230,439 Test Failures Later: An Empirical Evaluation of Flaky Failure Classifiers

Abdulrahman Alshammari
George Mason University
Fairfax, United States
aalsha2@gmu.edu

Paul Ammann
George Mason University
Fairfax, United States
pammann@gmu.edu

Michael Hilton
Carnegie Mellon University
Pittsburgh, United States
mhilton@cmu.edu

Jonathan Bell
Northeastern University
Boston, United States
j.bell@northeastern.edu

***Abstract*—Flaky tests are tests that can non-deterministically pass or fail, even in the absence of code changes. Despite being a source of false alarms, flaky tests often remain in test suites once they are detected, as they also may be relied upon to detect true failures. Hence, a key open problem in flaky test research is: How to quickly determine if a test failed due to flakiness, or if it detected a bug? The state-of-the-practice is for developers to re-run failing tests: if a test fails and then passes, it is flaky by definition; if the test persistently fails, it is likely a true failure. However, this approach can be both ineffective and inefficient. An alternate approach that developers may already use for triaging test failures is failure de-duplication, which matches newly discovered test failures to previously witnessed flaky and true failures. However, because flaky test failure symptoms might resemble those of true failures, there is a risk of missclassifying a true test failure as a flaky failure to be ignored. Using a dataset of 498 flaky tests from 22 open-source Java projects, we collect a large dataset of 230,439 failure messages (both flaky and not), allowing us to empirically investigate the efficacy of failure de-duplication. We find that for some projects, this approach is extremely effective (with 100% specificity), while for other projects, the approach is entirely ineffective. By analyzing the characteristics of these flaky and non-flaky failures, we provide useful guidance on how developers should rely on this approach.**

I. Introduction

Ideally, when an automated test case fails during development, this failure indicates that a defect has been detected. Tests that detect defects are good tests, because they can signal developers to pause working on new development, and to debug and fix the defect promptly. However, some test cases may be "flaky," and can non-deterministically pass or fail, even when repeatedly executed on the same version of the same code. For example, studies have shown that tests can be flaky due to strict reliance asynchronous computations completing in some specific order, reliance on undocumented platform dependencies, among other sources of randomness [1]. When a test fails *due to flakiness*, developers may still need to pause their development activities to confirm that the test failure does not actually represent a true defect.

While a flaky test is one that *can* fail due to some non-deterministic reason, the failures of flaky tests cannot be entirely ignored, as flaky tests can *also* detect defects. For example, Rahman and Rigby studied the Mozilla Firefox continuous integration system, finding that when developers ignored failures of flaky tests, there was a dramatic increase in the number of crashes reported by users [2]. Similarly, Haben et al. analyzed 9 months of test failures in the Google Chromium continuous integration system, finding that ignoring *all* failures of flaky tests would have resulted in missing 76% of the true regression faults [3]. Surveys of developers report that flaky tests waste developers time and are a moderate-to-severe problem for most developers [4], [5], [6].

In order to reduce the burden of inspecting every (possibly flaky) test failure, the state-of-the-practice approach for triaging test failures as flaky or true failures is to rerun failed tests [1], [4], [5], [6]. If, on the same version of the system under test, the test first fails, and then later the test passes, then it is a flaky failure that can be ignored. Unfortunately, this approach is not guaranteed to detect all flaky failures, since a flaky test may also persistently fail. Lam et al. observed that roughly half of the flaky tests in their dataset would persistently fail when re-run in isolation from the rest of the test suite [7]. Bell et al. studied the efficacy of Apache Maven's built-in test rerunning feature, finding that it could only confirm 23% of flaky test failures as flaky [8]. Particularly when tests might need to be re-run *many* times, this procedure is expensive and time consuming. A report from Google indicates that 2-16% of computing resources are regularly dedicated just to re-running flaky tests [9].

Hence, an important problem for flaky test research is: given a test failure, how to determine if this specific failure can safely be ignored (since it is flaky), or more thoroughly debugged (as a true failure). While a significant body of academic literature aims to determine which tests are flaky tests (i.e., could exhibit flaky failures) [7], [10], [11], [12], [13], [14], [15], [16], [17], the problem of distinguishing flaky failures from true failures is understudied. This article presents a large-scale empirical study that characterizes the design space for flaky failure detection, and preliminary results of several baseline approaches. From 22 open-source Java projects, we collect a dataset of 498 flaky tests, including 80,530 flaky failures and 147,613 true failures. This broad dataset complements existing case studies of flaky failures in Google Chromium [3] and SAP HANA [18] by highlighting the variability of flaky failure detection across different projects.

We evaluate the use-case where a developer has an existing history of test failures (both flaky and true failures) and is

This work was supported in part by NSF grant number 2100037

given the task of determining if a new failure is flaky or not. We find that flaky test failures can be extremely repetitive — when a test fails due to flakiness, it is likely to match other flaky failures from the same or other tests. We apply approaches based on failure de-duplication [19], [20], text-based matching, and simple machine learning classifiers. We find that, for some tests, these approaches can be extremely effective (with no false negatives or false positives), yet for other tests, these approaches are entirely ineffective. By examining attributes of tests and failures, we provide insights for future research on generalized approaches for detecting flaky failures.

The primary contributions of this paper are:

- **Evaluation**: A large-scale evaluation of failure de-duplication using failure messages and stacktraces to determine if a failure is flaky or true failure.
- **Dataset**: An extended dataset that contains both flaky and true failures of tests, constructed using a novel approach based on mutation analysis.

II. Matching Failures Logs

Failure logs provide a detailed understanding of the origin of the failure. Hence, developers typically debug logs to better understand the failure cause. In detecting test flakiness, a recent survey shows that some developers may manually debug failures logs to tell if a failure is flaky or not [6]. Developers can recognize a failure is flaky by examining the failure message and stacktraces as they could have encountered flaky failures with similar failure message and stacktraces [21].

We examine the applicability of failure de-duplication to help developers to determine if a new failure is flaky or not. We evaluate three approaches for failure de-duplication: text-based matching, which uses the text of failure logs to find the similarity of given two failures, the Failure Log Classifier, which adopts machine learning to predict if a failure is flaky or not, and TF-IDF (the details appears later in subsection II-C), which uses information retrieval techniques to group failures. We focus specifically on matching the output that is common to the test suites of all projects that we have studied: stack traces. When matching stack traces, we consider two use-cases: matching different failures from the same test, or matching failures from different tests.

A. Text-Based Matching

Text-based matching is our application of classic failure de-duplication approaches [19], [20], where we de-duplicate failures by matching common stack traces. This approach is also motivated by grey-literature suggestions that, "sometimes it's obvious to engineers that a test is flaky just by looking at the exception type and message" [21]. Intuitively, if an engineer has repeatedly seen the same flaky failure symptoms, they may be able to guess that a new failure is also flaky. text-based matching acts as an automated standin for this experience-based process.

We implement text-based matching by creating a dataset of parsed failure logs for each test. Each failure log is represented by its failure message and stacktraces. In terms of a failure message, it consists *exception type* (for example, AssertionFailedError) and everything follows this is treated as the *exception message*. For the stack traces part, it is a set of lines representing the calls before the exception occurs and during the parsing, we are considering the top lines pointing directly to the test name. These lines reflect the most recent operations preceding the exception and often provide more details about the root cause of the failure.

