A Dataset of Reproducible Flaky-Test Failures

SUZZANA RAFI*, George Mason University, USA
MAHBUB-UL-HOQUE SUMON*, Bangladesh Election Commission, Bangladesh
MD ERFAN, University of Alabama, USA
MARUF MORSHED KHAN, Ministry of Finance, Bangladesh
AUGUST SHI, The University of Texas at Austin, USA
WING LAM, George Mason University, USA

Flaky tests pass and fail non-deterministically when run on the same version of code. Previous research proposed techniques to detect, debug, and repair different categories of flaky tests. However, reproducing the flaky-test failures remains a major challenge due to their inherent non-determinism. Reliably reproducing flaky-test failures is essential for helping both developers and automatic techniques to debug and repair flaky tests. Many datasets related to flaky tests exist to help researchers study them, but these datasets are often composed of disjoint information, where each dataset provides some unique information over the others, such as flaky tests of many different categories, failure logs of flaky tests, or flaky tests reported by developers vs. flaky tests found by automated tools. Furthermore, several of them are missing the key aspect of providing a means to reliably reproduce the flaky-test failures. In this work, we aim to create a dataset of flaky tests, where each test's failure is reproducible and there is a comprehensive set of information for the flaky test.

Compared to prior flaky-test datasets, our dataset is the first to provide (1) an environment to compile the code for running the flaky tests, (2) scripts to run the tests to reproduce the flaky-test failures, (3) scripts to automatically apply flaky-test fixes and check that the test is no longer flaky, and (4) execution logs of the flaky test both passing and failing. We present ReproFlake, a dataset of 1115 flaky tests, spread across four different flaky-test categories. We also publish the guideline we developed to construct our dataset so others can contribute to this dataset by collecting the same information. We also study the categories of flaky tests in ReproFlake, the location and size of flaky-test fixes, and the code coverage of flaky tests before and after they are fixed. Our study highlights promising future flaky-test research directions, which our dataset helps enable.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Flaky tests, dataset, regression testing

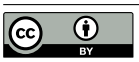
ACM Reference Format:
Suzzana Rafi, Mahbub-Ul-Hoque Sumon, Md Erfan, Maruf Morshed Khan, August Shi, and Wing Lam. 2026. A Dataset of Reproducible Flaky-Test Failures. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA125 (October 2026), 21 pages. https://doi.org/10.1145/3832216

1 Introduction

Developers rely on regression testing, the practice of rerunning tests after every code change, to ensure their code changes do not break existing functionality. One major problem in regression

*Both authors contributed equally to this research.

Authors' Contact Information: Suzzana Rafi, George Mason University, Fairfax, VA, USA, srafi@gmu.edu; Mahbub-Ul-Hoque Sumon, Bangladesh Election Commission, Dhaka, Bangladesh, mahbubsumon085@gmail.com; Md Erfan, University of Alabama, Tuscaloosa, AL, USA, merfan@crimson.ua.edu; Maruf Morshed Khan, Ministry of Finance, Dhaka, Bangladesh, mmk301@gmail.com; August Shi, The University of Texas at Austin, Austin, TX, USA, august@utexas.edu; Wing Lam, George Mason University, Fairfax, VA, USA, winglam@gmu.edu.



ACM 2994-970X/2026/10-ARTISSTA125
https://doi.org/10.1145/3832216

testing is the presence of *flaky tests* [30, 51], which are tests that can both pass and fail when run on the same version of code. When a test suite has flaky tests and a test fails, developers are often misled to believe that the failure indicates a bug was introduced in their recent code changes, but this test could have flakily failed regardless of any changes. Many developers have reported flaky tests as one of their biggest problems [6–8, 14, 15, 19, 30, 34, 37, 39, 41, 43, 44, 53–55, 61, 64, 67].

