# The Importance of Accounting for Execution Failures when Predicting Test Flakiness

Guillaume Haben
University of Luxembourg
Luxembourg
guillaume.haben@uni.lu

Sarra Habchi
Ubisoft
Canada
sarra.habchi@ubisoft.com

John Micco
Broadcom
USA
john.micco@broadcom.com

Mark Harman
Meta and UCL
United Kingdom
mark.harman@ucl.ac.uk

Mike Papadakis
University of Luxembourg
Luxembourg
michail.papadakis@uni.lu

Maxime Cordy
University of Luxembourg
Luxembourg
maxime.cordy@uni.lu

Yves Le Traon
University of Luxembourg
Luxembourg
yves.letraon@uni.lu

## ABSTRACT

Flaky tests are tests that pass and fail on different executions of the same version of a program under test. They waste valuable developer time by making developers investigate false alerts (flaky test failures). To deal with this issue, many prediction methods have been proposed. However, the utility of these methods remains unclear since they are typically evaluated based on single-release data, ignoring that in many cases tests that *fail flakily* in one release also *correctly* fail (indicating the presence of bugs) in some other, meaning that it is possible for subsequent correctly-failing cases to pass unnoticed. In this paper, we show that this situation is prevalent and can raise significant concerns for both researchers and practitioners. In particular, we show that flaky tests, tests that exhibit flaky behaviour at some point in time, have a strong fault-revealing capability, *i.e.*, they reveal more than 1/3 of all encountered regression faults. We also show that 76.2%, of all test executions that reveal faults in the codebase under test are made by tests that are classified as flaky by existing prediction methods. Overall, our findings motivate the need for future research to focus on predicting flaky test executions instead of flaky tests.

## CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

## KEYWORDS

Software testing, Flaky Tests, ML, Continuous Integration

**ACM Reference Format:**
Guillaume Haben, Sarra Habchi, John Micco, Mark Harman, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2024. The Importance of Accounting for Execution Failures when Predicting Test Flakiness. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3691620.3695261


*ASE '24, October 27-November 1, 2024, Sacramento, CA, USA*

ACM ISBN 979-8-4007-1248-7/24/10…$15.00
https://doi.org/10.1145/3691620.3695261

## 1 INTRODUCTION

It has been widely reported in the industry (in various companies including Meta [15] and Google [29]), that test flakiness is one of the biggest problems that inhibits test efficiency and effectiveness for practicing software engineers and industrial deployment of automated testing technology [3].

Traditionally, a flaky test is regarded as one that can pass on one execution yet fail on another, even though both executions take place in identical execution environments [15, 26, 28].

In this traditional view, flakiness is a property attributed to a *test case* (for each and *all* of its test executions), rather than either a property of the code under test (its non-determinism), or the execution of the test case (its potentially different outcome for the given execution).

Most practical approaches to test flakiness simply execute each test multiple times in the same environment. When a test passes on some and fails on other executions, then the test is marked as flaky. As a consequence of being so-marked, the test (and any signal that might accrue from its subsequent execution) is simply discarded.

Many companies cannot afford the drain on developers' time considering flaky test signals, so it is important to identify such tests and address them or remove them, either through repeated execution or more sophisticated techniques.

Fortunately, there has been a lot of research on test flakiness [33], and companies like Meta [27] and Google [23] use simply effective techniques such as repeated test execution to identify flaky tests.

Researchers have proposed advanced predictive techniques [6, 12, 17, 23, 34] that can detect with a relatively high precision which tests are flaky. However, all of this research and current deployment practice is based on an assumption that it is a *test itself* that is flaky or not, and once a test is marked as flaky, its signal is lost.

What if the test *correctly* fails (indicating a bug) in one execution, but fails due to flakiness in another? Such a test would be marked as flaky (due to the flaky failure), and its signal would be ignored, even in subsequent correctly failing scenarios, thereby needlessly discarding potentially useful regression test signal.

In this paper, we show that this situation is not only possible but also that it is prevalent: through a study of widely available

open-source systems, we reveal that about 1/3 of all encountered regression faults are triggered by tests that exhibited flaky behaviour at some point in time. This is a clearly significant concern for both research and practice in test flakiness, strongly implying that flaky tests are useful as they add testing value.

Based on these findings, we make the case for the alternative approach to flakiness which places the responsibility for flakiness, not on a test itself, but on the execution of a test. We believe it is important for the research community to revise its approach to test flakiness in order to take account of test execution flakiness, rather than nearly attributing flakiness to a test case (and thereby deeming all of its execution signals to be flaky and therefore unreliable). Such a revised research direction will be highly impactful for practitioners and industrial deployment of flakiness management approaches, ensuring far greater test effectiveness.

We provide evidence to support these claims in the paper. More specifically, based on an evaluation on open source systems, we show that approximately *three quarters* of all regression faults would be missed using the current traditional approach to test flakiness. Furthermore, we show that over half of the genuinely fault-revealing failures would be (incorrectly) classified as 'flaky' (merely because the test itself is regarded as flaky) by prediction methods.

By focusing on predicting test execution flakiness, we show that significant improvements can be achieved in flakiness assessment. However, our results also reveal that overall prediction performance for test *execution* flakiness (using currently available techniques) remains relatively low.

Taken together, we believe these results strongly motivate further study on test execution flakiness. We hope the paper will act as a 'rallying cry' to the research community to consider the assessment of test execution flakiness that is better aligned with the needs of the domain of the actual practice. As the results we present here indicate, this would unlock a 4x improvement in regression test effectiveness, with all the consequent impact this would have on system reliability, and regression test efficiency and effectiveness.

In summary, the contributions of our paper are:

- We report results from *a large empirical study on flakiness prediction, based on the Chromium project* – one of the biggest open-source industrial projects – involving 10,000 builds, more than 200,000 unique tests and 1.8 million test execution failures. This is the first study that investigates the difference between flaky tests and flaky test executions.
- We provide empirical evidence on existing prediction methods showing that despite being precise they are far from satisfactory since they undermine the test signal, i.e., they *flag as flaky the majority, approximately 76.2%, of all test executions that reveal faults in the codebase under test* (56.2% due to misclassifications of non-flaky tests and 20% due to correct classification of flaky tests that reveal faults in the particular studied executions).
- We provide empirical evidence that *flaky tests have strong fault-revealing capabilities*, indicating an inherent limitation of existing methods.

# 2 RELATED WORK

## 2.1 Test Flakiness

Flakiness is a known issue in software testing but research studies on this topic have only gained momentum in the past few years [32]. Luo *et al.* [25] conducted the first empirical study on the root causes of flakiness. They analyzed commit fixes from 51 open-source projects and created the first taxonomy of flaky tests. Later on, several studies on test flakiness followed using different settings. Lam *et al.* [19] conducted a study on flaky tests at Microsoft to identify and understand their root causes. They presented *RootFinder*, a framework that helps to debug flaky tests using logs and time differences in their passing and failing runs.

Romano *et al.* [37] analysed User Interface (UI) tests and showcased the flakiness root causes and the conditions needed to fix them. In the same vein, Gruber *et al.* [13] presented a large empirical analysis of more than 20,000 Python projects. They found test order dependency and infrastructure to be among the top reasons for flakiness in those projects.