We implement a pipeline to parse each failure into an XML file, cataloging all failures linked to a specific test. As shown in Listing 1, each failure block in the XML corresponds to one failure, containing four key components: the test name (**T**), exception type (**E**), exception message (**M**), and stacktrace lines (**S**). Within the **S** tag, individual lines are listed under the **line** tags, considering their order in the original log. If the test name is missing from the stacktrace (e.g. it fails in setup method), we consider the last line from the test class. For example, in Listing 1, the last line is not starting with the test name (present in **T**) but starts with the test class name. To categorize these XML files per project, the **T** tag includes a *project* attribute, referring the project name where the test belongs. In this phase, we also filter out non-deterministic stack trace lines internal to the JVM (e.g. `GeneratedMethodAccessor$XYZ` lines).

Listing 1. Two flaky failures reported in Alluxio project after parsing their failure logs

```
<Failure>
    <T project="alluxio">tachyon.JournalTest.TableTest</T>
    <E>UnknownHostException</E>
    <M>ip-172-31-48-81: ip-172-31-48-81: Temporary failure in name resolution</M>
    <S><line>java.net.Inet6AddressImpl.lookupAllHostAddr(Native Method)</line>
    <line>java.net.InetAddress$2.lookupAllHostAddr(InetAddress.java:929)</line>
    <line>java.net.InetAddress.getAddressesFromNameService(InetAddress.java:1324)</
       line>
    <line>java.net.InetAddress.getLocalHost(InetAddress.java:1501)</line>
    <line>tachyon.LocalTachyonCluster.start(LocalTachyonCluster.java:104)</line>
    <line>tachyon.JournalTest.before(JournalTest.java:33</line></S>
</Failure>
    ...
<Failure>
    <T project="alluxio">tachyon.JournalTest.TableTest</T>
    <E>UnknownHostException</E>
    <M>ip-172-31-58-81: ip-172-31-58-81: Temporary failure in name resolution</M>
    <S><line>java.net.Inet6AddressImpl.lookupAllHostAddr(Native Method)</line>
    <line>java.net.InetAddress$2.lookupAllHostAddr(InetAddress.java:929)</line>
    <line>java.net.InetAddress.getAddressesFromNameService(InetAddress.java:1324)</
       line>
    <line>java.net.InetAddress.getLocalHost(InetAddress.java:1501)</line>
    <line>tachyon.LocalTachyonCluster.start(LocalTachyonCluster.java:104)</line>
    <line>tachyon.JournalTest.before(JournalTest.java:33)</line></S>
</Failure>
```

The text-based matching relies on the text of the failure message and stacktraces. As shown in Listing 1, we found that the failure message (**M**) could contain information such as timestamp and IP address that make each (otherwise equivalent) failure unique. For example, in Listing 1, different details like an IP address within **M** can set two failures apart. Hence, the text-based matching does not rely on **M**, and consider only stack traces (**S**) and exception type (**E**). Given the challenges in capturing all potential cases where the failure message (**M**) could be identical, we avoid modifying these unique message details and discard the **M** during the comparison.

When given flaky and true failures, the text-based matching should be able to tell if a new failure is a de-duplication of flaky failures, true failures, or both. As this approach is designed to find failure de-duplication within the same test, it

could be useful to applicable across different tests especially if the failure stack traces do not cover the test body, similar to the example provided in Listing 1.

B. Failure Log Classifier

There are cases where a newly written test introduces flakiness, or when there is no prior failures for reference. Motivated by these scenarios, we propose the Failure Log Classifier, which is trained on both flaky and true failures from *all* tests in a test suite. Then the classifier would be able to predict if the new encounter failure is flaky or true failure. For training the Failure Log Classifier, we considered selected the features shown in Table I, based on their generality. We chose the features based on the text of the failure logs. Although other studies for predicting flaky failures use dynamic details [22], our goal is to determine if relying on the information in failure logs can effectively predict flaky failures.

We employ a simple *Decision Tree* as the supervised learning algorithm [23]. Based on the binary features used to train the classifier, decision tree provides a clear way to handle non-linear relationships. As a comparison, we also evaluate the applicability of a Naïve Bayes classifier as well.

C. TF-IDF

Term Frequency-Inverse Document Frequency (TF-IDF) is a commonly used numerical statistic that reflects how important a word is to a document in corpus [24]. TF-IDF has two components: Term Frequency (*TF*) which represents the frequency of a term (word) in a document and if a term appears frequently in a document, its *TF* will be high. Second, Inverse Document Frequency (*IDF*) which measures the significance of the term in the entire corpus and if a term appears in many documents, its *IDF* value will be low, reflecting its lower importance. The TF-IDF value of a term in a document is the product of its *TF* and *IDF* values. Equation 1 and 2 show the computation of *TF* and *IDF*, respectively.

$$\text{TF } (t) = \frac{\text{Number of times term } t \text{ in a document}}{\text{Total number of terms in the document}} \quad (1)$$

$$\text{IDF } (t) = \log(\frac{\text{Total number of documents}}{\text{Number of documents where } t \text{ in it}}) \quad (2)$$

In the context of studying failure logs, we refer *document* to a *failure* and the *t* to the token we extract from each failure message and stacktraces. For each failure in the generated XML file used in the text-based matching, we tokenize each line of each stacktrace (including the exception type) by split the words using the *dot* as separator (and removing the symbols such parentheses). For example, the last line in Listing 1 will be converted to the following set of tokens *(tachyon, JournalTest, before, JournalTest, java, 33)*. As our goal was to evaluate the overall potential for this approach, we did not consider more advanced tokenization approaches [25].

III. Evaluation Methodology

The core contribution in this work is a rigorous empirical evaluation of the three flaky failure detection approaches described in the prior section, using the following methodology:

A. Datasets

In order to effectively evaluate failure de-duplication for flaky failures, we need a dataset that contains a large number of both flaky and true failures for the same test. The"FlakeFlagger" dataset was built by executing the test suites of 26 open-source Java projects 10,000 times and recording their outputs, yielding a large dataset of flaky failures [10]. We choose the FlakeFlagger dataset, as it contains the complete failure logs for each flaky failure, as opposed to other flaky test datasets like DeFlaker's [11] or iDFlakies [12].

Whereas a dataset of flaky failures can be mined by repeatedly running the same versions of the same tests, a dataset of true failures can only be mined from buggy code. While datasets of true failures *do* exist [26], [27], [28], [29], these datasets are typically intentionally constructed from tests that are *not* flaky (to make studying the defects easier). However, we are not aware of any accessible datasets that provide both flaky and true failures logs for the same set of tests. Even if one were to mine failures of flaky tests, there would still be a tremendous dataset imbalance problem: there tend to be far more tests that only fail due to flakiness as opposed to those that might also reveal faults [3].

We propose a novel methodology for constructing a dataset for this experiment, based on mutation testing. Mutation testing runs a program's test suite on generated mutants (variants of the program under test), and evaluates how many of those mutants are detected by a failing test. Mutants have been shown to be an effective substitute for real faults in software testing [30]. Hence, for each of the flaky tests in our dataset, we use mutation testing to build a large dataset of failure logs for true failures. To avoid contaminating the true failure dataset with flaky failures (caused by tests failing due to flakiness on the mutated code), we apply Shi et al.'s approach for filtering flaky mutants [31].

