increasing recall from 50% to 76% while retaining a similar level of precision. Additionally, we find that our prediction can potentially save about 58% of machine time spent for rerunning the failed tests. Furthermore, the abstraction also enables the grouping of similar and recurring flaky failures, which can help the manual investigation by developers. Overall, these results demonstrate the substantial promise of using failure symptoms to identify recurring flaky failures in a CI pipeline.

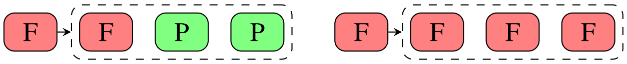
We summarise the contributions of this work as follows:

- **Novel Black-box Flakiness Detection**: While failure symptoms have previously been used to group similar failures, we are the first, to the best of our knowledge, to apply and evaluate their effectiveness specifically in the context of flakiness detection.
- **Enhanced Detection Through Abstraction**: We demonstrate the benefits of using two abstraction methods, number masking from the log analysis and stack trace purification from the failure deduplication, in detecting flaky failures.
- **Real-World Evaluation**: We extensively evaluate our approach using a substantial volume of real-world failure data obtained from SAP HANA.

The remainder of this paper is organised as follows. Section II provides background information on our target software, SAP HANA, and discusses the problem of flaky tests. Section III describes our method for identifying flaky failures using failure symptoms. We describe the evaluation settings in Section IV, and present the results and discussions in Section V and Section VI, respectively. In Section VII, we survey related work in the areas of flaky test detection and failure deduplication. Finally, we conclude in Section VIII.

II. Background

This section outlines the testing pipeline, and discusses the issue of flaky tests, in our target project, SAP HANA, a large-scale DBMS that consists of millions of lines of C++ code and about 1 million test cases [1].



(a) Flaky (b) Non-Flaky

Fig. 1: Example of flaky (left) and non-flaky (right) test results. Red and green rectangles represent failed and passed executions, respectively. After the first failure, each test is re-executed three times, which is denoted by the dashed line.

A. *Testing Pipeline of* SAP HANA

SAP HANA is rigorously and methodically tested across multiple stages within its CI environment: details have been reported by Bach et al. [1]. To briefly summarise, the testing pipeline consists of four main phases: local testing, pre-submit testing, post-submit testing, and extended testing. During development, developers locally validate the new changes by creating new tests or reusing existing regression tests. Once new changes are submitted, they are once again tested in the pre-submit testing stage, before being integrated into their respective components and, ultimately, the main branch. A change is incorporated into the main branch if and only if it passes the pre-submit testing. After being merged into the main branch, changes go through post-submit testing, which is conducted daily using additional tests that require more resources: this further ensures that the current version of the software functions properly. Finally, once the main codebase is ready to be released, extended testing is conducted, using both automated and manual testing, to ensure that the release candidate satisfies all requirements and lacks any regression.

We note that, in SAP HANA, tests are executed in the form of **test suites**, each of which includes multiple **test cases**.[1] Test cases are modular and can share helper functions. The automated testing runs a set of selected test suites in parallel.

B. *Flaky Test Problem in* SAP HANA

We consider a test result flaky when the test both passes and fails in multiple executions against the same version of the program [3]. Flaky test results are frequently observed during the testing of SAP HANA [1]. To identify flakiness in automated testing, SAP HANA adopts the popular *rerun* strategy: if a test suite reports a failure, it is rerun typically three times to determine whether the failure is flaky or not. The process is depicted in Figure 1.

This work specifically focuses on addressing the issue of flaky tests in the pre-submit testing phase of SAP HANA. Although flaky test results can be detrimental to any testing stage, they are particularly harmful to the pre-submit testing due to its higher frequency: according to Bach et al. [1], the pre-submit testing is performed about 80 times a day for the main branch, whereas the post-submit testing is done on a daily basis. Specifically, the flaky test issue induces the following two main problems in the pre-submit testing.

[1] Due to its large size and complexity, SAP HANA contains multiple test suites, each of which validates a certain functionality.

First, since pre-submit testing is performed more frequently than the subsequent phases, the rerun strategy consumes a much higher amount of machine resources when applied to the pre-submit testing. Our analysis of the historical CI data reveals that, often, hundreds of flaky test outcomes are produced per day. Consequently, multiple executions required by reruns not only incur a significant computational cost but also increase the overall turnaround time of testing, harming developer productivity. Our analysis of the past testing history of SAP HANA shows that on average 10% of total testing time is spent on reruns to diagnose flakiness, with a maximum of 67% when there are a large number of failures. A lightweight yet accurate technique that can predict whether an observed failure is flaky or not can reduce the rerun cost, particularly for test cases that are resource-intensive or time-consuming to execute [21].

Second, the large number of flaky results produced during the pre-submit testing phase also means that analysing and improving test flakiness would require a significant amount of human effort. Given its complexity, SAP HANA contains many potential causes of flakiness, such as bugs in source or test code, infrastructure issues, or external factors like errors in third-party libraries. The overwhelming number of flaky failures can lead developers to ignore flaky tests instead of analysing and improving them, potentially resulting in lower software quality standards. Such a loss of trust in the outcomes of tests can have a negative impact on the overall quality of the product [22]. An automated analysis technique that groups flaky results according to their shared root causes can help developers deal with flaky tests more effectively.

To sum up, we aim not only to predict whether the observed failure is flaky or not, but also to precisely group flaky results that share the same root cause, in the pre-submit testing phase.

```
/* stack trace #1, #2 */
Traceback (most recent call last):
  File NewDbTestCase.py line 937, in run
    self.setUp()
  File testCrossDBAtrMultiDB.py line 303, in setUp
    super(testCrossDBAtrMultiDB, self).setUp()
  File testCrossDBQuery.py line 1359, in setUp
    self.conn2 = self.conman2.createConnection()
  File connectionManager.py line 629, in createConnection
    return self.createNamedConnection(conn_id, **kw_args)
  File connectionManager.py line 704, in
        ↪createNamedConnection
    **props)
  File connectionManager.py line 113, in __init__
    retryChecker(dbapi.Connection.__init__, self, **keys)
  File RetryChecker.py line 20, in __call__
    return function(*args, **kwargs)
/* error message #1 */
Error: (-10709, Connection failed (RTE:[89006] System call '
      ↪connect' failed, rc=111:Connection refused {
      ↪1.2.3.3:30024 -> 1.2.3.3:31144} (1.2.3.3:30024 ->
      ↪1.2.3.3:31144)))
/* error message #2 */
Error: (-10709, "Connection failed (RTE:[89006] System call
      ↪'connect' failed, rc=111:Connection refused {
      ↪1.2.3.4:29616 -> 1.2.3.4:31144} (1.2.3.4:29616 ->
      ↪1.2.3.4:31144))")
```