To help debug, reproduce, and comprehend the causes of flaky tests several tools have been introduced. *DeFlaker* [4] detects flaky tests across commits, without performing any reruns, by checking for inconsistencies in test outcomes with regard to code coverage. Focused on test order dependencies, *iDFlakies* [20] detects flaky tests by rerunning test suites in various orders.

To increase the chances of observing flakiness, Silva *et al.* [39] introduced *Shaker*, a technique that relies on stress testing when rerunning potential flaky tests. Another line of work aims at repairing (automatically) flaky tests. To this end, Shi *et al.* [38] proposed iFixFlakies, a framework that recommends patches for order-dependent flaky tests based on test patterns found in non-flaky tests that exhibit similar behaviour as flaky tests.

## 2.2 Flakiness Detection Methods

While they remain scarce, the recent publication of datasets of flaky tests [4, 13, 20] enabled new lines of work. Prediction models were suggested to differentiate flaky tests from non-flaky tests. King *et al.* [17] presented an approach that leverages Bayesian networks to classify and predict flaky tests based on code metrics. Pinto *et al.* [34] used a bag of words representation of each test to train a model able to recognize flaky tests based on the vocabulary from the test code. This line of work has gained a lot of momentum lately as models achieved higher performances. Several works were carried out to replicate those studies and ensure their validity in different settings [7, 14].

In an industrial context, Kowalczyk *et al.* [18] implemented a flakiness scoring service at Apple. Their model quantifies the level of flakiness based on their historical flip rate and entropy (*i.e.*, changes in test outcomes across different revisions). Their goal was to identify and rank flaky tests to monitor and detect trends in flakiness. They were able to reduce flakiness by 44% with less than 1% loss in fault detection. In our study, we also rely on test history to help with flakiness prediction.

More recently, *FlakeFlagger* [2] has been proposed. It builds a prediction model using an extended set of features from the code under test together with test smells. The research community continue to draw attention to this field by considering other possible

**Table 1: Existing machine learning-based studies aiming at detecting test flakiness. The majority of the techniques focus on detecting flaky tests, while half of the approaches rely on vocabulary features.**

| Study | Model | Feature category | Features | Benchmark | Target | Year |
|---|---|---|---|---|---|---|
| King et al. [17] | Bayesian network | Static & dynamic | Code metrics | Industrial | **Flaky tests** | 2018 |
| Pinto et al. [34] | Random forest | Static | **Vocabulary** | DeFlaker | **Flaky tests** | 2020 |
| Bertolino et al. [5] | KNN | Static | **Vocabulary** | DeFlaker | **Flaky tests** | 2020 |
| Haben et al. [14] | Random forest | Static | **Vocabulary** | DeFlaker | **Flaky tests** | 2021 |
| Camara et al. [7] | Random forest | Static | **Vocabulary** | iDFlakies | **Flaky tests** | 2021 |
| Alshammari et al. [2] | Random forest | Static & dynamic | Code metrics & Smells | FlakeFlagger | **Flaky tests** | 2021 |
| Fatima et al. [12] | Neural Network | Static | CodeBERT | FlakeFlagger iDFlakies | **Flaky tests** | 2021 |
| Pontillo et al. [35] | Logistic regression | Static | Code metrics & Smells | iDFlakies | **Flaky tests** | 2021 |
| Lampel et al. [21] | XGBoost | Static & dynamic | Job execution metrics | Industrial | Flaky failures | 2021 |
| Qin et al. [36] | Neural Network | Static | Dependency graph | Industrial | **Flaky tests** | 2022 |
| Olewicki et al. [31] | XGBoost | Static | **Vocabulary** | Industrial | Flaky builds | 2022 |
| Ackli et al. [1] | Siamese Networks | Static | CodeBERT | Various | **Flaky tests** | 2022 |

features to predict flaky tests, this is the case of Peeler [36] for example, where the authors leveraged test dependency graphs to predict flaky tests.

Less attention has been given to flaky failures or false alert predictions. Herzig *et al.* [16] used association rules to identify false alert patterns in the specific case of system and integration tests that contain steps. They evaluated their approach on Windows 8.1 and Microsoft Dynamics AX.

Olewicki *et al.* investigated the prospect of leveraging vocabulary-based features on logs from failing builds to predict if failures are caused by defects in the code or by other non-deterministic issues including flaky test failures. It is interesting to note that their work focuses on builds and not tests as we do in our study.

Finally, a recent study by Lampel *et al.* [21] presented an approach that automatically classifies jobs by deciding if a job failure originates from a bug in the code or from flakiness. To do so, they relied on features from job executions, *e.g.*, CPU load, and run time. As such they are only concerned with some form of flaky failures and not with the utility of detecting flaky failures in CI, instead of tests as we do in this paper.

Table 1 summarizes the state-of-the-art of flakiness prediction in chronological order. By inspecting the table, we see that most of the studies focus on predicting flaky tests, with just one focusing on flaky test failures. We also notice that static features like code metrics and test smells are often used but features based on vocabulary (*i.e.*, bag-of-words) are the most popular ones. However, none of these studies investigates the utility of these methods in predicting flaky test executions and the impact in terms of lost test signal, *i.e.*,, potentially missed (regression) faults.

Overall, in contrast to previous work that predicts likely flaky tests (to be used by developers the way they see fit), our study investigates their utility as a replacement of test re-runs in relation to the risk of missing fault-triggering test execution failures.

## 3 THE CHROMIUM PROJECT

Started in 2008, with more than 2,000 contributors and 25 million lines of code, the Chromium web browser is one of the biggest open-source projects currently existing. Google is one of the main maintainers, but other companies and contributors are also taking part in its development.

Chromium relies on *LuCI* as a CI platform [40]. It uses more than 900 parallelized builders, each one of them used to build Chromium with different settings (*e.g.*, different compilers, instrumented versions for memory error detection, fuzzing, etc) and to target different operating systems (*e.g.*, Android, Mac OS, Linux, and Windows). Each builder is responsible for a list of builds triggered by commits made to the project. If a builder is already busy, a scheduler creates a queue of commits waiting to be processed. This means that more than one change can be included in a single build execution if the development pace is faster than what the builders can process. Within a build, we find details about build properties, start and end times, status (*i.e.*, pending, success or failure), a listing of the steps and links to the logs.

At the beginning of the project, building and testing were sequential. Builders used to compile the project and zip the results to builders responsible for tests. Testing was taking a lot of time, slowing developers' productivity and testing Chromium on several platforms was not conceivable. A swarming infrastructure was then introduced in order to scale according to the Chromium development team's productivity, to keep getting the test results as fast as possible and independently from the number of tests to run or the number of platforms to test. Currently, a fleet of 14,000 build bots runs tasks in parallel. This setup helps to run tests with low latency and handle hundreds of commits per day [11].

In this study, we focus on testers, *i.e.*, builders only responsible for running tests. They do not compile the project: when triggered, they simply extract the build from their corresponding builder and run tests on this version. At the time of writing, we found 47 testers

running Chromium test suites on distinct operating systems versions. About 200,000 tests are divided into different test suites, the biggest ones being *blink_web_tests* (testing the rendering engine) and *base_unittests* with more than 60,000 tests each.