Hence, the dataset for our experiment consists of all of the flaky failures extracted from the FlakeFlagger dataset [10], supplemented by true failures generated by executing Shi et al.'s version of the popular PIT mutation testing tool [31], [32]. This modified version of PIT is configured such that each test-mutant failure is confirmed by re-running the test on that mutant, 20 times. Each failure that is deterministically reproduced is included in our dataset of failures. This confirmation step is necessary to filter out any flaky failures from the mutation dataset, and is used only for confirming that the failure is deterministic (we do not include each failure 20 times from each of the confirmation runs). Then from the collected failure logs of each killed mutant, we collect the failure messages and stacktraces. We extend the XML file per test to include a list of killed mutants, each of them contains the failure message and stacktraces.

TABLE I
FEATURES USED BY THE FAILURE LOG CLASSIFIER

Feature Name	Type	Description
Exception Type	Str	The name of the exception e.g. UnknownHostException
Test name in Stacktrace	Boolean	*True* if one of Stacktrace lines starts with the test name else *False*
Test Class name in Stacktrace	Boolean	*True* if one of Stacktrace lines contains the test class name else *False*
Other Tests in Stacktrace	Boolean	*True* if one of Stacktrace lines starts with other tests names else *False*
JUnit in Stacktrace	Boolean	*True* if one of Stacktrace lines starts with any Junit Lines else *False*
CUT in Stacktrace	Boolean	*True* if one of Stacktrace lines contains any lines from Code Under Test else *False*

In practice, flaky failures tend to be far more common than true failures. Given that the failure message and stacktraces includes the name of each test, the performance of any failure classifier could be misrepresented by a dataset that contained a large proportion of tests that *only* failed due to flakiness. For example, in a 9-month period observing Google's Chromium CI, Haben et al. observed that 1,446 tests failed with only true ("fault-revealing") failures, 22,477 failed with only flaky failures, and 897 failed showing both failures. A predictor based on the historical flaky failure rate of a test would easily have quite high recall at predicting flakiness (e.g. having at most $897/22,477 = 4\%$ true failures incorrectly labeled as flaky). Our goal is to evaluate the performance of approaches that rely primarily on the failure message and stacktraces, and *not* just the historical flake rate of a test.

Hence, we include in our evaluation *only* tests with at least one flaky and non-flaky failure, and report the number of true and flaky failures in our dataset for each project. We were not able to successfully apply the PIT mutation testing tool to all of the projects despite significant efforts (one author expended at least 2 hours per-project to attempt to get it to work) — and hence, we were unable to gather a resource of failures for all projects. As a result, it is important to note that we do *not* include all projects or tests from the FlakeFlagger dataset. Whereas the FlakeFlagger dataset includes 811 flaky tests from 24 projects, we analyze only those tests for which we could collect a dataset of true failures: 498 flaky tests from 22 projects. For example: on the project "spring-boot," the dataset contained 163 flaky tests, but we were only able to successfully run PIT on 12 of the tests. The errors were primarily related to interactions between bytecode instrumentation, classloading, and custom JUnit runners.

To evaluate our classifier, we use cross validation to split the whole dataset (flaky and true failures) to a training set and testing set. As the dataset is imbalanced (with different proportions of failures that are flaky vs true failures), we apply SMOTE [33] when training and utilize stratified cross-validation [34] to ensure that each testing-fold part has at least one flaky failure.

B. Research Questions

Using this dataset of 498 tests with both flaky and true failures, we design an experiment to answer the following research questions:

RQ1: How often are flaky failures repetitive? We examine how frequently a flaky failure matches *at least* one other flaky failure — of the same test or of other tests within the same project. By doing this, we show the repetition of flaky failures and the efficacy of the failure de-duplication approach.

RQ2: With prior flaky and true failures, is it feasible to use failure de-duplication to determine if a failure is flaky or true one? The main objective is to evaluate the effectiveness of using text-based matching as an approach to find the differences between flaky and true failures. This helps practitioners and researchers if they can rely on the approach in detecting flaky failures. Since projects differ in their domain, root causes of flakiness, and the total number of flaky tests, we evaluate the approach on a project-by-project basis.

RQ3: To what extent is the utilization of machine learning helpful in finding the differences between flaky and true failures? We aim to demonstrate the efficacy of employing machine learning classifiers in predicting whether a failure is flaky or not based on specific features extracted from failure logs. We examine whether a classifier can leverage failures from other tests within the same project to enhance the learning process of the model to better predict failures.

IV. RESULTS

A. *RQ1: How often are flaky failures repetitive?*

We apply the text-based matching approach to de-duplicate flaky failures, and summarize the results in Table II. We show results matching the flaky failures within the same test (shown in column *Per Test*), and matching the flaky failures across all failures from all tests in the same project (shown in the column *Across Tests*). By considering *Alluxio-alluxio* as an example from Table II: in the first case, there are 114 flaky tests and those tests cumulatively have 16,858 flaky failures in total (16,847 of them are repetitive and 11 are not). The 16,847 failures that were an exact match for at least one other failure represent just 310 unique failures. In the second case, the number of failures that are *not* repetitive dropped to 5 (16,853 repetitive failures). When comparing a new failure to flaky failures from different tests, the text-based matching might produce mis-match results due to lines in the stacktrace pointing to the test. To mitigate this, we exclude such lines during this type of comparison, ensuring a more accurate match result.

While we found that each flaky test in *Alluxio-alluxio* could have different flaky failures, on average, each flaky test only had just over two different failures, each of which recurred many times. In most of the projects we studied in Table II,

TABLE II
REPETITIVE FLAKY FAILURES WITHIN AND ACROSS TESTS PER PROJECT. Failures column shows the number of flaky failures and the different failures (Set). The columns *Repet* and *Uniq* refer to flaky failures that are and are not repetitive, respectively. Per Test refers to matching the failures within the same test. Across Tests refers matching all flaky failures from all tests.

Projects	Tests	Failures		Per Test		Across Tests	
		Flaky	Set	Uniq	Repet	Uniq	Repet
Alluxio-alluxio	114	16,858	310	11	16,847	5	16,853
square-okhttp	99	26,486	120	40	26,446	17	26,469
apache-ambari	51	4,063	54	0	4,063	0	4,063
hector-client-hector	33	6,529	33	0	6,529	0	6,529
activiti-activiti	30	1,363	31	13	1,350	6	1,357
tootallnate-java-websocket	23	2,143	45	2	2,141	0	2,143
apache-httpcore	22	354	22	9	345	2	352
qos-ch-logback	20	438	21	8	430	4	434
apache-hbase	20	2,519	26	3	2,516	2	2,517
kevinsawicki.http-request	18	3,501	18	3	3,498	0	3,501
wildfly-wildfly	18	50	18	12	38	4	46
wro4j-wro4j	14	10,833	21	3	10,830	2	10,831
spring-projects-spring-boot	12	14	13	12	2	5	9
undertow-io-undertow	7	92	12	3	89	1	91
orbit-orbit	7	2,943	7	0	2,943	0	2,943
elasticjob-elastic-job-lite	3	7	4	3	4	0	7
doanduyhai-Achilles	2	121	3	1	120	1	120
joel-costigliola-assertj-core	1	974	1	0	974	0	974
ninjaframework-ninja	1	476	1	0	476	0	476
apache-commons-exec	1	33	1	0	33	0	33
jknack-handlebars.java	1	411	1	0	411	0	411
zxing-zxing	1	322	1	0	322	0	322
Total	498	80,530	763	123	80,407	49	80,481

there are a reasonable amount of repetitive flaky failures (by both considering the ratio of the number of flaky failures in column (**Uniq**) to the total number of flaky failures or even to the set of flaky failures) as some projects (8 out of 22) have all flaky failures repetitive. Hence, we conclude that, overall, flaky failures are extremely repetitive. While it is inappropriate to assume that each flaky test can only fail with a single set of symptoms, the number of unique failures is dwarfed by the frequency with which those failures recur.