(a) Symptoms of flaky failures

```
/* stack trace */
Traceback (most recent call last):
  File NewDbTestCase.py line 952, in run
    testMethod() # actually run the test
  File testCrossDBAtrMultiDB.py line 12487, in
        ↪testCrossDB_ATR_BinaryDataSync_SubTable
    self._execute(cursors[1], """ALTER REMOTE SUBSCRIPTION "%
          ↪s"."SUB_%s" DISTRIBUTE """ % (schemas[1], tables
          ↪[1]))
  File testCrossDBAtrMultiDB.py line 429, in _execute
    self.fail("%s failed with %s" % (statement, str(err)))
/* error message */
AssertionError: ALTER REMOTE SUBSCRIPTION "db2"."SUB_tbl2"
      ↪DISTRIBUTE failed with (129, 'transaction rolled back
      ↪ by an internal error: table REP::db2:TARGET_tbl2 (t
      ↪2030) not locked by tablelock(false) or rowlock(false
      ↪); $condition$=xlocked rowlocked')
```

(b) Symptoms of a non-flaky failure

Fig. 2: Symptoms from both flaky and non-flaky failures observed from the same test case in SAP HANA.

C. Motivating Example

Figure 2 provides a detailed example to demonstrate the motivation behind our proposed approach. We have selected a test case in SAP HANA that has exhibited both flaky and non-flaky behaviour in the past. Figure 2a shows parts of Python stack traces and error messages observed from two flaky failures of the test case. We can observe that the root cause of this flakiness is related to a database connectivity issue, as well as the specific call sequences that triggered this issue. Furthermore, the symptoms of flaky failures in this test case are distinguishable from those of non-flaky failures of the same test case, which are presented in Figure 2b. Based on this example, we conjecture that flaky failures sharing a root cause will result in similar error messages or stack traces, that are distinct from those of non-flaky failures. We also observe that it is common for flaky failures to be recurring across different pre-submit testing runs, as the underlying cause of the failure may not have been fully identified or resolved: in our example, the test case failed in 54 pre-submit testing runs between January and June in 2022. Out of these 54 failures, 52 (96%) were caused by the same database connectivity issue, and their symptoms were exactly identical to those shown in Figure 2a, except for the IP addresses highlighted in the grey background colour. These observations motivate us to detect recurring flaky failures using their symptoms.

III. Just-in-Time Flakiness Detection Using Abstracted Failure Symptoms

This section presents our approach to detect flaky test failures during the pre-submit testing of SAP HANA using failure symptoms, e.g., stack traces and error messages. These are lightweight and black-box information sources that can be accessed without incurring additional execution or instrumentation costs. This allows us to design an efficient flakiness detection technique that can be seamlessly integrated into the CI pipeline in a just-in-time manner. The remainder of this section explains the details of our approach.

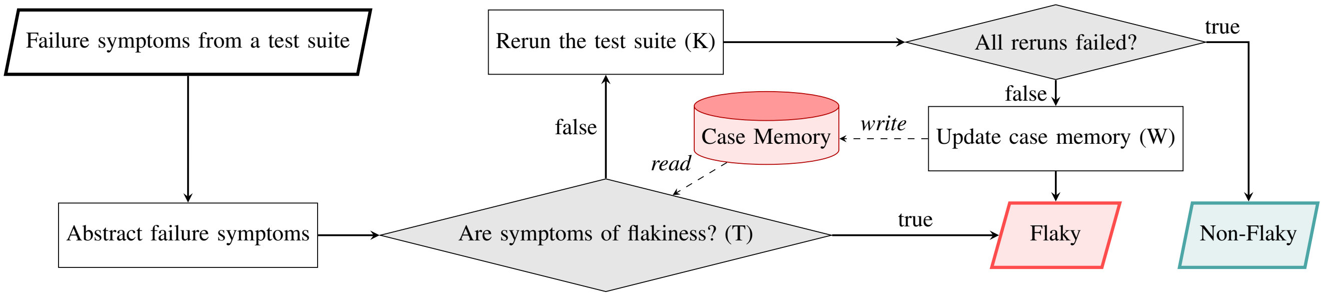


Fig. 3: The overview of our flakiness detection approach during a pre-submit testing phase. The solid and dashed lines represent the control and data flow, respectively. The hyperparameters T, K, and W respectively denote the minimum frequency threshold, the number of reruns, and the minimum word count for symptoms.

A. *Overview*

We propose a novel flakiness detection approach which is a hybrid of the conventional rerun strategy and the symptom-based flakiness detection. In our approach, the rerun strategy is used to systematically collect flaky test failures in a sound way, i.e., producing no false positives, whereas the symptoms of those failures are then regarded as a signal of flakiness and used to detect future flaky failures in a just-in-time manner.

The overall workflow of the proposed method is depicted in Figure 3. Suppose that a test suite fails during the pre-submit testing phase. To decide whether to rerun the test suite, we first collect the set of failure symptoms, S, i.e., stack traces and error messages of the test suite failure. For example, if N test cases within the test suite have failed, we collect the failure symptoms from each test case ($|S| = N$). Subsequently, we abstract each of the collected failure symptoms in S by discarding less relevant details (Figure 3: **Abstract failure symptoms**, see Section III-B for details). The set of abstracted symptoms, denoted as S', are used to look up the case memory, FFS, which contains the known **F**laky **F**ailure **S**ymptoms. FFS is a hash memory, where the key is the abstracted symptoms, and the value is the past observation count (default is 0). To determine whether the currently observed symptoms S' are the symptoms of flakiness or not, we use the following count-based matching function:

$$AreFlaky(S') := \begin{cases} true, & \text{if } \forall s \in S, FFS(s) \geq T \\ false, & \text{otherwise} \end{cases}$$

If all collected symptoms in S have a past observation count greater than a pre-defined threshold, T, the failure of the test suite is classified as flaky, and no further reruns are performed. However, if at least one of their observation count is lower than T, we explicitly check the flakiness by rerunning it K times (Figure 3: **Rerun the test suite**). If all K reruns fail consistently, the failure is classified as non-flaky. Otherwise, the failure is classified as flaky, and the case memory is updated accordingly by incrementing the observation count of the symptoms of the failed test cases (Figure 3: **Update case memory**). During this process, we heuristically filter out symptoms that are less likely to contain sufficient information about the root cause of the flakiness. We only store symptoms that have at least W unique tokens with only alphabetic characters in their error messages. Let $S'_W \subseteq S'$ denote the set of symptoms that satisfy such a condition. Then, the case memory is updated for each of the symptoms in S'_W, i.e., $FFS(s) := FFS(s) + 1$ for all $s \in S'_W$.