For each build performed by any tester, we have access to information about test results. The decision process followed by LuCI to determine a test failure outcome in a specific build is to run a test up to 5 times. A test is labelled as *pass* when it successfully passed after one execution. In case of a failure, LuCI automatically reruns the test up to 5 times. If all reruns fail, the test is labelled as *unexpected* and will trigger a build failure. In the remaining, we will be referring to *unexpected* tests as *fault-revealing tests*. If a test passes after having one or more failed executions during the same build, it is labelled as *flaky* and leads to a passing build.

# 4 DATA

## 4.1 Definitions

Some of our definitions are slightly different from the ones used by previous work since Chromium has its specific continuous integration setup. To make things clear, we define the elements we will discuss in this section. We employ the following definitions:

- **Fault-revealing test**: A test that consistently failed after reruns in the same build, revealing a regression fault at some point in the questioned time period.
- **Flaky test**: A test that failed once or more and then passed after reruns in the same build at some point in the questioned time period.
- **Non-flaky test**: A test that never exhibited flakiness behaviour in any of the builds in the questioned time period.
- **Flaky execution failure**: A test execution failure caused by a flaky test at a specific build.
- **Fault-triggering execution failure**: A test execution failure caused by a fault-revealing test at a specific build.

Flaky and Fault-triggering execution failure definitions regard a particular build in question and are mutually exclusive, while the Flaky and Fault-revealing test definitions regard a time period (a set of builds) in question and are not mutually exclusive, i.e., a test can be both fault-revealing and flaky at different points in time.

## 4.2 Data collection

To perform our study, we collected test execution data from 10,000 consecutive builds completed by the Linux Tester by querying the LuCI API. This represents a period of time of about 9 months taken between March 2022 and December 2022.

Table 2 summarises the list of information extracted and computed for all tests executed in all builds. The BUILDID corresponds to the build in which tests were executed. RUNDURATION is the execution time spent to run the test. RUNSTATUS gives information about the run result (*e.g.*, passing, failing, and skipped) and RUNTAGSTATUS returns more precise information about the result of a run depending on the type of test or test suite (*e.g.*, timeout and failure on exit). We retrieved information about the tests' source code by querying Google Git [1]. As builds often handle several commits, we select the revision corresponding to the head of the blame list: the one on which tests were executed. The TESTSUITE is simply the name of the test suite the test belongs to and TESTID is a unique identifier for a test composed of the test suite and the test name (the same test name can be present in different test suites).

All the scripts used to collect the data alongside the created dataset are available in our replication package [2] [3].

**Table 2: Description of our features. Column *Feature Name* specifies the identifiers used in our dataset, while Column *Feature Description* details the features**

| Feature Name | Feature Description |
|---|---|
| buildId | The build number associated with the test execution |
| flakeRate | The flake rate of the test over the last 35 builds |
| runDuration | The time spent for this test execution |
| runStatus | ABORT<br>FAIL<br>PASS<br>CRASH<br>SKIP |
| runTagStatus | CRASH<br>PASS<br>FAIL<br>TIMEOUT<br>SUCCESS<br>FAILURE<br>FAILURE_ON_EXIT<br>NOTRUN<br>SKIP<br>UNKNOWN |
| testSource | The test source code |
| testSuite | The test suite the test belongs to |
| testId | The test name |

## 4.3 Computing the flake rate

The historical sequence of test results is a valuable piece of information commonly used in software testing at scale [18, 23]. We analyse the history of fault-revealing tests and flaky tests by relying on their flakeRate.

This means that for a test $t$ failing (due to flaky or fault-triggering execution failure) in a build $b_n$, we analyse all the builds from a time window $w$ (*i.e.*, from $b_{n-w}$ to $b_{n-1}$) to calculate its flake rate as follows:

$$flakeRate(t, n) = \frac{\sum_{x=n-w}^{n-1} flake(t, x)}{w} \quad (1)$$

[1] https://chromium.googlesource.com/chromium/src/+/HEAD/

[2] Our dataset, experimental data and related code are released by the University of Luxembourg at https://github.com/serval-uni-lu/Importance-of-Discerning-Flaky-from-Fault-triggering-Test-Failures

[3] All data was collected and experiments were performed by the University of Luxembourg

**Table 3: Data collected from the Chromium CI. We used the *Linux Tests* tester, with *10,000* Builds mined over *nine* months. We extracted Passing, Flaky and Fault-revealing tests and their associated Flaky and Fault-triggering execution failures.**

| Tester | Nb of Builds | Period of Time | | Number of Unique Tests | | | Number of Failures | |
|---|---|---|---|---|---|---|---|---|
| | | From | To | Passing | Flaky | Fault-revealing | Flaky | Fault-triggering |
| Linux Tests | 10,000 | Mar 2, 2022 | Dec 1, 2022 | 198,273 | 23,374 | 2,343 | 1,833,831 | 17,171 |

where $flake(t, x) = 1$ if the test $t$ flaked in the build $b_x$ and 0 otherwise. This metric allows us to understand if the flakiness history of a test can help in the flakiness prediction tasks. The test execution history (*a.k.a.* heartbeat) has been used in multiple studies (especially industrial ones [18, 23]) to detect flaky tests. These studies assume that many flaky tests have distinguishable failure patterns over builds and hence can be detected by observing their history. We check whether this assumption also holds in the case of Chromium.

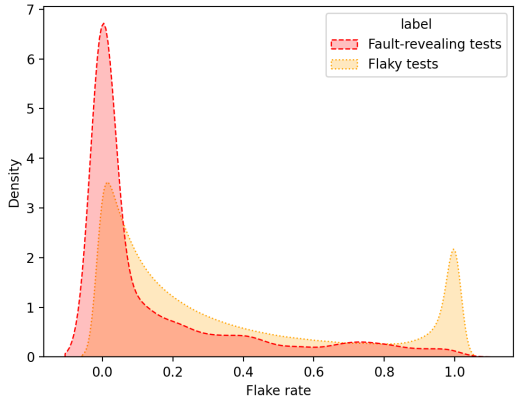


**Figure 1: Flake rate (*x-axis*) for Flaky and Fault-revealing tests. Density (*y-axis*) is the probability density function. The area under curves integrates to one. Many flaky tests are always flaky in their previous builds. A majority of fault-revealing tests have no history of flakiness at all.**

To illustrate the flake rate differences between flaky and fault-revealing tests, we plot the flake rate for both test categories in Figure 1. The flake rate is computed using a window of 35 builds. To set this time window, we checked the number of flaky tests having a $flakeRate() = 0$ for build windows ranging from 0 to 40 builds with a step of 5. We observed a convergence at size 35, meaning that higher numbers of builds do not provide additional information.

In the majority of cases, flaky tests have a history of flakiness: the percentage of flaky tests having a $flakeRate() > 0$ is in fact 87.9%. Furthermore, we see a pike for $flakeRate() == 1$ as 9.5% of flaky tests were always flaky in their 35 previous builds. Still, there is a non-negligible amount (45.3%) of fault-revealing tests that were flaky at least once in previous builds considered: with a $flakeRate() > 0$. From these observations, we may suggest that the flakeRate() can be used to detect flakiness. Nevertheless, there is still an important overlap between the history of flaky tests and fault-revealing tests.