Prior work on developing techniques to mitigate flaky tests also resulted in various flaky-test datasets [10, 20, 23, 25, 26, 30, 36, 45, 51]. Although many datasets related to flaky tests exist, they differ in the types of information they provide (e.g., flaky tests of many different categories [10, 20, 45], failure logs of flaky tests [23], or flaky tests reported by developers [20, 30, 51] vs. flaky tests found by automated tools [10, 23, 26, 45]). For example, one popular flaky-test dataset is the International Dataset of Flaky Tests (IDoFT) [10, 45]. IDoFT includes names of flaky tests detected using automated tools, along with the project the tests are from, the version in which the tests were detected, the category of the flaky test based on the tool that detected the flaky test, and information related to whether the test has been fixed or not. Since its creation, there has been much work that uses this dataset to evaluate new approaches [32, 33, 46–49, 56, 58]. However, IDoFT does not have flaky tests reported by developers, nor the failure logs associated with a flaky-test failure, which other datasets have [20, 23, 30, 51]. In contrast, IDoFT also has pull request links to fixes, which other datasets may not have.

We propose ReproFlake, a new dataset of flaky tests obtained from developer issue reports as well as flaky tests collected in prior research (IDoFT) but with enriched information. While IDoFT provides categorized flaky tests and links to pull requests, it lacks compilable and reproducible environments, failure artifacts (e.g., test failure logs and test reports), and issue reports. ReproFlake is the first dataset of flaky tests focused on providing an environment to reproduce the flaky-test failures. For each flaky test in this dataset, we provide (1) an environment (including Java and Maven versions along with project dependencies) to reliably compile the code and test, (2) scripts to run the test and reproduce the flaky-test failure, (3) the code changes accepted by developers that fix the flaky test, and (4) other information of the flaky test, such as execution logs of the test passing and failing and developer-written issue reports that describe the flaky test (when available). Each flaky test in ReproFlake is confirmed to be flaky with automated tools and therefore includes failure and fix information. Our dataset also includes four categories of flaky tests, which encompass most of the categories in prior datasets, and flaky tests that are reported by developers along with those found only by automated tools.

We construct ReproFlake by combining flaky tests from two sources: developer-reported issues in Jira and previously collected flaky tests from IDoFT. We manually inspect Jira issue reports to identify true flaky tests and retain only Java and Maven-based unit tests, to maintain consistency with the flaky tests taken from IDoFT. For both Jira and IDoFT tests, we require clearly identifiable commit SHAs representing the flaky and fixed versions of the tests. We also filter out tests that we cannot compile and run on the corresponding versions. We then reproduce flaky-test failures using automated scripts and package successfully reproduced tests into Docker environments.

ReproFlake enables researchers to tackle many open challenges related to flaky-test reproduction, analysis, and repair. By providing flaky and fixed versions of flaky tests, scripts that reliably reproduce flaky-test failures, and a reproducible environment to compile and rerun flaky tests, we allow researchers to better study flaky-test behavior. Similar to how other datasets, such as Defects4J [40] and BugSwarm [62], enable various automatic program repair techniques that leverage bug reports [24, 42, 50] and the reproducibility of test failures in the datasets, ReproFlake can enable researchers working on automatic repair of flaky tests to learn fix patterns from buggy/fixed code pairs and bug reports. Furthermore, the failure-related information offers a ground truth for log analysis of flaky execution or categorization. Researchers building tools for

Table 1. Key differences between our ReproFlake dataset and other flaky-test datasets. ✓ and ✗ denote that the dataset does and does not, respectively, have such information.

Dataset	Categories	Flaky Version	Fixed Version	Reproducible Environment	Error Logs & Surefire Reports	Pull Request	Issue Report
IDoFT [10]	ID, NIO, NOD, OD, TD, TZD	✓	✓	✗	✗	✓	✗
FlakeFlagger [23]	Does not categorize	✓	✗	✗	✓	✗	✗
FlakyFix [33]	ID	✓	✗	✗	✗	✗	✗
DeFlaker [26]	Does not categorize	✓	✗	✗	✗	✗	✗
FlakyCat [20]	ID, IO, Network, Randomness, TD	✓	✗	✗	✗	✗	✗
Barbosa et al. [25]	ID, Network, OD, Platform, Randomness, Resources, TD	✗	✗	✗	✗	✗	✓
ReproFlake	ID, NIO, OD, TD	✓	✓	✓	✓	✓	✓

flaky-test fault localization, categorization, or automated test repair can use our dataset to train, test, and evaluate their ideas on real-world flaky tests, all in an environment that enables flaky-test failure reproducibility. ReproFlake enhances the current state-of-the-art flaky-test datasets with bug reports, flaky and fixed code, Docker environments, failing and passing execution logs and reports, and pull request details. Table 1 highlights the key differences between existing flaky-test datasets and ReproFlake.