Note that we maintain the case memory of flaky failures instead of the non-flaky ones for a specific reason. While the opposite approach, i.e., collecting and matching symptoms of non-flaky failures, is possible, it would be less accurate because of the inherent limitations of testing: a finite number of reruns can only prove flakiness, not non-flakiness. Consequently, symptoms of flaky failures can be collected reliably, while those of non-flaky failures cannot.

Our approach is matching-based [16] rather than similarity-based [23], [24], [12], [14], because matching is more efficient and scalable. Unlike hash-based matching with constant computational complexity, a similarity-based approach would require comparing the current symptoms with every past symptom, which is computationally expensive to be performed during testing. To further increase the effectiveness of matching, we abstract the failure symptoms to include only the information most relevant to the failure. The next subsection describes the details of the abstraction.

B. *Abstraction of Failure Symptoms*

Symptoms of flaky failures with the same root cause may not be exact matches to each other, due to subtle differences such as the IP addresses in Figure 2a. To achieve better matching, we propose to *abstract* the failure symptoms to include only essential information related to the potential root causes, and to prevent minor differences from hindering correct matches. We apply purification and number masking to stack traces and error messages, respectively.

Stack Trace Purification: Since test cases of SAP HANA are written as Python functions, most of the test failures are reported with their Python stack traces. The traces contain function names, line numbers, file names, and the source code line for each call frame unless the corresponding test suite terminates abnormally. We purify the raw stack traces to contain only the essential information that captures the dynamic flow of the test execution: we first extract only the file and function names using regular expressions to filter out subtle

```
testCrossDBQuery.py,setUp
connectionManager.py,createConnection
connectionManager.py,createNamedConnection
connectionManager.py,__init__
RetryChecker.py,__call__
```

(a) Example of the purified stack trace. After extracting only the file and function names from the original stack trace (Figure 2a), the entry points for the test execution, i.e, the first two calls `run` and `setUp`, are discarded.

```
Error: (-#, Connection failed (RTE:[#] System call ’
     ↪connect’ failed, rc=#:Connection refused {
     ↪#.#.#.#:# -> #.#.#.#:#} (#.#.#.#:# -> #.#.#.#:#))
     ↪)
```

(b) Example of the abstracted error message. The numbers in the original error message (Figure 2a) are replaced with # by masking.

```
[callstack]
testCrossDBQuery.py,setUp
connectionManager.py,createConnection
connectionManager.py,createNamedConnection
connectionManager.py,__init__
RetryChecker.py,__call__
[message]
Error: (-#, Connection failed (RTE:[#] System call ’
     ↪connect’ failed, rc=#:Connection refused {#.#.#.#:#
     ↪ -> #.#.#.#:#} (#.#.#.#:# -> #.#.#.#:#)))
```

(c) The abstracted stack trace and the error message are concatenated to form the symptom of a failure.

Fig. 4: Example of the abstracted failure symptom

differences, such as line number or source code style change. Subsequently, we remove entry points for test execution, i.e., the functions that are called to initiate the testing process, from the stack trace. This is to enable the matching of failure symptoms across different test suites that eventually trigger the same function sequences containing the root cause of flakiness. As a result, the stack trace is represented as a sequence of file and function pairs. Figure 4a shows the purified version of the raw stack trace from Figure 2a.

Number Masking: We observe that many dynamic parts in the error message are specific numbers, e.g., IP addresses, dates, memory addresses, etc. To filter out such details, we replace all numbers with the character `#`. For example, the IP addresses in Figure 2a are masked to `#.#.#.#:#` as shown in Figure 4b. All hexadecimal numbers are also masked using the regular expression `0[xX][0-9a-fA-F]+`. This strategy is motivated by anonymization [25], abstraction [26], removal of variables [27], and removal of numbers [28] in previous test log analysis techniques.

Finally, the abstracted stack traces and error messages are then concatenated as shown in Figure 4c.

IV. Experimental Setup

We describe the experimental setup for evaluation.

TABLE I: Statistics of 4,576 pre-submit testing records collected from SAP HANA. “F” represents flaky test failures, and “NF” represents non-flaky test failures.

Statistics	**Total**
# executed test suites	8,750,036
# failed test suites (F+NF)	58,927
# failed test suites (F)	51,183
# failed test suites (NF)	7,744
# failed test suites w/ test case failures (F+NF)	15,114
# failed test suites w/ test case failures (F)	11,599
# failed test suites w/ test case failures (NF)	3,545
# failed test suites w/ valid symptoms (F+NF)	13,168
# failed test suites w/ valid symptoms (F)	9,857
# failed test suites w/ valid symptoms (NF)	3,311

A. Dataset Construction

We evaluate our approach using the past pre-submit testing records from SAP HANA. Specifically, after collecting pre-submit testing results from January to June 2022, we assume that our technique was deployed in January 2022, with an empty corpus of flaky failure symptoms, and was used to detect the flakiness of future failures until June 2022. While SAP HANA has various combinations for the compiler and platform options for testing, we consider a single combination in our evaluation for the sake of simplicity. As a result, 4,576 pre-submit testing records are collected from the specified date range. Table I shows the detailed statistics of the dataset, including the total number of executed test suites, the number of failed test suites (both flaky and non-flaky), the number of failed test suites with test case failures (both flaky and non-flaky), and the number of failed test suites with valid symptoms (both flaky and non-flaky). We observe that test suites often fail outside their test cases. For example, a test suite can crash during its setup or teardown process performed before and after the actual test case execution, or can be terminated due to timeout constraints. In such cases, the test suite cannot be associated with any test case failures. The data retention policy of SAP HANA does not keep the error messages in such cases for a long time, as such failures are outside the main testing processes. Therefore, we only consider the failures that occur during the execution of test cases in our analysis.[2] Further, we found that some failure symptoms are not informative to be used as a signal for flakiness: for example, an error message “Unit test failed - Log Preview not supported.” does not provide any helpful information about its root cause. Therefore, we have manually mined a set of non-useful patterns of error messages and filtered out failures in our dataset that match the mined patterns. In total, we collected 13,168 test suite failures that occurred during the execution of test cases and have valid symptoms for every test case, corresponding to 22.3% of the total failures. Among them, 9,857 failures are flaky, while 3,311 failures are non-flaky.

[2] We note that our method can be extended to failures outside of test cases as long as the failure symptoms can be collected. This is discussed further in Section VI.

This flakiness label is assigned based on the previous three rerun results of the failures. It should be noted that the non-flaky label may not be accurate as the reruns are not complete, i.e., they may not detect all flaky failures.