We also carefully examine when flaky failures are not repetitive, and occur only once in the dataset. Across all the studied projects, there are only 123 out of 80,530 flaky failures (also out of 763 sets of flaky failures) that have never matched other flaky failures within the same project. Out of 123 that failed once, we found 90 of them are actually lack of the history of flaky failures (from tests that only failed once). Out of 22 projects, there are only two projects where the number of repetitive flaky failure is just equal or less than the number of non-repetitive flaky failures (*elasticjob-elastic-job-lite* and *Spring-projects-spring-boot*), and all these failures (except one in *elasticjob-elastic-job-lite*) are from tests that only fail once.

While it is common for frequently failing flaky tests to exhibit repetitive flaky failures, this trend is not consistent across all projects. For example, within the project *Apache-hbase*, there are 5 flaky tests that failed more than 1,000 times have at least one non-repetitive flaky failure.

We investigated whether specific exception types were associated with these non-repetitive flaky failures. From the dataset we analyzed, among the top 10 most frequently occurring exceptions, two exceptions appeared more frequently in non-repetitive failures than repetitive flaky failures. Specifically, the *RuntimeException* was observed 13 times out of its 22 failure cases, while the *SocketException* was also observed 19 times (out of 31 failures). We found that every failures with the *SocketException* was linked within the *Square-okhttp* project.

We observed that certain test suite runs, especially those with a higher number of failed tests, tend to exhibit repetitive flaky failures across most or all the failed tests. For instance, within the *Apache-ambari*, 47 out of 51 flaky tests consistently failed together and displayed the same failure messages and stacktraces each time they failed, and none of their stacktrace lines contain the test names.

Summary. Flaky failures are often repetitive. This can serve as an indicator for developers: previous flaky failures can be a reference to check if a newly encountered failure is familiar. However, there are *few* cases where a failure is not similar with any previously observed flaky failures. In such situations, a deeper investigation is needed to detect its flakiness. A valuable step in this investigative process involves comparing the failure with flaky failures from other tests, especially when the failure's stacktrace lines do not reference the test itself.

*B. **RQ2: With prior flaky and true failures, is it feasible to use the failure de-duplication to tell if a failure is flaky or true one?***

We investigate if the text-based matching can be used to determine if a failure is flaky or not based on the failure de-duplication. As we consider both flaky and true failures, we use a confusion matrix as follows:

TP: Flaky failures that match at least one flaky failure and do not match any of the true failures.

FN: Flaky failures that match at least one true failure *or* does not match with any of the flaky failures.

FP: True failures that match at least one flaky failure.

TN: True failures that do not match with any of flaky failure.

This evaluation methodology follows our running use-case, where newly observed test failures are either labeled as flaky (and ignored), or triaged to developers for further debugging and analysis. We then evaluate the result of matching using the *Precision* (**P**), *Recall* (**R**),and *Specificity* (**SP**) as follows:

$$\text{Precision } (\mathbf{P}) = \frac{\text{TP}}{\text{TP + FP}} \tag{3}$$

$$\text{Recall } (\mathbf{R}) = \frac{\text{TP}}{\text{TP + FN}} \tag{4}$$

$$\text{Specificity } (\mathbf{SP}) = \frac{\text{TN}}{\text{TN + FP}} \tag{5}$$

We chose these metrics to reflect the use-case of developers encountering a failure and comparing it with historical flaky and true failures. Given a model where developers ignore test failures that are labeled as flaky, a safer approach would have a higher precision, as precision reports the frequency with which an approach falsely determines a test to be flaky. Since we

TABLE III

TEXT-BASED MATCHING TO LABEL FLAKY AND TRUE (NON-FLAKY) FAILURES. The *Total Tests and Failures* column provides the total flaky tests, the number of true (non-flaky) failures across these tests, and the count of flaky failures. The *Set of Failures* column displays the different failures within both flaky and true failures. The *Confusion Matrix and Evaluation By Failures* columns present the matching results between flaky and true failures. The # of Tests in TP and FN shows how many different tests have at least one failure in each category. The cumulative number of tests in *TP* and *FN* might exceed the total given in *Test* because a test might have multiple flaky failures in different categories.

	Total Tests and Failures			**Set of Failures**		**Confusion Matrix and Evaluation By Failures**							**# of Tests in**	
Project	**Tests**	**Non-Flaky**	**Flaky**	**True**	**Flaky**	**TP**	**FN**	**FP**	**TN**	**P**	**R**	**SP**	**TP**	**FN**
Alluxio-alluxio	114	32,795	16,858	6,232	310	9,173	7,685	1,933	30,862	82%	54%	94%	114	109
square-okhttp	99	33,949	26,486	18,546	120	16,517	9,969	107	33,842	99%	62%	99%	58	52
apache-ambari	51	11,045	4,063	4,562	54	4,003	60	5	11,040	99%	98%	99%	50	2
hector-client-hector	33	3,603	6,529	1,769	33	1,382	5,147	12	3,591	99%	21%	99%	32	1
activiti-activiti	30	44,937	1,363	15,863	31	932	431	2,272	42,665	29%	68%	94%	1	29
tootallnate-java-websocket	23	1,116	2,143	418	45	596	1,547	531	585	52%	27%	52%	20	23
apache-httpcore	22	8,021	354	667	22	0	354	2,096	5,925	0%	0%	73%	0	22
apache-hbase	20	585	2,519	185	26	1,209	1,310	162	423	88%	47%	72%	17	5
qos-ch-logback	20	2,614	438	895	21	56	382	368	2,246	13%	12%	85%	3	17
kevinsawicki.http-request	18	387	3,501	229	18	981	2,520	40	347	96%	28%	89%	4	14
wildfly-wildfly	18	4,364	50	1,497	18	38	12	0	4,364	100%	76%	100%	6	12
wro4j-wro4j	14	540	10,833	90	21	800	10,033	29	511	96%	7%	94%	9	11
spring-projects-spring-boot	12	2,150	14	244	13	2	12	0	2,150	100%	14%	100%	1	12
undertow-io-undertow	7	2,306	92	236	12	8	84	943	1,363	0%	8%	59%	2	6
orbit-orbit	7	812	2,943	302	7	87	2,856	57	755	60%	2%	92%	2	5
elasticjob-elastic-job-lite	3	111	7	68	4	4	3	0	111	100%	57%	100%	1	3
doanduyhai-Achilles	2	154	121	86	3	120	1	6	148	95%	99%	96%	1	1
jknack-handlebars.java	1	147	411	61	1	0	411	16	131	0%	0%	89%	0	1
zxing-zxing	1	76	322	37	1	322	0	0	76	100%	100%	100%	1	0
joel-costigliola-assertj-core	1	18	974	10	1	974	0	0	18	100%	100%	100%	1	0
apache-commons-exec	1	59	33	13	1	0	33	2	57	0%	0%	96%	0	1
ninjaframework-ninja	1	120	476	4	1	0	476	8	112	0%	0%	93%	0	1
22 Projects Total	498	149,909	80,530	52,014	763	37,204	43,326	8,587	141,322				323	327

consider scenarios where developers may be most interested in minimizing false positives, we also report specificity, which evaluates the percentage of true failures correctly labeled. Lower recall scores indicate that an approach inadvertently labels more flaky failures as true failures — indicating that a developer might spend more time debugging them.