# 5 OBJECTIVES AND METHODOLOGY

## 5.1 Research questions

We start our analysis by assessing the effectiveness of the existing flakiness prediction methods by considering the critical cases where fault-revealing execution failures are flagged as flaky by the prediction methods in various CI cycles. We are interested in investigating the methods' performance under realistic settings, i.e., correctly detected and missed flaky and fault-triggering execution failures, when trained with past CI data and evaluated on future ones. In contrast to previous work, we perform an in-time evaluation that considers test executions at different CI cycles. We investigate the gains and losses of using a prediction method. Thus, we ask:

**RQ1:** How well do flaky test prediction methods discern flaky execution failures from fault-triggering ones?

To establish realistic settings, we train the prediction models using the information available (flaky tests and non-flaky ones, as done by previous work) at a given point in time, where we have sufficient historical data to train on. We then evaluate the models in subsequent builds with respect to flaky and fault-triggering test failures. We replicate the vocabulary-based methods since they are popular, easy to implement and quite effective, and aimed at learning to predict flaky tests, as done by previous studies.

After checking the prediction performance in a realistic setting (execution failures), we repeat the entire process but now we train on historical test execution failures instead of tests. We make this adaptation with the hope of improving further our predictions and perhaps improving our understanding of the impact that such predictions may have on missed fault-triggering execution failures (those marked by the models as flaky). Hence, we ask:

**RQ2:** How well do flaky execution failure prediction methods discern flaky execution failures from fault-triggering ones?

To better understand the interplay between flaky and fault-triggering executions we investigate the distribution of flaky and fault-revealing tests in the different builds in our dataset. We also seek to quantify the critical cases where a test having a history of flakiness happens to reveal a fault. Therefore, we ask:

**RQ3:** How prevalent are flaky tests and fault-revealing tests across builds?

## 5.2 Experimental procedure

*5.2.1 Selection of a flaky test detection approach.* Being a recent topic of interest, several techniques have been introduced in the scientific literature. Approaches relying on code coverage such as FlakeFlagger [2] or DeFlaker [4] are challenging to implement in our case. Chromium's code base consists of several languages and code coverage is both costly and non-trivial to retrieve. Test

smells [6] approaches are also difficult to extract as tests are written in many different languages and tools do not always exist. Having those constraints in mind, we decide to use the vocabulary-based approach introduced by Pinto *et al.* [34]. This approach received significant attention with several replication [7, 14] and follow-up studies and its simplicity makes it easy to implement regardless of the programming languages used in the system under study.

*5.2.2 Training and validation of the existing approaches (RQ1).* We evaluate the ability of the vocabulary-based approach, trained to differentiate flaky from non-flaky tests and used to predict flaky test failures. To do so, we divide our dataset into a training set (containing test information about the first 8,000 builds) and a test set (containing test information about the last 2,000 builds). We train our model following the existing methodologies. The flaky set includes all tests marked as flaky in the training set. The non-flaky set includes all fault-revealing tests and all passing tests in the $8{,}000^{\text{th}}$ build (*i.e.*, the last build of the training set) minus the tests that are found as flaky in any of the builds under study (to increase the confidence of being non-flaky). The test set includes all flaky test failures and all fault-triggering execution failures (reported by fault-revealing tests).

*5.2.3 Implementation of a failure execution classifier (RQ2).* We select flaky failures in our dataset as all failures produced by flaky tests and fault-triggering execution failures are all execution failures produced by fault-revealing tests. There are no duplicated data in the case of test failures, as each test execution is unique. For RQ2, we train our classifier on non-flaky executions (passing and fault-revealing tests execution) and flaky failures.

*5.2.4 Time-sensitive evaluation.* We split our data into two parts: the first 80% builds are selected as a training set and the last 20% as a holdout set. By doing so, we respect the evolution of failures across time and avoid any data leakage that could occur by randomly selecting data. This time-sensitive aspect is very important to consider. We found that not taking this condition into account and training a model on a shuffled dataset would greatly overestimate the performance. Figure 2 shows a representation of our dataset. Flaky tests are present in all builds and Fault-revealing tests are occasional: they happen in 1/4 of builds (See Section 7). To mitigate imbalance, we collected all passing tests for 1 build: $b_{8{,}000}$ and use them in our set of non-flaky tests, for training.

*5.2.5 Classifier selection and pipeline description.* We use a random forest classifier to perform the predictions. Unfortunately, our dataset is imbalanced with the minority class being 1% of the data. Using a simple random forest would greatly increase the chance of having few or no elements from our minority class in the different trees, making the overall model poor in predicting the class of interest. To alleviate this issue, we decide to use a Balanced Random Forest classifier [9] to facilitate the learning. This implementation artificially modifies the class distribution in each tree so that they are equally represented. Furthermore, we use SMOTE in the training phase to augment data for the minority class [8].

To represent the tests, we use *CountVectorizer* to convert texts as a matrix of token counts. This technique, known as bag-of-words, is used in previous vocabulary-based approaches [5, 7, 14, 31, 34]. These vectors initially contain as many features as the words appearing in source code of the tests. As the generated dictionary can become big (in terms of size) we need to use feature selection to reduce it, remove irrelevant features (reducing noise in the data) and select the most informative features. Feature selection is thus, helping to reduce the model training time and to improve the overall performance and interpretability of the model.

We use SelectKBest[22] which retains the $k$ highest score features based on the univariate statistical test $\chi^2$. Hyper-parameters of the machine learning pipeline, *i.e.*, the number of trees in the forests, the sampling strategy for SMOTE and the number of features to be retained are tuned using a grid search approach and cross-validation in the training set. Once optimized, we retrain a model fitted on the whole training set and evaluate it on the holdout set.

*5.2.6 Metrics.* Finally, to evaluate the different models, we rely on the following metrics derived from true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN):

$$\textbf{Precision} = \frac{TP}{TP + FP} \qquad \textbf{Recall} = \frac{TP}{TP + FN}$$

The accuracy of a model is sensitive to class imbalance. In particular, the precision and recall metrics can easily be impacted when one class is underrepresented. To alleviate this issue, we report the Matthews Correlation Coefficient (MCC) which is a more reliable statistical rate to avoid over-optimistic results in the case of an imbalanced dataset [10]. This metric takes into consideration all four entries of the confusion matrix. MCC ranges from -1 to 1 and is given by the following formula:

$$\textbf{MCC} = \frac{TN \times TP - FP \times FN}{\sqrt{(TN + FN)(TP + FP)(TN + FP)(FN + TP)}}$$

In addition to those metrics, we also report the false positive rate (FPR), that is, the ratio of fault-triggering execution failures misclassified as flaky over all fault-triggering execution failures. It is defined by:

$$\textbf{FPR} = \frac{FP}{FP + TN}$$

## 5.3 Data Labeling

This subsection summarises the way data is labelled. As shown in Figure 2, our dataset consists of a training set and a test set containing information about the first 8,000 builds and the last 2,000 builds under study, respectively.