B. Hyperparameter Settings and Other Details

Three hyperparameters, T, W, and K, can be adjusted to optimise performance. First, the matching threshold, T, determines whether a given set of failure symptoms is an indicator of a flaky failure. A higher value of T would lead to a more conservative detection. During our evaluation, T is set with values of $\{1, 2, 3, 4, 5, 6\}$. Second, the minimum required number of unique words in error messages, W, is used to heuristically control the quality of the failure symptoms. Like T, a higher value of W would be more conservative. During our evaluation, W is set with values of $\{1, 2, 3, 4, 5, 6\}$. Lastly, the hyperparameter K is used to determine the number of times each failed test suite should be rerun, and set to 3 to align with the established practice in SAP HANA.

Furthermore, we assume that the pre-submit testing runs are executed sequentially in order of their starting time, for the sake of simplicity. In addition, in a single pre-submit testing run, the case memory, FFS, is updated all at once after all necessary reruns for any failed test suites have been completed, because the test suites are executed in parallel in SAP HANA. We assume a sequential order between pre-submit test runs to ensure that the case memory is fully updated after each run, before the subsequent run starts.

V. Results

In this section, we present the findings from our evaluation. Section V-A reports the accuracy of our symptom-based flakiness detection approach. Section V-B studies the impact of abstraction on the effectiveness of flakiness detection. Section V-C analyses the potential savings in test resources achievable through our approach. Finally, Section V-D provides a more in-depth analysis of false positive cases and discusses their implications.

A. Detection Accuracy

We measure the precision and recall of our flakiness detection approach based on the simulation results. As a baseline for precision, we calculate the precision obtained from a naive model that always predicts positive, which is equivalent to the proportion of flaky examples (among all failures with valid symptoms) in our dataset, $\frac{9{,}857}{13{,}168} \approx 0.749$. To assess the overall performance of the detection, we also compute the F1 score.

The performance of our symptom-based detection approach is presented in Figure 5, where we examine the precision, recall, and F1 scores for different hyperparameter settings of T and W. Our approach consistently achieves precision of at least 0.958, which is 28% higher than the naive baseline. These results indicate that the failure symptoms can serve as effective indicators of flakiness. In contrast, the recall values show a wider distribution, ranging from 0.423 to 0.758, which in turn results in F1 scores ranging from 0.591 to 0.847. Overall, the best performing hyperparameter configuration is $T = 1$, $W = 1$, with the F1 score of 0.847. Note that, due to the inherent nature of our method, recall values are sensitive to the frequency of recurring flaky failures. If a specific root cause of flaky failure manifests itself only once, our method will not be able to detect it, even under the least conservative hyperparameter configuration of $T = 1$. As such, we note that the reported recall values are dependent on the specific data we used, i.e., the CI history from the six-month period.

We observe a trade-off between precision and recall against different hyperparameter settings. Increasing T and W leads to a more conservative detection approach, thereby increasing the precision, whereas lowering them would match more flaky symptoms and result in higher recall, saving more testing resources for reruns (See Section V-C). This trade-off provides the flexibility to finetune hyperparameters for meeting the specific requirements of the testing process. For example, if saving computational resources is the more pressing concern, one can opt for higher recall at the cost of spending human analysis cost to filter out false positives. On the other hand, if human analysis cost is the more pressing concern, one can finetune for higher precision and instead accept more reruns. In addition, other aspects can influence the selection of hyperparameters, such as whether there exists a subsequent safeguard (e.g., re-execution of all tests in the later post-submit testing stage) in the CI pipeline.

B. Impact of Abstraction

We perform an ablation study to see the impact of each of the abstraction methods on the performance of our flakiness detection approach. The boxplots in Figure 6 show the precision and recall of our approach with different abstraction settings (y-axis). Note that each boxplot shows the precision and recall values across all tested hyperparameter settings. Abstracting the failure symptoms increases recall on average by about 0.220 (from 0.352 to 0.572) against all hyperparameter values. At the best-performing hyperparameter configuration, $T = 1$ and $W = 1$, the abstraction increases the recall by 0.255 (from 0.503 to 0.758). While applying the symptom abstraction significantly increases recall, we can see that it does not sacrifice precision much; the precision drop is only 0.002 on average. These results show that abstraction can help effectively matching flaky failure instances that have slightly different symptoms from each other.

We also evaluate the direct impact of the abstraction on the symptoms. Figure 7 shows the average number of characters in failure symptoms (i.e., length) and the number of unique failure symptoms of test cases at each abstraction setting. The average length of failure symptoms decreases via abstraction, which is expected because the abstraction removes unnecessary information from the symptoms. We observe that the decrease in the number of unique failure symptoms is much more significant than the decrease in the length of symptoms: the number of unique failure symptoms is significantly reduced by the abstraction, from 102,529 to 16,345 (-84%). Especially, masking numbers in error messages is effective in reducing the

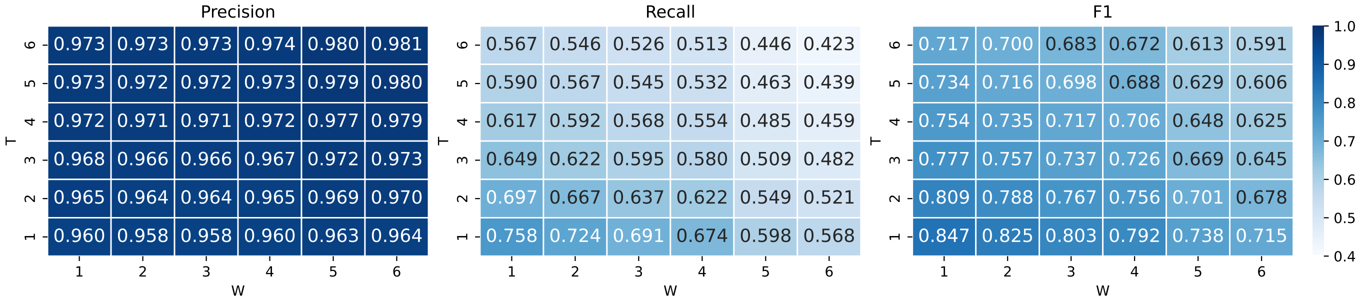


Fig. 5: The performance of flaky failure detection on different hyperparameter settings. Each heatmap represents the precision, recall, and F1 of the detection results for each combination of T and W. The darker the cell is, the higher the value is.