Beyond constructing ReproFlake, we also use it to understand (1) what are the categories of flaky tests in our dataset, (2) where are code changes to fix flaky tests located and how large are these changes, and (3) how deterministic is the code coverage of flaky tests and how do flaky-test fixes affect code coverage? Our findings are primarily relevant to researchers and developers working on flaky-test reproduction and repair. For example, by analyzing flaky-test fix locations, our results inform them where repair efforts should be prioritized (e.g., focusing on test code over code under test for certain categories of flaky tests). We also inspect and categorize the flaky-test failures and find that the majority of the failures come from assertion errors. While just knowing that a failure comes from an assertion error does not help with determining the category of a flaky test, we do observe other categories of failures that occur less often, but are more strongly correlated with specific flaky-test categories. Future work can further analyze the use of errors to categorize flaky tests to aid in the debugging [20] and fixing [28] of flaky tests.

This paper makes the following main contributions:

- We use publicly available issue reports and the existing IDoFT dataset to create ReproFlake, a new dataset containing 1115 flaky tests spread across four different flaky-test categories.
- ReproFlake consists of a Docker environment to compile code under test and tests, along with scripts to reproduce the flaky-test failure for each flaky test.
- We provide a set of systematic guidelines for how we and others should determine whether a given flaky test is reproducible from issue reports.
- We present an empirical study using ReproFlake on the location and size of flaky-test fixes and the code coverage of flaky tests before and after they are fixed. Our results provide insights on how future flaky-test research can better aid developer's debugging and fixing of flaky tests and our dataset is publicly available [17].

2 Background

When a developer runs tests after making code changes and observes test failures, the failures should signal that their recent code changes introduced a fault. However, some tests can fail non-deterministically, even on the same version of code. Such tests are known as *flaky tests* [30, 51].

Understanding and reproducing flaky-test failures often requires a clear categorization of their causes and familiarity with the tools that can detect and reproduce them. In this section, we first

introduce the main categories of flaky tests in our dataset and then present the tools that we use to detect and reproduce such categories of flaky tests.

- **Implementation-Dependent (ID):** These tests assume deterministic outcomes from APIs whose specifications are actually non-deterministic, such as iteration order over data structures like `HashMap`, whose order is not guaranteed [59].
- **Non-Idempotent-Outcome (NIO):** These tests fail when they are run in the same environment (e.g., JVM) more than once; the test passes when run the first time in the environment but fails when run again [63]. These failures arise because the test leaves behind or modifies shared state (heap, file system, etc.), so running it again without resetting that state leads to a different (failing) result. Although the same test is normally executed only once, NIO behavior provides an early indication that the leftover state may later cause order-dependent test failures.
- **Order-Dependent (OD):** These tests' pass/fail outcomes depend on the order in which they are run [45, 66]. Other tests, known as **OD-relevant tests (ODR)** [56, 60], change some shared state (e.g., global variables) between the ODR and OD test, thereby influencing the pass/fail outcome of the OD test.
- **Timing-Dependent (TD):** These tests will pass or fail depending on the interleaving of threads or whether asynchronous computations finished or not [57, 58].

Flaky-test reproduction and analysis tools. Several tools support the detection and reproduction of different categories of flaky tests.

- **NonDex [35, 59] for ID tests:** NonDex identifies APIs with non-deterministic specifications, such as those with uncontrolled iteration order, and changes their output to something different but valid, such as randomizing the iteration order. If a test fails when run with a different order of its output, then it is flaky.
- **iDFlakies [45] for NIO and OD tests:** iDFlakies detects OD tests by running test suites in different orders. Given passing and failing orders, prior work used delta-debugging to identify the corresponding OD-relevant test [45, 65]. iDFlakies was later extended to detect NIO tests by running the same test multiple times in the same JVM [63].
- **Docker [5]:** To provide a reliable environment for reproducing flaky-test failures, we utilize Docker images. We can configure a Docker image to contain the relevant operating system as well as source code and dependencies needed for execution. Our dataset includes Docker images that others can use to reproduce flaky-test failures in specific runtime environments.