*5.3.1 Training set.* Existing prediction methods focus on differentiating flaky tests from non-flaky tests. In our case, we label as flaky all tests marked as flaky in a build and as non-flaky tests all failing tests and all passing tests minus tests that were found to be flaky in previous builds. Duplicated tests (based on their source code) are removed. Note that we added and kept passing tests in the training set to increase the performance of each model.

*5.3.2 Test set.* The test set consists of all flaky failures (positive elements) and all fault-triggering execution failures (negative elements). There are no duplicated elements as each test execution has a unique set of dynamic properties (run duration and flake rate).

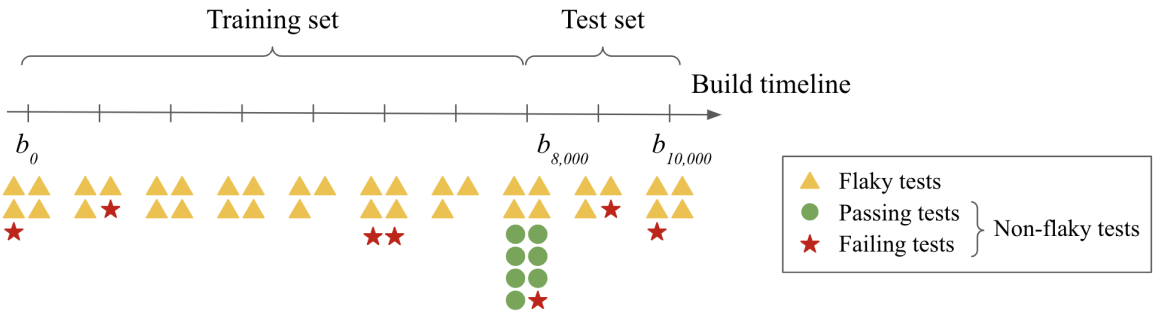


**Figure 2: The data collected from Chromium's CI consists of flaky, fault-revealing and passing tests spread across 10,000 builds. The build timeline ranges from build $b_0$ to $b_{10,000}$ and depicts the distribution of the collected tests: flaky tests are spread across all builds and fault-revealing tests happen occasionally. Due to a large number of passing tests, we collected them from the $b_{8,000}$ build (*i.e.*, at the end of our training set).**

# 6 RESULTS

## 6.1 RQ1: Discerning flaky from fault-triggering execution failures when training on tests

We trained a model on 69,159 passing tests, 910 fault-revealing tests and 8,857 flaky tests (unique tests). Then, we evaluated it on 217,503 flaky failures caused by flaky tests and 2,320 fault-triggering execution failures caused by fault-revealing tests. Table 4 reports the obtained performance. Similar to the performance achieved by previous vocabulary-based models on other datasets, our model was able to reach high accuracy with a precision of 99.2% and a recall of 98.9%. However, we note a high false-positive rate. This is due to an important amount (76.2%) of fault-triggering execution failures classified as flaky (FP). This is concerning: fault-triggering execution failures should not be misclassified as they reflect the existence of real faults. Overall, the MCC value is equal to 0.20, which is relatively low and shows that the model struggles (compared to random selection) to identify fault-triggering execution failures.

**Table 4: Vocabulary-based model performance for the prediction of *flaky test failures vs fault-triggering execution failures* when trained on flaky vs non-flaky (fault-revealing and passing tests). Despite a high accuracy on flaky execution failures, the low MCC and high FPR show us that it remains challenging for the model to classify negative elements (in our case: fault-triggering execution failures)**

| Precision | Recall | MCC | FPR |
|---|---|---|---|
| 99.2% | 98.9% | 0.20 | 76.2% |

The confusion matrix of our model decisions is depicted in Figure 3. The x-axis reports the predicted label and the y-axis the actual label. Correct classifications are displayed in the top left (TN) and bottom right (TP). We clearly observe that the model is able to detect flaky tests with high precision. We also see that 2,435 flaky

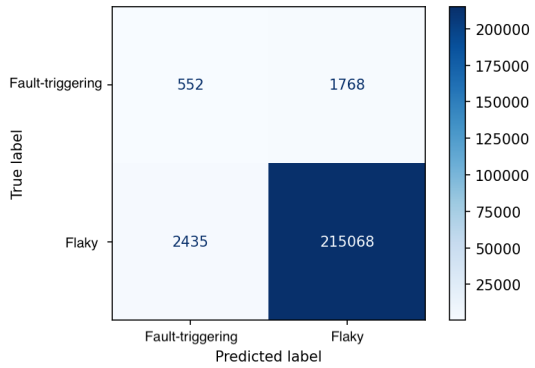


**Figure 3: Confusion matrix for the vocabulary-based model. High accuracy is reached similar to the performance reported in previous works. Nonetheless, 1,768 (76.2%) out of the 2,320 fault-triggering execution failures are mislabeled as flaky.**

tests are classified as non-flaky (FN). This number is also important to consider: it translates in all cases where developers will be required to investigate irrelevant failures.

We want to further understand the reasons behind the classification of fault-triggering execution failures. Therefore, we analyse the (fault-revealing) tests causing those failures. Out of the 2,320 fault-triggering execution failures, 1,768 are in the set of false positives (76.2%) among which we found 463 (20% of all fault-triggering execution failures) whose tests have a history of flakiness (flakeRate > 0) and 1,305 (56.2% of all fault-triggering execution failures) without flakiness history. Here it must be noted that depending on the size of the history considered, we may have more tests with past flakiness or less. Overall in our data, 1/3 of all fault-triggering executions are due to tests that have exhibited flakiness behaviour.

**RQ1:** We find accurate predictions when aiming at flaky tests. However, a high proportion (76.2%) of all fault-triggering execution failures is classified as flaky (missed faults) and still an important number (2,435) of flaky tests are marked as fault-triggering execution failures (false alerts).

## 6.2 RQ2: Discerning flaky from fault-triggering executions when trained on test failures

RQ1 showed that a vocabulary-based model trained to detect flaky tests would miss an important number of faults and will incur false alerts despite having high accuracy. In RQ2 we aim at checking whether we can improve the performance of recognising fault-triggering execution failures by training on test failures (flaky and non-flaky).

Table 5 reports the results of such a model. We notice a high precision and recall, 99.7% and 91.3% respectively, when predicting flaky execution failures. The false positive rate is 20.3%, a better result than in RQ1. However, the MCC only slightly increased to 0.25 showing that classifying failures accurately still remains a challenge.

**Table 5: Vocabulary-based model performance for the prediction of *flaky execution failures vs fault-triggering execution failures* when training on flaky vs non-flaky (fault-triggering and passing test executions). The approach yields better performance but is still insufficient to be reliable.**

| Precision | Recall | MCC | FPR |
|---|---|---|---|
| 99.7% | 91.3% | 0.25 | 20.3% |

**RQ2:** When training on test failures to predict if a test failure is flaky or fault-triggering, model performance slightly improves but is still not effective in the context of the Chromium CI.