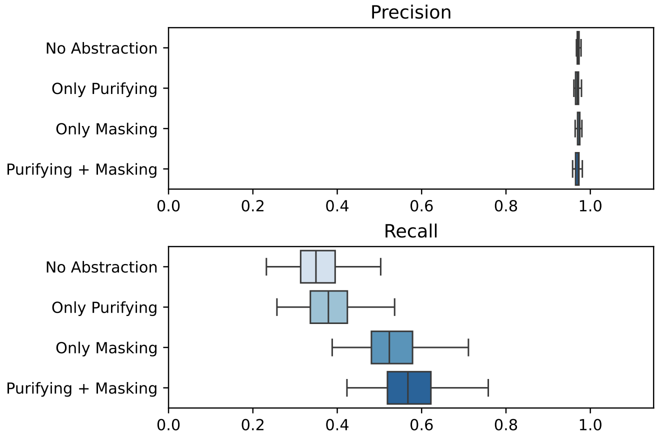


Fig. 6: Precision and Recall for each abstraction setting: without abstraction, after only purifying stack trace, after only masking numbers, and after both purifying and masking. Each box shows the results for all hyperparameters.

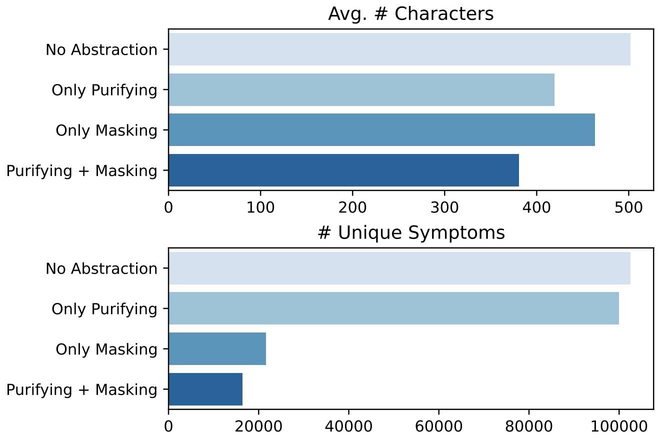
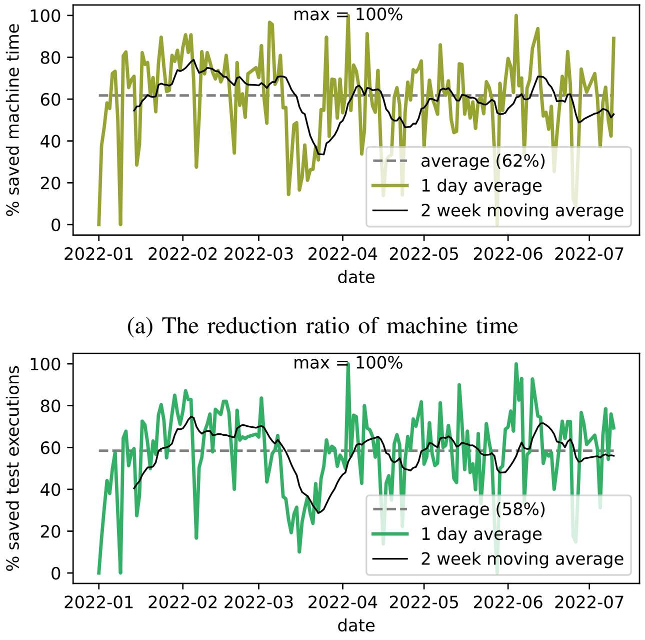


Fig. 7: The average character length of symptoms and the number of unique failure symptoms for each abstraction stage: without abstraction, after only purifying stack trace, after only masking numbers, and after both purifying and masking.



(a) The reduction ratio of machine time

(b) The reduction ratio of the number of test executions

Fig. 8: Percentages of the potential savings in the machine time and the number of test executions by our approach ($T = 1$ and $W = 1$) compared to the rerun strategy for each day

number of unique failure symptoms. This demonstrates that the abstraction of symptoms not only enhances the recall of our approach but also facilitates the grouping of similar failures into a single class based on the symptoms. For example, the abstracted symptoms in Figure 4c are matched to raw symptoms from 1,522 failures across 120 test cases in 56 pre-submit testing runs. We expect that this automated failure grouping can help developers identify the root cause of flaky failures more efficiently.

C. Resource Savings

We compute the percentages of the machine time and the number of test executions that can be saved by using our flakiness detection approach, compared to the conventional rerun strategy. Originally, each pre-submit testing run requires,

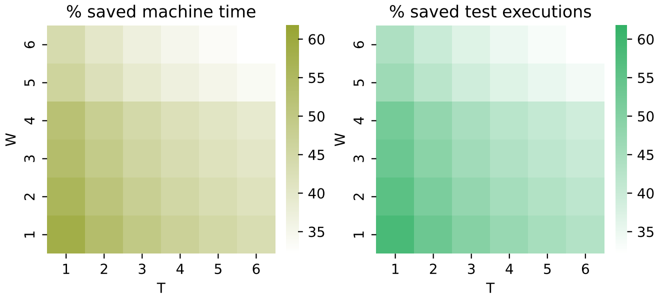


Fig. 9: Heatmaps showing the average percentages of the machine time and the number of test executions saved by using our flakiness detection approach at each hyperparameter setting compared to the conventional rerun strategy

on average, nine additional test executions and 3.13 hours of machine time for rerunning the studied failures (i.e., the 13,168 test failures with valid symptoms), which amounts to 219 executions and 78 hours spent per day.

Figure 8a and Figure 8b show the total average, one-day average, and two-week moving average of the percentages of the machine time and the number of test executions saved by our approach with $T = 1$ and $W = 1$. Our approach can save up to 100% of both test executions and machine time associated with reruns a day. On average across the entire period of evaluation, 62% of test executions and 58% of machine time can be saved when compared to the rerun strategy. Figure 9 shows the trend in the resource savings for every hyperparameter setting. Higher T and W values lead to lower resource savings, as they lead to fewer flaky failures being detected (i.e., lower recall). At the most conservative hyperparameter setting, $T = 6$ and $W = 6$, test executions and machine time could be potentially saved by 33% and 32%, respectively. The results show that, although reducing both the values of T and W may not always be ideal (due to more false positives), it can effectively increase the cost savings in testing, particularly if coupled with an additional safeguard in subsequent testing stages. In the case of SAP HANA, since our symptom-based matching targets only the pre-submit testing stage, subsequent post-submit testing will still consider tests that have been flagged to be flaky in the pre-submit stage. Consequently, any false positives can be rectified through reruns in the post-submit testing.