3 Example Flaky Test

We identified a Timing-Dependent (TD) flaky test for our dataset while examining issue report *COLLECTIONS-812* [3] from the Apache Commons-Collections project. The flaky test is shown in Figure 1a, and the developer's fixed version is shown in Figure 1b. This test compares the behavior of the `save` method when invoked on an empty, immutable `EmptyProperties` object defined in Apache Commons-Collections compared to the standard JDK implementation, `java.util.Properties`. In Figure 1a, the `save` method writes a String `comments` (Line 2) and the current date and time (recorded to the nearest second) in the code under test to the provided `OutputStream` in Line 4 and `PrintStream` in Line 15. Assume that the `OutputStream` object `actual` at Line 3 is written at some time T_0 by `EMPTY_PROPERTIES.save(...)` on Line 4, and the `OutputStream` object `expected` is written at some other time T_1 by `new Properties().save(...)` on Line 15. If T_0 and T_1 happen within the same second, the test passes; if they happen in different seconds, the assertion on Line 16 fails, because it compares the entire `OutputStream`, which includes the current date and time. The pass/fail outcome thus depends solely on the wall-clock timing of the `save` calls.

```
1 @Test public void testSave() throws IOException {
2   final String comments = "Hello world!"; {
3   try (ByteArrayOutputStream actual = new ByteArrayOutputStream()) {
4     PropertiesFactory.EMPTY_PROPERTIES.save(actual, comments);
5     try (ByteArrayOutputStream expected = new ByteArrayOutputStream()) {
6       PropertiesFactory.INSTANCE.createProperties().save(expected, comments);
7       String expectedComment = getFirstLine(expected.toString("UTF-8"));
8       String actualComment = getFirstLine(actual.toString("UTF-8"));
9       assertEquals(expectedComment, actualComment, () ->
10        String.format("Expected String '%s' with length '%s'", expectedComment, expectedComment.length()));
11      expected.reset();
12      // Added to simulate flakiness and ensure the test fails
13      // try { Thread.sleep(2000); } catch (InterruptedException e) { }
14      try (PrintStream out = new PrintStream(expected)) {
15        new Properties().save(out, comments);
16        assertArrayEquals(expected.toByteArray(), actual.toByteArray(), expected::toString);
17    } catch (UnsupportedEncodingException e) { fail(e.getMessage(), e); }
18  }
19 }
```

(a) Example TD flaky test.

```
20 @Test public void testSave() throws IOException {
21   final String comments = "Hello world!";
22   try (ByteArrayOutputStream actual = new ByteArrayOutputStream();
23        ByteArrayOutputStream expected = new ByteArrayOutputStream()) {
24     PropertiesFactory.EMPTY_PROPERTIES.store(actual, comments);
25     PropertiesFactory.INSTANCE.createProperties().store(expected, comments);
26     String expectedComment = getFirstLine(expected.toString("UTF-8"));
27     String actualComment = getFirstLine(actual.toString("UTF-8"));
28     assertEquals(expectedComment, actualComment, () ->
29       String.format("Expected String '%s' with length '%s'", expectedComment, expectedComment.length()));
30     expected.reset();
31     // Added to simulate flakiness in fixed version, but test should still pass
32     // try { Thread.sleep(2000); } catch (InterruptedException e) { }
33     try (PrintStream out = new PrintStream(expected)) { new Properties().store(out, comments); }
34     String[] expectedLines = expected.toString(StandardCharsets.UTF_8.displayName()).split("\\n");
35     String[] actualLines = actual.toString(StandardCharsets.UTF_8.displayName()).split("\\n");
36     assertEquals(expectedLines.length, actualLines.length);
37     assertEquals(expectedLines[0], actualLines[0]);
38   }
39 }
```

(b) Developer fixed version of the TD flaky test.

Fig. 1. Example of a TD flaky test and the developer fixed version of it from Apache Commons-Collections. The red-commented lines are added by us to more frequently reproduce the flaky-test failure. These lines make the failure deterministic in the flaky version, while no flaky-test failures occur in the fixed version.