Table III shows the confusion matrix of using our approach as described in Section III. The performance of the approach varies across projects. For example, there are projects with at least 95% precision (10 out of 22) while some projects with 0% (5 out 10 projects).

We examine the results per-project to gain further insights into the performance of the approach. We find that in projects where the text-based matching approach struggles to differentiate between flaky and true failures, failures are typically presented as *assertion* exceptions. For example: *all* of the false negative (*FN*) flaky failures in *Tootallnate-java-websocket*, *all* of the *FN* flaky failures in *orbit-orbit*, and 98% of the *FN* in *Square-okhttp* are assertion exceptions. In the case of *Alluxio-alluxio*, we found that 90% of the *FN* failures were *NullPointerException*s. Even with the availability of stacktraces in these failures, these exceptions remain challenging to be used in finding the differences between flaky and true failures. On the other side, the projects which have reasonable precision and recall scores (or at least precision scores) like the case in *doanduyhai-Achilles*, there are a verity of different exceptions like *UnknownHostException*, and less likely to have general exceptions such as *assertion* and *NullPointerException*.

We also examined the relationship between the performance of the approach and factors such as the proportion of true failures and the number of times that a test flakes. Examining the table, we see two projects with a comparable ratio between flaky and non-flaky projects: *Apache-httpcore* and *Wildfly-wildfly*. However, we also note that the performance of the approach (particularly in terms of true positives) varies significantly between these two projects. Overall, we do not note any significant correlation between the ratio of flaky to true failures and the performance of the approach.

Examining projects with more than one flaky test, we found the project *Apache-ambari* has the best performance in *precision*, *recall*, and *specificity*. In this project, we found that the majority of the flaky failures failed with the exception *ProvisionException*. As discussed in RQ1 (Section IV-A), the majority of flaky tests in this project failed together, in the same test suite execution. This case is somewhat different from the other projects, in which flaky failures occur in different test runs — it is indeed quite likely that all of the flaky failures have the same root cause.

We also show the number of tests with at least one true positive or false negative failure (the last two columns of Table III). For example, in the case of *Alluxio-alluxio*, we see that of 114 tests in total, all 114 had at least one failure that was correctly classified as a true positive. However, 109 of those 114 tests *also* had at least one failure falsely classified as a false negative (falsely labeled as "not flaky"). These data indicate that naïve approaches that rely on test name to determine whether or not a failure is flaky or not are unlikely to be effective on this dataset.

To gain insight into the value of matching stack traces (in addition to exceptions), we also examined the performance of matching failures *only* using exception type. Table IV presents the top ten most frequently occurring exceptions observed in

TABLE IV
TOP 10 MOST OCCURRENCE EXCEPTION IN FLAKY AND TRUE FAILURES

The *Exception Occurrence* column details the frequency of a specific exception, indicating in how many projects, tests, and failures this exception has been observed. The *Match Result (with Stacktraces)* column displays the match distributions, considering stacktraces and the related test count while the, *Match Result (without Stacktraces)* column indicates match results based on exception types, excluding stacktraces.

	Exception Occurrence					Match Result (with Stacktraces)								Match Result (without Stacktraces)							
						By Failures				By Tests				By Failures				By Tests			
Exception Name	**Projects**	**Tests**	**Failures**	**True**	**Flaky**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**
NullPointerException	21	475	50,427	42,264	8,163	1,168	6,995	711	41,553	40	106	105	475	0	8,163	8,444	33,820	0	120	120	355
AssertionError	21	401	48,751	19,727	29,024	5,977	23,047	4,376	15,351	63	120	94	367	253	28,771	13,191	6,536	4	174	171	223
IOException	7	219	16,959	15,800	1,159	642	517	515	15,285	10	16	13	206	642	517	3,718	12,082	10	16	13	193
UnknownHostException	9	225	9,369	315	9,054	9,052	2	0	315	125	2	0	98	9,052	2	0	315	125	2	0	98
ActivitiException	1	30	9,277	9,205	72	0	72	342	8,863	0	9	7	29	0	72	2,809	6,396	0	9	8	21
IllegalArgumentException	17	393	8,666	8,663	3	0	3	189	8,474	0	3	3	393	0	3	203	8,460	0	3	3	390
AssertionFailedError	7	91	8,644	6,881	1,763	66	1,697	1,598	5,283	1	20	20	88	66	1,697	4,074	2,807	1	20	20	70
NoSuchMethodError	1	1	8,539	0	8,539	8,539	0	0	0	1	0	0	0	8,539	0	0	0	1	0	0	0
PersistenceException	2	30	8,399	8,398	1	0	1	156	8,242	0	1	1	30	0	1	389	8,009	0	1	1	29
WroRuntimeException	1	7	6,520	33	6,487	0	6,487	11	22	0	5	5	3	0	6,487	13	20	0	5	5	2

the analyzed flaky failures. In some cases, the exception by itself cannot determine the differences of matching compared the case when we consider the stacktraces such as the case of *NullPointerException*. However, we see that *UnknownHostException* occurred almost exclusively in the context of flaky failures. However, this certainly does *not* support the conclusion that all instances of this exception in a failure indicate a flaky failure: in three other projects in which this exception occurred, it occurred only in true failures. This suggests that some exceptions could be linked to flakiness within a project, but it is likely not possible to generalize this correlation across projects.

We also discovered some failures where exceptions match both flaky and true failures, as indicated in Table IV. In the context of our experiment, the most frequently occurring exception is the *AssertionError* which roughly 20% of these failures appear in **TP**. However, when considering only the exception type and excluding stacktrace lines, the proportion drops to less than 1%. The reason behind this observation is the generality of the *AssertionError* exception. For example, a test may have multiple assertion statements, and if they fail for different reasons, they match the exception but differ in the stacktrace. Therefore, it becomes challenging to attribute this type of exception to a specific type of failure.

Summary. We found that using the de-duplication approach to find flaky and true failures effective in some projects especially when their failures logs more informative than just assertion failures. For most of the projects, relying on the stacktraces in addition to the exception type is helpful as most failure exceptions could be seen in both flaky and true failures. The approach performs best when failure messages are specific.

C. *RQ3: To what extent is the utilization of machine learning helpful in finding the differences between flaky and true failures?*

We evaluate the Failure Log Classifier using two algorithms (decision tree and Naïve Bayes) and two approaches for balancing the dataset. In terms of balancing the dataset, we use the SMOTE technique if the ratio of one type of failures is less than 10% of the total number of failures of the other type. We also consider training on the dataset as it is without balancing. We use stratified cross-validation and leave one fold for testing purposes.

To further understand the efficacy of machine learning in this context, we looked for a state-of-the-art classifier based on the failure logs. Existing methods to detect flaky failures, like the work of Lampel et al. [22], do not align with our dataset, which is based on the failure logs. Given this and the discussed features, we considered an alternative baseline approach. We utilized TF-IDF to provide a comparison for our classifier's predictions. Furthermore, we investigated whether TF-IDF could serve as an alternative method, especially since the features of the Failure Log Classifier are directly from the syntax of the failure logs without involving dynamic features.