D. Analysis of False Positives

The previous results in Section V-A show that our approach achieves a high precision of above 96%. However, there is still a small number of *false positive* test suite failures that are labelled as non-flaky in the dataset but predicted as flaky by our approach. Theoretically, these false positives can be classified into two categories:

- (Case 1) The *Non-Flaky* label is incorrect: As discussed in Section IV-A, test suites are labelled based on the reruns with $K = 3$. However, a limited number of reruns can still result in an incorrect "non-flaky" label [29]. Failures in this category are not *real* false positives from our approach, but rather the result of the limitations of the rerun strategy.
- (Case 2) The *Flaky* prediction is incorrect: In instances where symptoms stored in the case memory are not a valid indicator of flakiness, they may lead our approach to incorrectly predict non-flaky test suite failures as flaky. Samples in this category are *real* false positives.

```
/* example 1 */
failureException: one or more single tests failed in <
      ↪UnitTests>
/* example 2 */
Test is marked as failed because it generated 1 unexpected
      ↪ output
/* example 3 */
Test failed. Most likely, the test crashed or hit the test
      ↪ timeout.
There could be other reasons as well, e.g., subsequent XML
      ↪ update could not be written.
/* example 4 */
AssertionError: Test returned errorcode (rc = 2), error:
/* example 5 */
Failure in 1 out of 32 batched results. Failed tests:
      ↪_traceMdsSc3.QMDS5781Queries in batch:
      ↪test_traceMdsSc3.QMDS5755,<... stripped ...>,
      ↪test_traceMdsSc3.QMDS5787
```

Fig. 10: Examples of uninformative error messages

Based on this categorisation, we analysed 82 false positive predictions that are shared by all hyperparameter settings. To categorise false positive results, we first consult the historical manual review of the pre-submit test results. We conjecture that, if a failure belongs to Case 2 and is actually non-flaky (called *test breakages* in SAP HANA), the corresponding code change will not be merged into the main branch. However, we find that, for 39 out of 82 false positive failures (48%), the corresponding code changes have been successfully merged into the main branch after developers manually reviewed the test results. This suggests that the flaky label for these 39 failures is likely to be incorrect, i.e., they actually belong to Case 1 (we hereby refer to them as C1 candidates). Since the testing system of SAP HANA allows us to re-trigger past pre-submit testing runs whose corresponding code change has been successfully merged, we attempted to rerun the C1 candidates to actually verify whether they are incorrectly labelled. Among the 39 C1 candidates, we were unable to verify 15 candidates due to limitations in the testing infrastructure or technical issues. The additional reruns for the remaining 24 C1 candidates reveal that all 24 are indeed verified to be flaky, i.e., their non-flaky labels are incorrect. The analysis of C1 candidates suggests that at least 29% (=24/82) of the initial false positive samples are actually true positives (i.e., Case 1) so that the actual precision and recall of our approach are higher than those reported in Section V-A.

To determine the cause of incorrect predictions for Case 2, we have manually examined the failure symptoms of the remaining 43 false positive predictions. By definition, false positives in Case 2 mean that some symptoms stored in, and matched from, the case memory are not exclusive to flaky failures. We find that most of these symptoms are uninformative and vague, despite our attempt to filter out such symptoms (see Section IV-A). Figure 10 shows false positive symptoms from Case 2: they only indicate that a failure has occurred, without providing any information on the internal program states or the location where the crash occurred. This shows that, in order to further enhance the precision of our approach, it is necessary to either construct a more thorough list of patterns of uninformative symptoms to filter them out or, more fundamentally, improve the quality of the test cases so that their failure symptoms contain more meaningful information. In this regard, a careful manual analysis of false positives from the historical data (i.e., non-flaky test runs confirmed by reruns) can be useful not only for improving the precision of the proposed technique (by filtering out unhelpful symptoms), but also for improving the overall quality of tests (by rewriting them to be more informative).

VI. Discussion

This section presents the developer feedback on the usefulness of failure symptoms and discusses a potential extension of our approach to detect other flaky failures in SAP HANA.

A. Developer Feedback on the Usefulness of Failure Symptoms in Debugging

We collected a subset of abstracted failure symptoms for flaky failures whose observation counts are more than 20. We asked developers of SAP HANA for their assessments of the usefulness of these symptoms in identifying the root causes of the flakiness. This is to gain an understanding of the developers' perspectives on the utility of the failure symptoms in the debugging process. Note that the answers and feedback can be biased by the experience of the developers. The feedback from developers about the abstracted failure symptoms collected so far is mixed. Some of the failure symptoms are considered to be valuable in determining the source of the flakiness. For example, symptoms that display specific types of errors (such as timeouts, missing attribute errors, or import issues), or those that include specific file names or program components known to cause flakiness, are considered to be effective indicators of potential root causes. However, some more general symptoms are seen as not specific enough to provide enough information to pinpoint the source of the problem. For instance, some symptoms consist only of a stack trace with generic file and method names frequently used by many test cases, or an error message that is too brief, e.g., `AssertionError: # != #`. Essentially, these are symptoms that are similar to Case 2 false positives, described in Section V-D. They do not highlight the cause of flakiness and can be produced by failures due to a variety of reasons, including the test suite, the testing environment, or the program being tested.

The feedback collectively highlights the importance of having informative failure symptoms for effectively detecting and debugging flaky tests. Without detailed symptoms, it becomes challenging or even impossible for developers to accurately determine the source of the failure. In turn, this points to the importance of writing test cases of SAP HANA with more descriptive error messages that clearly indicate the issue including information about relevant program states and any other details that can aid in diagnosis.

B. Addressing Failures Outside of the Main Testing Body

Let us consider the test failures that occur outside the scope of our evaluation. As explained in Section II-B, the test suite of SAP HANA is composed of multiple test cases, which form the main testing body that validates the behaviour of the program. To run the test cases, the test suite first sets up the testing environment, executes the test cases, and then tears down the environment. In Section IV-A, we observe that a large number of flaky failures in SAP HANA happen outside of the main testing body. Only 15,114 (or 30%) of the failures occur during the execution of test cases. We have sampled and manually investigated some of the remaining 70% of failures, and identified two primary reasons for these failures: (1) the setup/teardown part outside of the test cases leads to exceptions or errors (including a timeout) (2) a timeout occurs while executing the test cases, but is not handled gracefully, leaving no meaningful symptom. These points are in line with the results from a previous study [4], which found that developers rate issues with setup/teardown to be the most common causes of flakiness. During the evaluation, we have focused on the flaky failures that occur in the main body of the test suites, because those are the only failures for which we can collect past symptom data. However, we argue that our approach can be extended to flaky failures that occur during the setup and teardown process, as long as they produce valid failure symptoms. Under these circumstances, the set of failure symptoms for a given test suite (S in Section III-A) can be a singleton set containing a symptom from the failure. Similarly, test suites that time out without leaving any symptoms should be improved with better graceful shutdown mechanisms, so that they can produce more informative error messages that summarise the timeout context. We believe that these suggestions, along with the recommendations mentioned in the previous section, can serve as a guide for improving the failure handling practices in SAP HANA.

VII. Related Work

This section covers the related work on flaky test detection and failure deduplication.