Rerunning the flaky test multiple times does not lead to the same failure for every run, making it difficult to observe the failure for debugging later. However, if we insert a delay, `Thread.sleep(2000)` at Line 13, before the call to `new Properties().save(...)`, we can ensure that $T_1 \geq T_0 + 2\,\text{s}$, so the times always differ by at least two seconds. As such, the TD flaky test now fails 100% of the time, making the failure a deterministically reproducible one.

In Figure 1b, we show how the developer later fixed the flaky test by changing the assertion to check that the two outputs have the same number of lines and that only the first line matches (Lines 36– 37). Now, even if we introduce a delay at the same location (the red-commented `Thread.sleep(2000)` on Line 32), the test no longer fails, regardless of the time between T_0 and T_1.

Our dataset includes this flaky test, along with all the code and information needed to help a user compile and run this test, for both the flaky version (where the test could fail), as well as the fixed version (where the test should pass). To help one reproduce the flaky-test failure, we also include the changes needed to reliably reproduce the failure, namely the inserted `Thread.sleep` delay as a `patch` file to help make this test fail. In other words, for each flaky test in our dataset, we include up to four code versions: Flaky, Fixed, FlakyCodeChange (e.g., Flaky version with `Thread.sleep`

added to reproduce the failure), and FixedCodeChange (e.g., Fixed version with `Thread.sleep` added to confirm that the failure no longer occurs).

4 ReproFlake Dataset and Framework

We present ReproFlake [17], a dataset of reproducible flaky tests. We curate the dataset with flaky tests from issue reports posted on Jira from popular open-source projects, and a popular flaky-test dataset, IDoFT [10]. All flaky tests we obtain from Jira issue reports and IDoFT are from Java Maven-based projects. Our dataset consists of zip archives that provide Docker images containing the environments to reproduce flaky-test failures across multiple categories.

4.1 Dataset Structure and Framework

For the same Maven project and a module contained within[1], different flaky tests may exist at different versions of that code. ReproFlake maintains a separate zip archive representing each version of a Maven module that contains at least one flaky test. If a version of a module contains multiple reproducible flaky tests, we still provide only one zip archive that includes everything needed for all flaky tests in that module version. The zip archive contains the following information to help reproduce flaky-test failures:

- Source code of the project at the version when the tests in the module were found to be flaky, so that we can run the flaky version to reproduce the failures.
- The patch file, `Fixed.patch`, which a user can apply to the flaky version of the project to get the developer fixed version. The developer fixed version contains the fixed flaky test. If multiple flaky tests exist for a particular module version, there will be one `Fixed.patch` file for all flaky tests.
- The patch file, `FlakyCodeChange.patch`, which a user can apply to the existing flaky version, changing it into a version with a higher rate of failure. This patch file is relevant only for TD flaky tests as other categories have deterministic failure reproduction scripts. If multiple TD tests exist for a particular module version, there will be one `FlakyCodeChange.patch` file for all TD tests.
- The patch file, `FixedCodeChange.patch`, which a user can apply to the fixed version of the code to see if it still fails after the `Fixed.patch` has been applied. This patch file is relevant only for TD flaky tests. If multiple TD tests exist for a particular module version, there will be one `FixedCodeChange.patch` file for all TD tests.
- The Maven dependency repository (i.e., the `.m2` directory), which contains all dependencies needed to compile and run the test on the flaky code version and on the fixed version of the module. If the module version contains multiple flaky tests, we still use the same `.m2` directory as dependencies rarely change from our patches.
- An information file containing metadata for the flaky test, including the GitHub commit SHA, the issue report and pull request link (when available).

We provide a framework to automatically compile and run the test, and to reproduce the flaky-test failure. Our framework will use our zip archives and run the tests in Docker environments to generate a "results" directory containing the result of each test. The folder contains the test run logs, test reports from running through the Maven Surefire test runner plugin, and a summary file that summarizes the results (number of passes and failures) from multiple test runs.

[1]In Maven, a module is a directory within a project that contains its own test suite and is built and tested as a separate unit.