Table V shows the result of using the Failure Log Classifier and TF-IDF in predicting a failure if it is flaky or not. While we considered two different classification algorithms (Decision Tree and Naive Bayes), we find that decisions trees (without any dataset balancing) performed the best, and present only those results here (complete results are available in our public repository [35]). To ensure that each fold had at least one flaky failure, we include only projects with at least 10 flaky failures in this analysis. The relative performance of the Failure Log Classifier and the TF-IDF varies as in some projects they have at least 90% F1 scores with zero *FN* failures while in few projects it is worse than being randomly guessing. The performance of the two classifiers close to each others (78,181 *TP* in the Failure Log Classifier versus 79,428 in TF-IDF *TP*). Both classifiers have less False Positive rates (4,745 in the Failure Log Classifier and 1,437 in TF-IDF) than the rate of using the text-based matching (8,587).

Comparing the significant increase in true positives versus the text-based matching, we find that Failure Log Classifier and TF-IDF both benefit significantly from the ability to match failures from one test to a different test. This is because, in our implementation of text-based matching, we do *not* remove test-specific lines from the stack trace. Future work might extend our approaches to abstract these elements out of the stack trace, making matches between tests more likely [18].

Comparing TF-IDF versus the Failure Log Classifier, we find that TF-IDF performs somewhat better in all cases. One

TABLE V
COMPARISON OF FAILURE LOG CLASSIFIER AND TF-IDF PERFORMANCE ON FLAKY AND TRUE FAILURES PREDICTION.
For the Failure Log Classifier and TF-IDF, we show the confusion matrix, precision (P), recall (R), and F1 score of the overall prediction result. This analysis only includes projects with at least 10 flaky failures (excluding the project "elasticjob-elastic-job-lite").

	Total Flaky Tests and Failures				Failure Log Classifier							TF-IDF						
Project	**Test**	**Failures**	**Flaky**	**True**	**TP**	**FN**	**FP**	**TN**	**P**	**R**	**F1**	**TP**	**FN**	**FP**	**TN**	**P**	**R**	**F1**
Alluxio-alluxio	114	49,653	16,858	32,795	16,014	844	1,706	31,089	90%	94%	92%	16,616	242	555	32,240	96%	98%	97%
square-okhttp	99	60,435	26,486	33,949	26,056	430	897	33,052	96%	98%	97%	26,459	27	104	33,845	99%	99%	99%
apache-ambari	51	15,108	4,063	11,045	4,055	8	481	10,564	89%	99%	94%	4,063	0	6	11,039	99%	100%	99%
hector-client-hector	33	10,132	6,529	3,603	6,529	0	404	3,199	94%	100%	96%	6,529	0	12	3,591	99%	100%	99%
activiti-activiti	30	46,300	1,363	44,937	947	416	300	44,637	75%	69%	72%	1,023	340	67	44,870	93%	75%	83%
tootallnate-java-websocket	23	3,259	2,143	1,116	2,130	13	436	680	83%	99%	90%	2,130	13	437	679	82%	99%	90%
apache-httpcore	22	8,375	354	8,021	315	39	108	7,913	74%	88%	81%	314	40	17	8,004	94%	88%	91%
apache-hbase	20	3,104	2,519	585	2,377	142	24	561	99%	94%	96%	2,386	133	31	554	98%	94%	96%
qos-ch-logback	20	3,052	438	2,614	172	266	104	2,510	62%	39%	48%	235	203	37	2,577	86%	53%	66%
kevinsawicki.http-request	18	3,888	3,501	387	3,498	3	124	263	96%	99%	98%	3,498	3	54	333	98%	99%	99%
wildfly-wildfly	18	3,895	48	3,847	0	48	0	3,847	0%	0%	0%	50	0	0	4,364	100%	100%	100%
wro4j-wro4j	14	11,373	10,833	540	10,833	0	67	473	99%	100%	99%	10,833	0	29	511	99%	100%	99%
spring-projects-spring-boot	12	2,164	14	2,150	6	8	0	2,150	100%	42%	60%	9	5	3	2,147	75%	64%	69%
undertow-io-undertow	7	2,398	92	2,306	3	89	0	2,306	100%	3%	6%	4	88	0	2,306	100%	4%	8%
orbit-orbit	7	3,755	2,943	812	2,943	0	69	743	97%	100%	98%	2,943	0	59	753	98%	100%	99%
doanduyhai-Achilles	2	275	121	154	120	1	0	154	100%	99%	99%	120	1	0	154	100%	99%	99%
joel-costigliola-assertj-core	1	992	974	18	974	0	1	17	99%	100%	99%	974	0	0	18	100%	100%	100%
jknack-handlebars.java	1	558	411	147	411	0	16	131	96%	100%	98%	411	0	16	131	96%	100%	98%
ninjaframework-ninja	1	596	476	120	476	0	8	112	98%	100%	99%	476	0	8	112	98%	100%	99%
zxing-zxing	1	398	322	76	322	0	0	76	100%	100%	100%	322	0	0	76	100%	100%	100%
apache-commons-exec	1	92	33	59	0	33	0	59	0%	0%	0%	33	0	2	57	94%	100%	97%
21 Total Projects	495	229,802	80,521	149,281	78,181	2,340	4,745	144,536				79,428	1,095	1,437	148,361			

explanation for this is the presence of line numbers in the failure messages. As described in Section II, we manually filtered this noisy information out from the text-based matching and Failure Log Classifier— but allowed it to remain for TF-IDF to prioritize. We found including the stacktrace lines numbers added more values as reflecting different stacktraces. Classifiers that use more complex features (particularly those that are specific to a project) might be able to outperform TF-IDF. On the other side, the generality of the features that the Failure Log Classifier could be a reason that, even with high performance in most projects, still not outperform the the TF-IDF.

We note two projects where the Failure Log Classifier performs significantly worse than TF-IDF (*Wildfly-wildfly* and *apache-commons-exec*). In the *Wildfly-wildfly*, we found all flaky failures with had the exception *RuntimeException* with very low repetitive rate (each test failing at most 7 times) while the same exception often appears in the true failures. This project performs well in the TF-IDF and even in the text-based matching. The main observation in these failures is that the line numbers in the tests differ, which is not captured from the features we proposed to train the Failure Log Classifier. In *apache-commons-exec*, we have a similar situation with a frequently-occurring exception, *AssertionFailedError*.

The usability of different machine learning approaches varies based on the specific use case and objectives. If the main goal is to maximize the number of true positives (*TP*) without being overly concerned about the rate of false positives, the approach with more *TP* is the better. In this scenario, the model is more focused on correctly identifying as many flaky failures as possible, even if it means accepting a higher number of false positives. One of the main advantages of the Failure Log Classifier is its flexibility in extending the learned features. The model can be easily augmented with additional static and dynamic features extracted from each failure. The proposed features shown in Table I are not intended to be a final set for immediate adoption, but serve as a starting point. By leveraging additional features (particularly those specific to a project), the failure log classifier can potentially enhance its performance identifying flaky failures.

Summary. We found that both the Failure Log Classifier and TF-IDF are able to predict flaky and true failures in most the projects. We found TF-IDF is slightly better in terms of the total number of false positives and negatives failures compare to the Failure Log Classifier result.

V. THREATS TO VALIDITY

We construct an experiment to evaluate the efficacy of approaches that determines whether or not a test failure is flaky based on matching that failure against previously witnessed failures. This methodology accurately represents cases where developers already have a historical set of failures, and are evaluating new failures as they arrive. However: developers may be more interested in other usecases, which we did not evaluate. In a real-world scenario, flaky failures could be a collected by merging diverse failure logs collected in CI, in different runtime environments, different timeframes, and even different code revisions under testing. Based on data availability, our evaluation methodology collects failures from a single code revision — we hypothesize that our key findings regarding the repetitive nature of flaky failures would generalize, but leave such a study for future work. Our evaluation methodology also assumes that a significant proportion of failures have been labeled as true or flaky failures using existing methodologies. However, developers may not have all the failures being labeled.