A. Flaky Test Detection

A widely adopted way to detect flaky tests is the rerun strategy [5], [6], i.e., to rerun the failed test cases multiple times and check if they eventually pass or not. Gruber et al. [30] find that a large number of reruns is needed to diagnose test flakiness. However, doing numerous reruns is not feasible in practice due to its high cost. To address this challenge, several techniques have been proposed to detect flaky tests without rerunning them.

First, there is a group of techniques that use dynamic features of test executions to detect flakiness. DeFlaker [7] uses coverage information to detect flaky tests that do not execute any of the changed code. FlakeFlagger [29] trains a machine learning model that takes both static code features including test smells and dynamic features such as coverage as input, and predicts whether a given test failure is flaky, with up to 86% accuracy. Among the studied features, dynamic information such as execution time and coverage is found to be the most important features. In case of SAP HANA, its size as well as the overhead for coverage collection forces us to collect coverage only on a weekly basis, making it difficult to apply DeFlaker to our use case. However, in cases where coverage data for the changed code is available, we propose a two-step approach. First, DeFlaker could be used to detect flaky failures by identifying tests that fail without executing any of the changed code. For the remaining failures that cannot be detected by DeFlaker, we would then apply our symptom-based detection technique. This combined approach is expected to enhance the precision of flaky test detection by identifying a broader range of flaky failures.

Second, some approaches look at previous test execution histories to detect flaky failures. Herzig et al. [31] collect both test features (e.g., test case identifier) and test results (i.e., passed or failed), and subsequently use association rule learning to identify patterns of the flaky test results. Kowalczyk et al. [32] quantitatively model the flakiness of a test case based on the temporal variance of its results during test history. Gruber et al. [33] uses several features related to the code evolution and test history data, such as the number of changed files in the most recent pull request or the flip rates of test outcomes, to detect flakiness without reruns.

Last, there are approaches that use static features of the given program, such as source code tokens contained in the test code [8], [9], [10] or test smells [34], [11], to detect the flaky tests. Pinto et al. [8] identify the vocabulary of flaky test cases, i.e., tokens such as job, action, and services that are highly associated with flakiness, and show that a model that solely depends on static code features can achieve high accuracy on flakiness detection. Pontillo et al. [34] investigate the difference between flaky and non-flaky tests in terms of 25 code metrics and smells, which has later been replicated [11] using a different dataset from another study [29].

Our approach can be considered a hybrid of all existing techniques. We use dynamic features from test executions, but only those that do not require costly code instrumentation such as stack traces and error messages. We consider test execution history, but instead of focusing on the overall test outcome of pass or fail, we maintain a case memory focused on the symptoms of flaky failures. Finally, while some of the symptoms we collect are part of the source code, they are not static, as we obtain them via test executions. Note that the same test case can result in both flaky and non-flaky failures without changing its source code, as shown in Figure 2. Our approach can accommodate such variability because it collects failure symptoms dynamically: in contrast, a static approach will permanently label a test case as flaky or not as long as its source code does not change.

B. Failure Deduplication

Flakiness detection can be considered a specific form of failure root cause analysis. Jiang et al. [35] aim at identifying causes of test failures from predefined categories, including test flakiness. By matching test log outputs with textually similar past logs, the technique can suggest detailed causes of the current test failure, which could be more informative than a binary flakiness label.

Another form of failure root cause analysis is failure deduplication, i.e., grouping test failures based on shared root causes [15], [14], [36], [12], [23], [13]. Since deduplication typically follows test execution, outputs of failures, such as stack traces and error messages, are often adopted as inputs: the intuition is that the more similar two stack traces or error messages are, the more likely that the corresponding failures share the common root cause. Bartz et al. [14] confirmed that the edit distance between two stack traces is an important feature for a machine learning classifier trained to identify failures with shared root causes. Lerch et al. [13] reported that stack traces are the most valuable information contained in bug reports that can be used for deduplication.

Since not all failures caused by the same root cause exhibit the exactly same stack trace, various ways of abstracting stack traces have been suggested. Brodie et al. [15] filter out entry points and common error handling routines using a "stop-words" list provided by a domain expert, and remove recursive function calls, before matching stack traces. Modani et al. [17] also remove less relevant functions from stack traces before measuring either the edit distance or the length of the longest common subsequence between them. Joshy et al. [36] use only the top N calls on the stack trace to group the failures. Our stack trace purification is similarly motivated.

There are more complicated approaches for comparing two stack traces to group similar failures. Dang et al. [12] proposed a weighted common subsequence measure to quantify the similarity between two stack traces. Rodrigues et al. [23] designed a new similarity metric between two stack traces based on the optimal global alignment between them. However, as similarity calculation is computationally expensive, these approaches are hard to be applied to our just-in-time flakiness detection.

In addition to the stack traces, error messages, which contain unstructured information about exception types and output values, have been also used to group similar failures. Erman et al. [37] used raw error messages, together with the test case names, to cluster test results. CloudBuild [18], a Microsoft's build-and-test system, contains a flaky test management system called Flakes, which groups and reports flaky failures with similar error messages together [19]. However, our approach differs from the similarity-based approaches as it uses hash-based matching. While matching is much more efficient, it is susceptible to irrelevant details. We apply the number masking to prevent such details from hindering exact matching between similar flaky failures; our masking method is inspired by prior studies on test log analysis [25], [26], [28].

VIII. Conclusion

We report our experience of using failure symptoms as a means to detect flaky failures in SAP HANA in a just-in-time manner. Our approach is inspired by previous failure deduplication studies. We collect symptoms of flaky failure using the conventional rerun strategy, and later detect flaky failures by matching their symptoms to the previous flaky failures, with a precision of up to 98%. Our empirical evaluation with real-world CI data from SAP HANA, along with feedback from developers, yields the following takeaways: 1) Stack traces and error messages are a valuable resource for recognising flaky test failures in a CI pipeline; 2) Abstracting failure symptoms can significantly increase the recall of flakiness prediction, while allowing the automated grouping of flaky failures; 3) Having tests produce detailed and informative failure symptoms is crucial to the accurate detection and debugging of flaky tests. SAP is planning to deploy our symptom-based flaky failure detection technique into the CI/CD pipeline of SAP HANA. In future work, we aim to reduce the false positive rate of our approach by automatically detecting and filtering out uninformative symptoms. In the longer term, we hope that our analysis of the failure symptoms can guide the developers to improve the quality of test outputs.

Acknowledgment

Gabin An, Juyeon Yoon, and Shin Yoo have been supported by SAP Labs, Samsung Electronics (Grant No. IO201210-07969-01), the National Research Foundation of Korea (NRF) funded by the Korean Government MSIT (RS-2023-00208998), the Engineering Research Center Program (2021R1A5A1021944), as well as the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government MSIT (2021-0-01001).