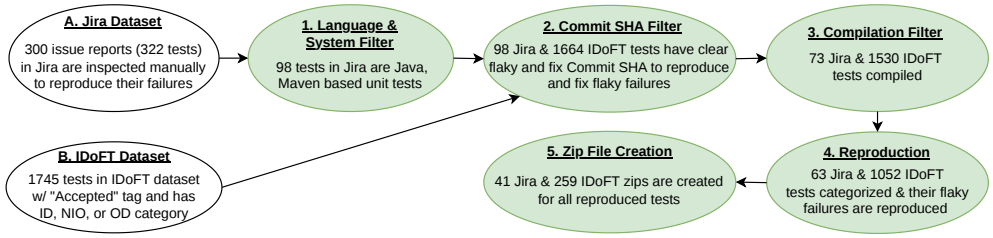


Fig. 2. Overview of how the ReproFlake dataset is created.

4.2 Data Collection

Our dataset involves flaky tests from two data sources: (a) Jira Issue Reports [12] and (b) IDoFT dataset [10]. Jira is a widely used issue-tracking platform, especially for Apache projects, whereas IDoFT (International Dataset of Flaky Tests) is a popular dataset that provides a large and diverse collection of real-world flaky tests identified from open-source GitHub projects. In Figure 2, we can see that A and B represent the two data sources, respectively. We curate our dataset from these sources by following the steps outlined in the figure. The process starts with collecting data from Jira issue reports and IDoFT dataset, followed by filtering out tests based on language, checking for relevant information availability, constructing the compilation steps for the code, developing scripts for reproducing the flaky-test failures, and creating the final zip archive.

4.2.1 Jira Issue Report. To obtain flaky-test-related issue reports from developers, we mined Jira for issue reports that explicitly mention flaky behavior using the keywords “intermittent”, “flaky”, “flak” in the title, description, or comments. We used the Jira Python library [13] to query issue reports from all Apache projects on Jira. Our search resulted in 10,808 issues. The complete list of issues we obtained from Jira is available online on our website [17]. Among the issues we obtained from Jira, we randomly selected around 3% of the issue reports (300) and distributed them among four reviewers for manual inspection.

Dataset Creation Guideline. To create a reliably reproducible dataset of flaky tests, we created and followed a comprehensive issue inspection guideline. All reviewers independently followed the guideline, updating the guideline as needed (in which case all reviewers were notified and updated their labeling accordingly). We make our guideline available online [9] so future work on flaky tests can follow it to create additional reproducible flaky-test datasets.

To ensure consistency and reliable reproducibility, we followed a five-step process, represented in Figure 2 and described in detail in the guideline. The workflow bubbles in the figure illustrate the main steps of the guideline. In some cases, a single bubble may consist of multiple steps of the guideline, but the section number always corresponds to the bubble number. For example, Language and System Filter in Bubble 1 maps to sections 1.1 and 1.2 in our guideline.

Bubble 1 in the workflow checks whether an issue involves a Java–Maven flaky test. Bubble 2 identifies the relevant commits and test information (i.e., module name, fully qualified test name), Bubble 3 handles compilation, Bubble 4 categorizes and reproduces flaky tests, and Bubble 5 creates and validates the final data artifacts.

1. Language & System Filter. We first verified whether each issue corresponds to a Java/Maven-based project and a flaky test. Reviewers inspected the project repository to confirm the presence of a POM file (the build configuration file needed for a Maven project) and Java source code. We analyzed the issue report description, comments, and pull request to determine whether the issue

involves flaky unit-test behavior rather than integration or system-level test. We exclude any issues that do not pass this check.

2. Commit SHA Filter. For valid Java/Maven-based flaky tests, we identified the developer-fixed merged commit and its immediate preceding commit, which we treated as the flaky version. To locate these commits, reviewers searched for the issue ID in the project's GitHub repository, and examined associated pull requests and commits. When a merged pull request was available, the merge commit was treated as the fixed version, and its immediate predecessor as the flaky version.

3. Compilation Filter. We attempted to compile each module corresponding to the flaky/fix commit using Maven. We later categorized the compilation errors to understand the main causes of exclusion (Section 4.3).

4. Reproduction. Differences in flaky-test categories affect the methodology needed to reproduce their flaky-test failures. We described details about the flaky-test category and tools to detect them in Section 2. Below, we summarize the approaches used to reproduce each category of flaky test we came across while mining the Jira issue reports.