We use failures of tests detected during mutation analysis as a stand-in for true regression failures, as we were unable

to prepare a dataset containing both flaky and true regression failures for multiple projects. However, studies have repeatedly demonstrated that mutants are a valid substitute for real faults in many software testing contexts [30]. Moreover, our results are complemented by existing case studies of individual projects like Google Chromium [3] and SAP HANA [18].

Our dataset of tests is drawn from existing work [10], and has been re-used in other recent works as well [36], [37], [13]. This dataset includes projects from different domains, but all projects are implemented in Java. The overall performance of these failure matching approaches may vary between different languages and testing frameworks. We leave an extension of this study to other languages to future work. Based on our results, it is clear that the performance of flaky failure prediction approaches will vary across projects. While our experiment demonstrates a full range of performance of these approaches (from near-perfect performance to extremely poor performance), it is difficult to extrapolate what the "average" case would be. We take care to avoid drawing such conclusions, and instead aim to identify patterns in projects, tests, and failures that may impact the performance of these approaches.

VI. Discussion

By examining the exception types and frequency, we concluded that the causes of flaky failures in our dataset range from code-related issues to environmental factors. Experienced developers can sometimes easily identify if a failure is flaky just by examining its log [21], indicating that certain failures are readily detectable as flaky. Our examination of exception types confirmed that detecting these environmental-related failures can be easy.

However, our evaluation also revealed a category of failures that are *hard* to classify as flaky or not — in particular, failures with an *AssertionFailedError* or *NullPointerException*. Failures with these kinds of exceptions appeared to be fairly low in information: the failure does not provide enough information to determine if it is caused by flakiness or not. One approach to improve flakiness detection for these kinds of failures might be to enhance the tests or system under test. By providing richer logging information about the symptoms that led to the failure, matching-based approaches may be able to discriminate better between flaky and non-flaky failures. Future work might study these failures further.

Using machine learning in addressing this problem may offer another solution. In our study, we use the decision tree algorithm due to its ease of implementation, particularly in datasets with a minimal number of features, with the majority being boolean features. However, it is worth considering other supervised learning algorithms if the user of this approach intends to expand the feature list to include additional details. Mining additional features is likely to improve prediction performance, although we expect it would be challenging to uniformly improve performance across many projects.

VII. Related Work

Detecting Flaky Failures. Rerunning failing tests has been a de-facto approach for developers to determine if a test failure is caused by flakiness, or is a true failure [9], [38]. Bell et al. studied the efficacy of different rerunning strategies, finding that simply re-running failing tests immediately upon observing a failure is often ineffective at confirming that a failure is flaky [11]. They proposed *DeFlaker*, an alternative strategy for determining if a failure is flaky or not by intersecting the line coverage of each failing test with the set of lines that changed since the last test suite execution. If a test fails but does not cover any changed lines, it can be confirmed as flaky without being re-run. While this approach may be elegant, it can be challenging to apply in practice, as it requires developers to use a specialized code coverage instrumentation agent. Moreover, while Bell et al. show that the approach is faster than typical code coverage agents, it still imposes a runtime overhead of up to 12%. In contrast, in this article, we evaluate approaches for determining whether a test failure is caused by flakiness *without* requiring any code instrumentation, and imposing no overhead on test execution.

Concurrent to our work, two other research teams have been working on this same important challenge. Haben et al. empirically demonstrate the importance of determining precisely which failures of flaky tests are true by showing the danger of assuming that all failures of flaky tests are to be ignored [3]. This work relies on training classifiers from the *code* of flaky tests, while we rely on approaches that utilize the *failures* of flaky tests. More similar to our approach, An et al. use abstracted information from error messages and stack traces to determine which test failures are flaky in the SAP HANA database [18]. An et al. abstract failure symptoms using techniques similar to those that we use, for example, removing line numbers and test entry points from stack traces. On SAP HANA, they report a precision of 96% and recall of 76% in detecting flaky failures, results that are comparable to those some of the open-source projects that we evaluated. We believe that these two lines of work are quite complementary, with our work providing a replicable open-source dataset of failures on multiple projects, and An et al's work providing a deep case study of how to successfully apply flaky failure detection in production at a large software company.

Flaky tests often surface when test suite are run in continuous integration (CI) platforms. With CI, each revision of the system under test is automatically built and tested. Flaky tests can be a nuisance by resulting in builds appearing to have "failed" when they would in fact have passed, if not for flakiness. While our approach aims to determine which failures of which tests are caused by flakiness, other approaches have studied this problem at the level of entire CI builds. Lampel et al. study intermittent job failures in the Mozilla CI platform, training models based on various telemetry (including runtime, CPU load and OS version) to determine which failed builds are flaky. In contrast, Olewick et al. utilize a bag-of-words model, extracting a vocabulary from each CI build log in

order to determine which builds failed due to flakiness [39]. An advantage to these approaches is that they are language and platform-agnostic, requiring *no* knowledge of the structure of log files. However, a corresponding disadvantage is that they may achieve lower predictive performance: An et al. conduct an ablation study examining the importance of failure abstraction, finding performance to drop by 50% [18].

Detecting Flaky Tests. A related line of research aims to determine not which test *failures* are flaky, but which tests *could* fail due to flakiness. If tools could inform developers that a test is flaky immediately as the test is being created, then perhaps flaky tests could be avoided all together. One class of approaches aim to detect flaky tests that are flaky due to a particular root cause. For example: *NonDex* proactively detects flaky tests that rely on non-deterministic behavior (such as the order of iteration of an unsorted collection) [17]. *iDFlakies* detects order-dependent flaky tests — those that have flaky failures when executed in different orders [12].

Given the goal of broadly detecting which tests might be flaky (not tied to a specific root cause), a baseline approach is to re-run a test suite many times. Alshammari et al. examined a dataset of 26 open-source Java projects, re-running each test suite 10,000 times to identify which tests could be flaky [10]. They proposed *FlakeFlagger*, a machine learning-based approach to determine which tests might be flaky using a set of features collected while tests run. Other flaky test prediction approaches rely on the *vocabulary* of test methods, with the insight that tests that perform tasks similar to flaky tests are also flaky[40], [14], [36], [37], [13]. Parry et al. extend this body of work by demonstrating how to efficiently combine machine learning-based flaky test detection with test reruns [15]. Our goal is not to detect which *tests* could be flaky, but rather, which *failures* are flaky, assuming that a developer has previously identified a set of flaky test failures.

Test Failure Clustering. We evaluate a text-based matching approach for determining if a test failure is flaky or not. This approach is inspired by work in a closely related area of failure de-duplication. Classic approaches in this field aim to automatically group multiple test failures together by a shared root cause using stack traces and/or failure logs [41], [42], [43], [20], [19]. However, applying this approach to the problem of failure flakiness detection is relatively under-studied. Lam et al. describe a Microsoft-internal tool that suppresses failures of known flaky tests, and suggests that the error message is used as part of that process [16]. However, they provide no evaluation of how often this approach incorrectly suppresses a test failure that should be investigated as a true failure. We advance this field of study by providing a detailed analysis of the efficacy of test failure clustering for the purposes of labeling test failures as flaky or not. We make our entire dataset and pipeline publicly available along with this article, in order to allow other researchers to study more advanced test failure clustering applications for this use-case.