## 6.3 RQ3: How prevalent are flaky tests and fault-revealing tests across builds?

RQs 1 and 2 showed that a vocabulary-based model trained to detect flaky tests would miss an important number of faults and will incur false alerts despite having high accuracy. To better understand the situation, in this RQ we aim at providing insights on the interplay between flaky and fault-revealing execution of the tests that are at some point in time flaky.

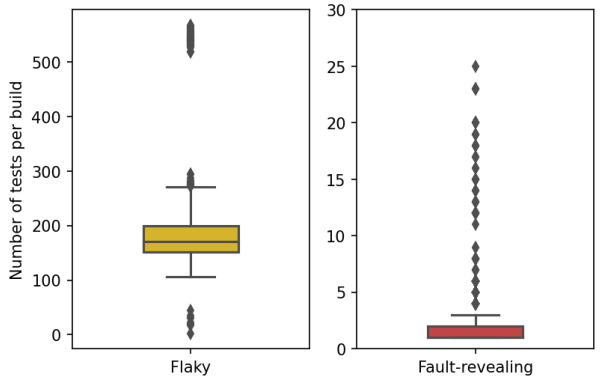


**Figure 4: Number of flaky tests and fault-revealing tests per build. On average, there are 250 flaky tests per build and 1 fault-revealing test per failing build.**

Figure 4 shows the distribution of flaky tests and fault-revealing tests in the studied builds. We observe that there is an average of 178 flaky tests per build with a low standard deviation (41), showing that flakiness is prevalent in the Chromium CI. In the case of fault-revealing tests, taking into account all builds would result in an average number of tests close to 0 as a majority of builds are exempt from them. Thus, for better visualisation, we only considered builds containing at least one fault-revealing test (*i.e.*, failing builds). The average number of fault-revealing tests per failing build is 2.7. The standard deviation for fault-revealing tests is 14.9 and the number of fault-revealing tests reported in one build is up to 579.

**Table 6: Number of builds containing each studied test type. All builds contain flaky executions. ¼ contain fault-revealing tests. Among the failing builds, ¾ contain only fault-revealing tests that are flaky in other builds.**

| Builds containing | Number |
|---|---|
| Flaky tests | 10,000 |
| Fault-revealing tests | 2,415 |
| Fault-revealing flaky tests | 1,974 |
| Faults revealed only by fault-revealing flaky tests | 1,766 |

Table 6 records, for each type of test, the number of builds that contain at least one instance of the studied test types. We note that all builds contain at least one flaky test (a test that flaked during this build). In Chromium CI, execution failures do not cause build failures. That is, tests flaking within the build are ignored.

Developers are expected to investigate test failures only when they occur consistently across 5 reruns (resulting in a fault-revealing test). Such fault-revealing tests occur in 24.2% of the builds (i.e. in 2,415 builds). Interestingly, 1,974 of these builds (i.e. 81.7%) contain fault-revealing tests that flaked in previous builds, indicating that *tests with a flake history should not be ignored in future builds*. In 1,766 builds *all* fault-revealing tests have flaked in some previous builds, indicating that no "reliable" tests identified the fault(s).

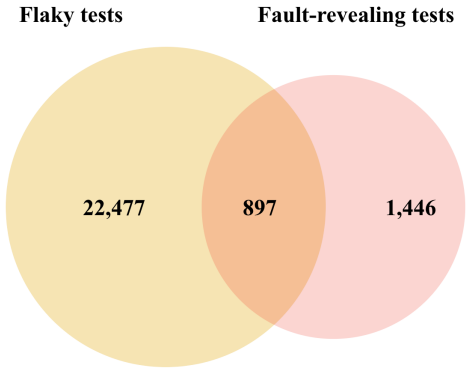


**Figure 5: Distribution of tests in our dataset. 22,477 tests are exclusively flaky among all builds. 2,343 tests are fault-revealing, among which ⅓ are flaky in other builds.**

By investigating the status of all tests across all builds – see Figure 5. Among the 209,530 tests of the Chromium project, 24,820 have failed in at least one build, including 22,477 that were always flaky. Thus, 2,343 tests were fault-revealing in at least one build, i.e., they attested the presence of faults, 897 were also flaky in at least one other build. That is, *38.3% of tests that have been useful to detect faults have also a history of flakiness.*

**RQ3** Flakiness affects all Chromium CI builds and mixes critical (fault-revealing) signals with false (flakiness) signals. Crucially, 81.7% of builds contain fault-revealing tests that were flaky in some previous builds, and 38.3% of all tests flake in some builds and reveal faults in other builds. This indicates the strong ability of (previously detected) flaky tests to reveal faults.

# 7 DISCUSSION

## 7.1 Tests being both flaky and fault-revealing

We further investigate the characteristics of the 897 tests that were found both flaky and fault-revealing in different builds (See RQ3). Figure 6 plots the number of times the examined tests revealed a fault (y-axis) and flaked (x-axis) for each test. We observe that flakiness is dominant, as most of the tests flake significantly more frequently than triggering faults (833 out of 897 points fall below the $y = x$ line). Indeed, most of the tests reveal faults only once, with very few exceptions that reveal faults multiple times. This indicates that developers cannot predict a fault-triggering execution given the test execution history since only a few tests revealed a fault in past builds.

To quantify these numbers, Figure 7 displays the number of tests identified as fault-revealing and flaky in less (or equal) than 1, 2, 5, 10 and 50 different builds. Interestingly, we see that half of the 897 tests (50.4%) were fault-revealing in only one build. This means that it is unlikely that a developer would consider these tests as fault-revealing based on their history as they alsmost always are flaky. Indeed, a majority of the examined tests were found to be flaky in at least 50 different builds (40.5% were flaky in less than 50 builds).

## 7.2 Industrial Focus

In large industrial organisations, a strong testing culture is promoted through well-designed automation infrastructures and a prominent DevOps culture. This means that every software engineer is expected to write his own tests [24]. Test results and code changes information are stored in databases keeping years of execution history. This data is then decisively used for conducting accurate regression test selection and prioritisation [23] in order to provide fast feedback and increase the development pace.

Unfortunately, flakiness has major implications for such techniques [23] requiring particular attention on the transitions of tests (from Pass to Fail or vice-versa)[29] instead of merely tests. Interestingly, at Google 84% of such transitions originates from flakiness despite only 16% of tests having some level of flakiness[30]. Apple also emphasises the value of test transitions for flakiness detection as they use it as a strong indicator in their flakiness scoring system based on the flip rate of test results[18].