Implementation-Dependent (ID) Tests: ID tests are reproduced by exposing non-deterministic behavior in APIs or execution environments with tools, such as NonDex [16], which explores different allowed behaviors of under-determined Java APIs (like `HashMap` iteration order) and reports tests that fail under these variations. For the failure run, we store the NonDex seed that caused the failure, which represents the specific randomization configuration that caused the failure, allowing the same behavior to be reproduced deterministically in later runs. We use this stored seed during the flaky-test failure reproduction to guarantee deterministic failure reproduction.

Order-Dependent (OD) Tests: OD tests can be reproduced in two ways:

- **Unknown OD-relevant test:** If we do not know the OD-relevant test, we may use tools like iDFlakies [45], which runs the tests in a test suite in random orders to expose flaky-test failures. As iDFlakies relies on a customized test runner, which experienced some incompatibilities with a few JUnit tests in our dataset, we instead made small changes to the Maven Surefire plugin (the built-in plugin for running JUnit tests in Maven projects), customizing it to run tests in specific orders. We first gather all tests in a module and execute them in 20 randomized orders using our custom Maven Surefire plugin. If a test shows non-deterministic results (at least one pass and one fail), we then rerun it in both one failing and one passing order five times each. If the test consistently fails in the failing order and passes in the passing order, we confirm it as an OD test.
- **Known OD-relevant test:** If the OD-relevant test is known from the report or pull request, we reproduce the flakiness by running the OD test and its OD-relevant test in the specific execution order required for the OD test to fail, using our custom Maven Surefire plugin.

Timing-Dependent (TD) tests: TD tests are those that exhibit non-deterministic behavior due to variations in execution timing, such as asynchronous operations, race conditions, or thread scheduling. We tried to simulate the exact behavior that makes the test flaky by inserting delays either in the test code or the code-under-test (CUT). The major challenge here is where to add this delay. We manually inspected the issue report along with its comments, the pull request, the changed code to resolve the flakiness, the test code, code under test, and other associated code of this test. Based on the inspection, we tried adding delays of 500 milliseconds (via adding calls to `Thread.sleep(500)`) manually in different portions of the code, typically targeting locations near the code where timing or synchronization could affect the test result, such as before assertions or around asynchronous operations. If adding the delay caused a substantial increase in the failure rate out of 10 runs compared to runs without the delay, we consider the test to be a TD test. We

then increased the delay by 100 milliseconds in each iteration until the test failed in all 10 runs, ensuring deterministic reproduction of flakiness in our environment. If increasing the delay did not make the test fail deterministically, we do not label it as a TD test.

While we start by using the category-specific flaky-test steps to reproduce the failure, if the flaky-test failure still cannot be reproduced reliably, we will run all other steps. If none are successful, we execute the test 100 times. Tests that fail intermittently are labeled Unreliably Reproducible (with a presumed category when available, or Unclassified otherwise), while tests that never fail are marked as Unreproducible. We do not attempt to reproduce NIO tests from Jira issue reports as these tests are usually not flaky yet, and we did not find any issue report for such tests.

5. Zip File Creation. To build zip archives for flaky tests, we first prepare the project by cloning the repository and creating directories for the flaky and fixed versions. Then we construct patches to capture differences across versions, including `Fixed.patch` for the fixed version, `FlakyCodeChange.patch` for a more-flaky version of TD flaky tests, and `FixedCodeChange.patch` for the fixed version to check if the flakiness persists. We generate patch files by computing the source-code differences between the `Flaky` and `Fixed` versions. We also cache the local Maven dependencies for all code versions in a single directory called `Flakym2`, along with the information about the flaky test and patches with code changes. The test is then executed multiple times to collect various statistics. We package all information into a ZIP archive and add it to our dataset for easy reproduction.

Our framework uses a CSV file and the ZIP files to run a specific flaky test on a specific module version. This run can be configured to be for the flaky version, fixed version, or flaky version but the failure should be deterministic. To use our framework for such runs, the user will need to provide a CSV file where each row contains the flaky-test name, category, project and module information, corresponding ZIP and folder information, its OD-relevant test (for OD tests), its seed (for ID tests), the number of iterations to run, the version to run, and Java version to use to compile the project.