VIII. Conclusion

Using a novel methodology, we constructed a ground-truth dataset of 149,909 true (non-flaky) failures and 80,530 flaky failures from 22 open-source Java projects. Our analysis shows that, even for the same test, there can be multiple flaky failure symptoms, but that a small set of failing stack traces often reoccur. This finding provides strong evidence that heuristic-based approaches that determine whether a failure is caused by flakiness or a true defect can be effective. Our evaluation of three heuristic approaches for determining whether a test failure is flaky or not showed that performance can vary widely between projects, and that TF-IDF is the best approach overall.

Our results show that some projects may be able to adopt these approaches immediately with no (or almost no) false positives. In other projects where the failure logs lack informative details, like failures with the presence of assertions statements, the approaches may not be effective. Increasing the amount of information in test failure logs can greatly improve the performance of automated approaches for de-duplicatign failures, and can further help with manual analysis.

Given the variability between different projects and inherent non-determinism of flaky tests, it will be challenging to create general-purpose solutions for determining whether a failure is flaky or not. Instrumentation-based approaches (such as DeFlaker [8]) can help add important context to otherwise low-information test failures, but deploying them in production can bring challenges. Future work might continue to study the application of these approaches as case studies in single projects (e.g. recent work studying Google Chromium [3] and SAP HANA [18]). We make our entire dataset and experiments available under an open-source license to enable and encourage future research in this important topic.

IX. Data Availability Statement

All analysis notebooks and dataset related to this work are publicly available at [35] [44].

X. Acknowledgement

We thank Kevin Moran and Wing Lam for their valuable discussions related to this work.

REFERENCES

[1] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, oct 2021. [Online]. Available: https://doi.org/10.1145/3476105

[2] 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*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 857–862. [Online]. Available: https://doi.org/10.1145/3236024.3275529

[3] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. L. Traon, "The importance of discerning flaky from fault-triggering test failures: A case study on the chromium ci," *arXiv preprint arXiv:2302.10594*, 2023.

[4] M. Gruber and G. Fraser, "A survey on how test flakiness affects developers and what support they need to address it," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2022, pp. 82–92. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICST53961.2022.00020

[5] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 830–840. [Online]. Available: https://doi.org/10.1145/3338906.3338945

[6] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. Le Traon, "A qualitative study on the sources, impacts, and mitigation strategies of flaky tests," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 244–255.

[7] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in java projects," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 403–413.

[8] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 433–444. [Online]. Available: https://doi.org/10.1145/3180155.3180164

[9] J. Micco, "The state of continuous integration testing @ google," https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45880.pdf.

[10] 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)*. IEEE, 2021, pp. 1572–1584.

[11] 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)*. IEEE, 2018, pp. 433–444.

[12] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, 2019, pp. 312–322.

[13] V. Pontillo, F. Palomba, and F. Ferrucci, "Static test flakiness prediction: How far can we go?" *Empirical Softw. Engg.*, vol. 27, no. 7, dec 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10227-1

[14] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino, "Know you neighbor: Fast static prediction of test flakiness," *IEEE Access*, vol. 9, pp. 76 119–76 134, 2021.

[15] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models," *Empirical Softw. Engg.*, vol. 28, no. 3, apr 2023. [Online]. Available: https://doi.org/10.1007/s10664-023-10307-w

[16] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1471–1482. [Online]. Available: https://doi.org/10.1145/3377811.3381749

[17] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov, "Nondex: A tool for detecting and debugging wrong assumptions on java api specifications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 993–997. [Online]. Available: https://doi.org/10.1145/2950290.2983932

[18] G. An, J. Yoon, T. Bach, J. Hong, and S. Yoo, "Just-in-time flaky test detection via abstracted failure symptom matching," 2023.

[19] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. USA: IEEE Computer Society, 2003, p. 465–475.

[20] 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 *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 712–723. [Online]. Available: https://doi.org/10.1109/ICSE.2017.71

[21] D. Welter, "Preventing Flaky Tests from Ruining your Test Suite — gradle.com," https://gradle.com/blog/prevent-flaky-tests/, [Accessed 29-May-2023].

[22] J. Lampel, S. Just, S. Apel, and A. Zeller, "When life gives you oranges: detecting and diagnosing intermittent job failures at mozilla," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1381–1392.

[23] A. Navada, A. N. Ansari, S. Patil, and B. A. Sonkamble, "Overview of use of decision tree algorithms in machine learning," in *2011 IEEE control and system graduate research colloquium*. IEEE, 2011, pp. 37–42.

[24] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242, no. 1. Citeseer, 2003, pp. 29–48.

[25] L.-P. Jing, H.-K. Huang, and H.-B. Shi, "Improved feature selection approach tfidf in text mining," in *Proceedings. International Conference on Machine Learning and Cybernetics*, vol. 2. IEEE, 2002, pp. 944–946.

[26] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[27] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs. jar: A large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 10–13.

[28] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 339–349.

[29] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "Bears: An extensible java bug benchmark for automatic program repair studies," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE Computer Society, feb 2019, pp. 468–478. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SANER.2019.8667991

[30] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 654–665.

[31] A. Shi, J. Bell, and D. Marinov, "Mitigating the effects of flaky tests on mutation testing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 112–122. [Online]. Available: https://doi.org/10.1145/3293882.3330568

[32] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 449–452.

[33] A. Fernández, S. Garcia, F. Herrera, and N. V. Chawla, "Smote for learning from imbalanced data: progress and challenges, marking the 15-year anniversary," *Journal of artificial intelligence research*, vol. 61, pp. 863–905, 2018.

[34] D. Berrar *et al.*, "Cross-validation." 2019.

[35] A. Alshammari, P. Ammann, M. Hilton, and J. Bell, "Failure log classifiers," https://github.com/AlshammariA/FailureLogClassifiers, 2024.
[36] Y. Qin, S. Wang, K. Liu, B. Lin, H. Wu, L. Li, X. Mao, and T. F. Bissyande, "Peeler: Learning to effectively predict flakiness without running tests," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 257–268.
[37] D. Dell'Anna, F. B. Aydemir, and F. Dalpiaz, "Evaluating classifiers in se research: the ecser pipeline and two replication studies," *Empirical Software Engineering*, vol. 28, no. 1, p. 3, Nov 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10243-1
[38] "Maven Surefire plugin - Rerun failing tests," https://maven.apache.org/surefire/maven-surefire-plugin/examples/rerun-failing-tests.html, 2023. [Online]. Available: https://maven.apache.org/surefire/maven-surefire-plugin/examples/rerun-failing-tests.html
[39] D. Olewicki, M. Nayrolles, and B. Adams, "Towards language-independent brown build detection," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 2177–2188.
[40] 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, 2020, p. 492–502. [Online]. Available: https://doi.org/10.1145/3379597.3387482
[41] K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle, "Finding similar failures using callstack similarity," in *Proceedings of the Third Conference on Tackling Computer Systems Problems with Machine Learning Techniques*, ser. SysML'08. USA: USENIX Association, 2008, p. 1.
[42] 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)*, 2012, pp. 1084–1093.
[43] J. Lerch and M. Mezini, "Finding duplicates of your yet unwritten bug report," in *2013 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 69–78.
[44] A. Alshammari, P. Ammann, M. Hilton, and J. Bell, "Flaky and True Failures Logs to Accompany "230,439 Test Failures Later: An Empirical Evaluation of Flaky Failure Classifiers"," Jan. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10531160