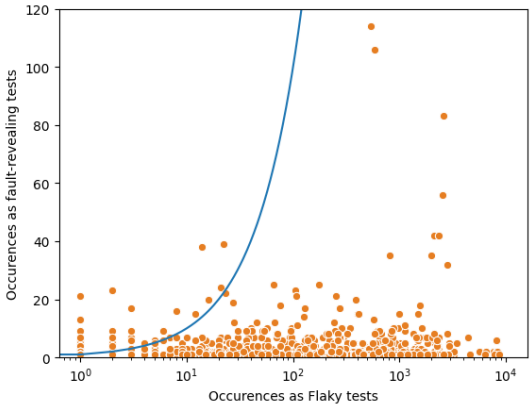


**Figure 6: Number of times a test revealed a fault (y-axis) and flaked (x-axis) for each of the 897 tests that are both flaky and fault-revealing. Note that the scale is logarithmic for the x-axis as the number of flaky occurrences is larger compared to fault-revealing ones. The line y=x is represented in blue.**

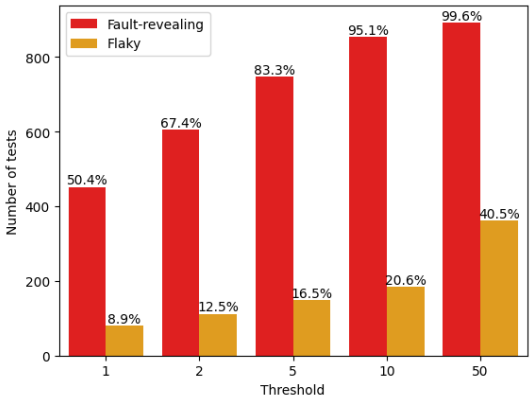


**Figure 7: Fault-revealing and flaky occurrences of the tests that are both flaky and fault-revealing. X-axis records the maximum number of builds that a test was found fault-revealing/flaky. The y-axis records the number of tests.**

All in all, this shows that industries rely more heavily on test transitions rather than test alone (name, id or source code) to get valuable information about tests' health. In such contexts, test results (especially negative) are non-blocking in development pipelines and the key is to get high-confident signals as quickly as possible that will then feed probabilistic or AI-driven systems to assist developers taking actions (or not)[27]. This strong focus on test transitions comforts the message and findings of our study: future research should focus their efforts and attention on test executions rather than test alone.

Another recurrent concern regards the confidence of the provided feedback. In the DevOps culture that is followed by most of software companies, developers are the ones who decide what and how to do stuff, so low confidence or wrong signals quickly lead to abandonment of the tools/technologies. This means that it is imperative to provide quick and highly confident signals to developers to have some chance of adoption.

Perhaps the most important point related to flakiness prediction regards the time when related feedback should be provided. In particular, it is worth developing methods that would provide feedback across multiple release/commit cycles, for cases where there is a high number of cycles, instead of directly for every single run. This way the system could accumulate information over time, provide confident suggestions, and make useful suggestions to their users following their release/commit cycles.

Researchers should also consider revising their evaluation metrics when dealing with flakiness prediction methods. These should be evaluated in an over-time fashion, which takes into consideration the utility of the signal, and its value, and the related development process. Considering single-release data is certainly a good first step, however insufficient for CI-related processes that have multiple release cycles every day, and thus are more interested in the state changes (e.g., test transitions) of the codebase.

# 8 THREATS TO VALIDITY

## 8.1 Internal validity

The main threat to the internal validity of our study lies in the use of vocabulary-based approaches as predictors of flaky tests and failures. Approaches leveraging other features, *i.e.*, dynamic, static, or both, could perform differently. As explained in section 5.2.1, many features are difficult to extract in the case of Chromium (*e.g.*, test smells or test dependency graphs) and features relying on code coverage are not considered due to the overheads they introduce and the difficulty of instrumenting the entire codebase. Although this limits our feature set, the same situation appears in many major companies such as Google and Facebook. Nevertheless, our key insight is that many regression faults are discovered by flaky tests, meaning that they would have been missed even by any flaky test detector that correctly considers them as flaky.

Most of the research on flakiness prediction focuses on classifying tests. Although in this paper we highlight the need for —and focus on— detecting failures, one may wonder what would be the performance of the studied techniques when aiming at detecting flaky tests (instead of flaky test failures). To this end, we trained a model using our dataset to distinguish flaky from non-flaky tests and found similar results with those reported by the literature, i.e., MCC 0.77 when shuffling data and MCC 0.52 when performing a time-sensitive evaluation. The above result shows that the problem of targeting flaky tests is easier and more predictable. However, as shown by our analysis it is misleading as more than 2/3 of the regression faults are missed by such methods.

We evaluate the performance of the predictors in assisting developers, in a sense replacing the decisions of the developers (on whether test failures are due to faults or due to flakiness), which may not reflect the actual case. This is because developers may guess better than the studied predictors. Unfortunately, there are clear limits to what we can achieve through an empirical study as the one we conduct here that does not involve users. While we simply admit that this is a potential threat to our study, we would argue that developers are not in a position to predict whether their tests fail due to faults, as they have no information of fault-triggering from the history of their test cases for more than 50% of the cases.

## 8.2 External validity

We show that detecting flaky tests (instead of failures) is harmful as it can miss many regression faults. This is the case for the Chromium project and, while we believe Chromium to be representative of other software systems, we cannot guarantee that findings would generalise to other projects. Similarly, the performance of the different models we report may vary depending on the project. Here, we mainly focus on web/GUI tests and flakiness might have different causes in HTML and Javascript testing compared to other programming languages.

## 8.3 Construct validity

We assume that all fault-revealing tests in our dataset indeed reveal one or several issues in the code. This is the information reported by the Chromium CI as of today. It is possible that some fault-revealing tests are actually flaky tests as they might not be executed in a sufficient amount of time. In a sense, the cases that reruns consistently failed but did not reveal a regression fault are mistakenly considered as fault-revealing. However, reruns cannot guarantee that a test is not flaky. As this information is currently used by Chromium's developers and further verification is non-trivial, we rely on it as ground truth for our dataset. Passing tests used as non-flaky tests could also be mislabeled in our dataset. However, there is a consequent number of passing tests and it is unlikely that many would actually be flaky. Furthermore, we also removed from the set of passing tests any tests that were found to be either flaky or fault-revealing in any of the 10,000 builds.

# 9 CONCLUSION

In this paper, we investigated the utility of existing vocabulary-based flaky test prediction methods as a replacement for test reruns, in the context of a continuous integration pipeline. To do so, we collected data about 23,374 flaky tests and 2,343 fault-revealing tests composing a dataset of 1.8 million test failures representing the actual development process of more than 10,000 builds corresponding to a period of 9 months. Thus, we empirically evaluated the prediction methods and found similar performance compared to previous studies in terms of precision and recall. Despite the (very) high accuracy to detect flaky test failures, we also found that 76.2% of fault-triggering test failures were misclassified as flaky by the prediction methods, indicating major losses on the fault revelation capabilities of the test suites. Going a step further, we showed that flaky tests have a strong ability to detect faults, with 1/3 of all regression faults being revealed by tests that have experienced flaky behaviour at some point in the project's lifetime.

These findings motivated the need for execution-focused prediction methods. To this end, we extended our analysis by checking the performance of execution-focused models (trained on test executions instead of tests) and found that they resulted in similar accuracy and fewer false positives. Therefore, we believe that additional research is needed in order to tackle this vastly ignored problem of flaky test failure prediction over flaky tests.

# 10 ACKNOWLEDGEMENTS

This work is supported by the Luxembourg National Research Funds (FNR) through the CORE project grant C20/IS/14761415/TestFlakes.

# REFERENCES


[1] Amal Akli, Guillaume Haben, Sarra Habchi, Mike Papadakis, and Yves Le Traon. 2022. Predicting Flaky Tests Categories using Few-Shot Learning. https://doi.org/10.48550/ARXIV.2208.14799

[2] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting flakiness without rerunning tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, IEEE, Madrid, Spain, 1572–1584.