References

[1] T. Bach, A. Andrzejak, C. Seo, C. Bierstedt, C. Lemke, D. Ritter, D. W. Hwang, E. Sheshi, F. Schabernack, F. Renkes, G. Gaumnitz, J. Martens, L. Hoemke, M. Felderer, M. Rudolf, N. Jambigi, N. May, R. Joy, R. Scheja, S. Schwedes, S. Seibel, S. Seifert, S. Haas, S. Kraft, T. Kroll, T. Scheuer, and W. Lehner, "Testing very large database management systems: The case of SAP HANA," *Datenbank-Spektrum*, nov 2022. [Online]. Available: https://doi.org/10.1007%2Fs13222-022-00426-x

[2] W. Zheng, G. Liu, M. Zhang, X. Chen, and W. Zhao, "Research Progress of Flaky Tests," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2021, pp. 639–646, iSSN: 1534-5351.

[3] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A Survey of Flaky Tests," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 17:1–17:74, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3476105

[4] ——, "Surveying the Developer Experience of Flaky Tests," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May 2022, pp. 253–262.

[5] [Online]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

[6] S. Engineering, "Test flakiness - methods for identifying and dealing with flaky tests," Nov 2019. [Online]. Available: https://engineering.atspotify.com/2019/11/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/

[7] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically Detecting Flaky Tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, May 2018, pp. 433–444, iSSN: 1558-1225.

[8] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the Vocabulary of Flaky Tests?" in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 492–502. [Online]. Available: https://doi.org/10.1145/3379597.3387482

[9] B. H. P. Camara, M. A. G. Silva, A. T. Endo, and S. R. Vergilio, "What is the Vocabulary of Flaky Tests? An Extended Replication," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, May 2021, pp. 444–454, iSSN: 2643-7171.

[10] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. Le Traon, "A Replication Study on the Usability of Code Vocabulary in Predicting Flaky Tests," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, May 2021, pp. 219–229, iSSN: 2574-3864.

[11] V. Pontillo, "Static Test Flakiness Prediction," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, May 2022, pp. 325–327, iSSN: 2574-1926.

[12] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: A method for clustering duplicate crash reports based on call stack similarity," in *2012 34th International Conference on Software Engineering (ICSE)*, Jun 2012, pp. 1084–1093.

[13] J. Lerch and M. Mezini, "Finding duplicates of your yet unwritten bug report," in *2013 17th European conference on software maintenance and reengineering (CSMR)*. IEEE, 2013, pp. 69–78.

[14] K. Bartz, J. Stokes, J. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihile, "Finding similar failures using callstack similarity," in *SysML08: Third Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*. USENIX, December 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/finding-similar-failures-using-callstack-similarity/

[15] M. Brodie, S. Ma, G. Lohman, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn, "Quickly finding known software problems via automated symptom matching," in *Second International Conference on Autonomic Computing (ICAC'05)*. IEEE. [Online]. Available: https://doi.org/10.1109%2Ficac.2005.49

[16] M. Brodie, S. Ma, L. Rachevsky, and J. Champlin, "Automated problem determination using call-stack matching," *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 219–237, jun 2005. [Online]. Available: https://doi.org/10.1007%2Fs10922-005-4443-8

[17] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet, "Automatically identifying known software problems," in *2007 IEEE 23rd International Conference on Data Engineering Workshop*. IEEE, apr 2007. [Online]. Available: https://doi.org/10.1109%2Ficdew.2007.4401026

[18] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "CloudBuild," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, may 2016. [Online]. Available: https://doi.org/10.1145%2F2889160.2889222

[19] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A Study on the Lifecycle of Flaky Tests," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, Oct. 2020, pp. 1471–1482, iSSN: 1558-1225.

[20] T. Leesatapornwongsa, X. Ren, and S. Nath, "FlakeRepro: automated and efficient reproduction of concurrency-related flaky tests," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, nov 2022. [Online]. Available: https://doi.org/10.1145%2F3540250.3558956

[21] F. Leinen, D. Elsner, A. Pretschner, A. Stahlbauer, M. Sailer, and E. Jürgens, "Cost of flaky tests in continuous integration: An industrial case study," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2024.

[22] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, oct 2018. [Online]. Available: https://doi.org/10.1145%2F3236024.3275529

[23] I. M. Rodrigues, A. Khvorov, D. Aloise, R. Vasiliev, D. Koznov, E. R. Fernandes, G. Chernishev, D. Luciv, and N. Povarov, "TraceSim: An alignment method for computing stack trace similarity," *Empirical Software Engineering*, vol. 27, no. 2, mar 2022. [Online]. Available: https://doi.org/10.1007%2Fs10664-021-10070-w

[24] J. C. Campbell, E. A. Santos, and A. Hindle, "The unreasonable effectiveness of traditional information retrieval in crash report deduplication," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, may 2016. [Online]. Available: https://doi.org/10.1145%2F2901739.2901766

[25] A. Amar and P. C. Rigby, "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, 2019, pp. 140–151.

[26] M. Nagappan and M. A. Vouk, "Abstracting log lines to log event types for mining software system logs," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, may 2010. [Online]. Available: https://doi.org/10.1109%2Fmsr.2010.5463281

[27] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, jun 2017. [Online]. Available: https://doi.org/10.1109%2Ficws.2017.13

[28] F. Salfner and S. Tschirpke, "Error log processing for accurate failure prediction," in *Proceedings of the First USENIX conference on Analysis of system logs*, ser. WASL'08. USA: USENIX Association, 2008, p. 4.

[29] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "FlakeFlagger: Predicting Flakiness Without Rerunning Tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021, pp. 1572–1584, iSSN: 1558-1225.

[30] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An Empirical Study of Flaky Tests in Python," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2021, pp. 148–158, iSSN: 2159-4848.

[31] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 39–48.

[32] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and Ranking Flaky Tests at Apple," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Oct. 2020, pp. 110–119.

[33] M. Gruber, M. Heine, N. Oster, M. Philippsen, and G. Fraser, "Practical flaky test prediction using common code evolution and test history data," *arXiv preprint arXiv:2302.09330*, 2023.

[34] V. Pontillo, F. Palomba, and F. Ferrucci, "Toward static test flakiness prediction: a feasibility study," in *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution*, ser. MaLTESQuE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 19–24. [Online]. Available: https://doi.org/10.1145/3472674.3473981

[35] H. Jiang, X. Li, Z. Yang, and J. Xuan, "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, 2017, pp. 712–723.

[36] A. K. Joshy and W. Le, "Fuzzeraid: Grouping fuzzed crashes based on fault signatures," p. 12, 2022.

[37] N. Erman, V. Tufvesson, M. Borg, P. Runeson, and A. Ardo, "Navigating information overload caused by automated testing-a clustering approach in multi-branch development," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–9.