[3] Saswat Anand, Antonia Bertolino, Edmund Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Jenny Li, Phil McMinn, and Hong Zhu. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (August 2013), 1978–2001.

[4] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*. ACM Press, New York, New York, USA, 433–444.

[5] Antonia Bertolino, Emilio Cruciani, Breno Miranda, and Roberto Verdecchia. 2020. Know Your Neighbor: Fast Static Prediction of Test Flakiness. *Proceedings of the International Conference on Software Engineering (ICSE)* 9 (2020), 76119–76134. https://doi.org/10.32079/ISTI-TR-2020/001.Istituto

[6] Bruno Camara, Marco Silva, Andre Endo, and Silvia Vergilio. 2021. On the Use of Test Smells for Prediction of Flaky Tests. In *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing* (Joinville, Brazil) *(SAST '21)*. Association for Computing Machinery, New York, NY, USA, 46–54. https://doi.org/10.1145/3482909.3482916

[7] B. Camara, M. Silva, A. T. Endo, and S. Vergilio. 2021. What is the Vocabulary of Flaky Tests? An Extended Replication. In *2021 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) (ICPC)*. IEEE Computer Society, Los Alamitos, CA, USA, 444–454. https://doi.org/10.1109/ICPC52881.2021.00052

[8] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[9] Chao Chen, Andy Liaw, Leo Breiman, et al. 2004. Using random forest to learn imbalanced data. *University of California, Berkeley* 110, 1-12 (2004), 24.

[10] Davide Chicco and Giuseppe Jurman. 2020. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC genomics* 21, 1 (2020), 1–13.

[11] Chromium contributors. 2021. The Chromium Projects. https://www.chromium.org/. (Accessed on 08/17/2021).

[12] Sakina Fatima, Taher A. Ghaleb, and Lionel Briand. 2022. Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests. *IEEE Transactions on Software Engineering* 2022 (2022), 1–17. https://doi.org/10.1109/TSE.2022.3201209

[13] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An empirical study of flaky tests in python. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, IEEE, Online, 148–158.

[14] Guillaume Haben, Sarra Habchi, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2021. 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)*. IEEE, IEEE, Madrid, Spain, 219–229.

[15] Mark Harman and Peter O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis (keynote paper). In $18^{th}$ *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2018)*. IEEE, Madrid, Spain, 1–23.

[16] Kim Herzig and Nachiappan Nagappan. 2015. Empirically Detecting False Test Alarms Using Association Rules. *Proceedings - International Conference on Software Engineering* 2 (2015), 39–48. https://doi.org/10.1109/ICSE.2015.133

[17] Tariq M King, Dionny Santiago, Justin Phillips, and Peter J Clarke. 2018. Towards a bayesian network model for predicting flaky automated tests. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, IEEE, Lisbon, Portugal, 100–107.

[18] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. 2020. Modeling and Ranking Flaky Tests at Apple. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice* (Seoul, South Korea) *(ICSE-SEIP '20)*. Association for Computing Machinery, New York, NY, USA, 110–119. https://doi.org/10.1145/3377813.3381370

[19] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *Proceedings ofthe 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. ACM Press, Beijing, China, 101–111.

[20] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, IEEE, Xian, China, 312–322.

[21] Johannes Lampel, Sascha Just, Sven Apel, and Andreas Zeller. 2021. 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1381–1392. https://doi.org/10.1145/3468264.3473931

[22] Scikit learn developers. 2022. sklearn.feature_selection.SelectKBest — scikit-learn 1.1.2 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html. (Accessed on 08/11/2022).

[23] Claire Leong, Abhayendra Singh, Mike Papadakis, Yves Le Traon, and John Micco. 2019. Assessing transition-based test selection algorithms at Google. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, Helen Sharp and Mike Whalen (Eds.). IEEE / ACM, Montréal, Canada, 101–110.

[24] Jeff Listfield. 2017. Google Testing Blog: Where do our flaky tests come from? https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html. (Accessed on 01/12/2021).

[25] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Vol. 16-21-November-2014. Association for Computing Machinery, Hong Kong, 643–653. https://doi.org/10.1145/2635868.2635920

[26] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In $22^{nd}$ *International Symposium on Foundations of Software Engineering (FSE 2014)*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne Storey (Eds.). ACM, Hong Kong, China, 643–653.

[27] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, IEEE/ACM, Montreal, QC, Canada, 91–100.

[28] Atif M. Memon and Myra B. Cohen. 2013. Automated testing of GUI applications: models, tools, and controlling flakiness. In $35^{th}$ *International Conference on Software Engineering (ICSE 2013)*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, San Francisco, CA, USA, 1479–1480.

[29] Atif M. Memon, Zebao Gao, Bao N. Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In $39^{th}$ *International Conference on Software Engineering, Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, Buenos Aires, Argentina, 233–242.

[30] John Micco. 2017. The State of Continuous Integration Testing Google.

[31] Doriane Olewicki, Mathieu Nayrolles, and Bram Adams. 2022. Towards Language-Independent Brown Build Detection. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2177–2188. https://doi.org/10.1145/3510003.3510122

[32] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 17 (oct 2021), 74 pages. https://doi.org/10.1145/3476105

[33] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2022. A Survey of Flaky Tests. *ACM Trans. Softw. Eng. Methodol.* 31, 1 (2022), 17:1–17:74. https://doi.org/10.1145/3476105

[34] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the vocabulary of flaky tests?. In *Proceedings of the 17th International Conference on Mining Software Repositories*. IEEE, Seoul, South Korea, 492–502.

[35] Valeria Pontillo, Fabio Palomba, and Filomena Ferrucci. 2021. Toward Static Test Flakiness Prediction: A Feasibility Study. In *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution* (Athens, Greece) *(MaLTESQuE 2021)*. Association for Computing Machinery, New York, NY, USA, 19–24. https://doi.org/10.1145/3472674.3473981

[36] Yihao Qin, Shangwen Wang, Kui Liu, Bo Lin, Hongjun Wu, Li Li, Xiaoguang Mao, and Tegawendé F. Bissyandé. 2022. Peeler: Learning to Effectively Predict Flakiness without Running Tests. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. ICSME, Limassol, Cyprus, 257–268.

[37] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An Empirical Analysis of UI-based Flaky Tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, IEEE, Madrid, Spain, 1585–1597.

[38] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies : A Framework for Automatically Fixing Order-Dependent Flaky Tests. In *27th ACM Joint European Software Engineering Conference and Symposium on the Foundations ofSoftware Engineering (ESEC/FSE '19)*. ACM, TALLINN, ESTONIA, 545–555. https://doi.org/10.1145/3338906.3338925

[39] Denini Silva, Leopoldo Teixeira, and Marcelo d'Amorim. 2020. Shake it! detecting flaky tests caused by concurrency with shaker. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, IEEE, Adelaide, Australia, 301–311.

[40] The Chromium Development team. 2021. chromium/chromium: The official GitHub mirror of the Chromium source. https://github.com/chromium/chromium. (Accessed on 07/06/2021).