To ensure consistency across inspections when creating the ReproFlake dataset, we maintain a structured CSV log of all relevant information. Each row corresponds to a flaky test and includes information about the reproduction steps, such as the issue and commit details, compilation status, presumed flaky-test category from the report, steps to reproduce the failure, results from multiple runs (including statistics from 100 runs), failure rates, final detected category, and supporting artifacts like patch files, NonDex seeds, test orders, and reproducibility status.

4.2.2 IDoFT Dataset. We also reproduced flaky tests from the International Dataset of Flaky Tests (IDoFT) [10], because it provides a large and diverse collection of real-world flaky tests, representing various categories of flakiness observed and fixed in well-known, Java and Maven-based open-source projects. The methodology for including flaky tests from IDoFT into ReproFlake is illustrated in Figure 2, starting from Bubble B. Unlike the Jira dataset, the IDoFT dataset (i) exclusively contains flaky tests from Maven projects, allowing us to skip the Language and System Filter step (Bubble 1), (ii) already provides the flaky-test category, so the Reproduction step does not need to categorize tests (Bubble 4), (iii) includes only tests associated with accepted pull requests, and (iv) contains Non-Idempotent-Outcome (NIO) flaky tests.

Inspection Methodology. We filtered 1745 flaky tests labeled with *Accepted* status, where the pull requests containing the fixes were accepted. We do the commit SHA filtering step slightly differently than for tests from the Jira data source. As the IDoFT dataset already contains a link to an accepted pull request, we directly take the fixed version of the flaky test from the pull request, and we use its preceding commit as the flaky version, without requiring issue-based commit searches. Unfortunately, we are unable to confidently identify the commit SHA for some flaky tests as the

Table 2. Summary of flaky-test outcomes from Jira and IDoFT datasets.

Outcome	Reason / Category	# Tests
Jira Dataset		
Included	Implementation-Dependent (ID)	14
Included	Order-Dependent (OD)	4
Included	Timing-Dependent (TD)	36
Included	Unclassified (UD)	9
Included Total		**63**
Excluded	Compilation Error	25
Excluded	Not Reproducible / Others	234
Excluded Total		**259**
IDoFT Dataset		
Included	Implementation-Dependent (ID)	805
Included	Non-Idempotent-Outcome (NIO)	125
Included	Order-Dependent (OD)	122
Included Total		**1052**
Excluded	Compilation Error	134
Excluded	Not Reproducible / Others	559
Excluded Total		**693**

branch of that pull request no longer exists, or the developer decided to customize the changes of the pull request in their own commit and we could not find that change-implementing commit. Also, issue reports are not explicitly included in IDoFT. Therefore, we traced back to its corresponding issue report by automatically matching issue IDs in pull request descriptions, followed by manual inspection when automated mapping was unsuccessful. For NIO tests, we also used the custom JUnit test runner provided by iDFlakies, configured to execute the same test twice within the same JVM. We categorize a test as NIO if it passed on the first execution but failed on the second execution in the same JVM. All remaining steps (compilation filter, reproduction, zip file creation) are the same as the ones we use when handling tests from the Jira data source. After looking for the flaky and fix commits of these flaky tests, compiling them, and reproducing their flaky-test failures, we identify 1052 flaky tests whose failures are reliably reproducible, and 693 flaky tests are excluded due to issues, such as Not Reproducible and Compilation Error.

4.3 Dataset Statistics

The dataset comprises 300 zip archives, covering 1115 reproducible flaky tests across the categories ID, NIO, OD, TD, and Unclassified (UD). Each zip file may contain multiple flaky tests associated with a specific module and commit version where the test was found to be flaky in the same project. Table 2 summarizes the composition of our dataset, detailing how flaky tests from Jira issue reports and IDoFT are included or excluded. The table reports the number of tests per flaky-test category, along with the exclusion reasons, including Not Reproducible failures, Compilation Errors, and other reasons, such as tests that were not flaky unit tests or were not Java/Maven-based.

Not Reproducible / Others. The most common reason we excluded tests was because of tests for which we could not reproduce the flaky-test failure. Among the 793 tests grouped under Not Reproducible / Others in Table 2, 559 are from IDoFT and 234 are from Jira. All Not Reproducible tests were run with the reproduction steps described previously in sections 4.2.1 and 4.2.2. It is

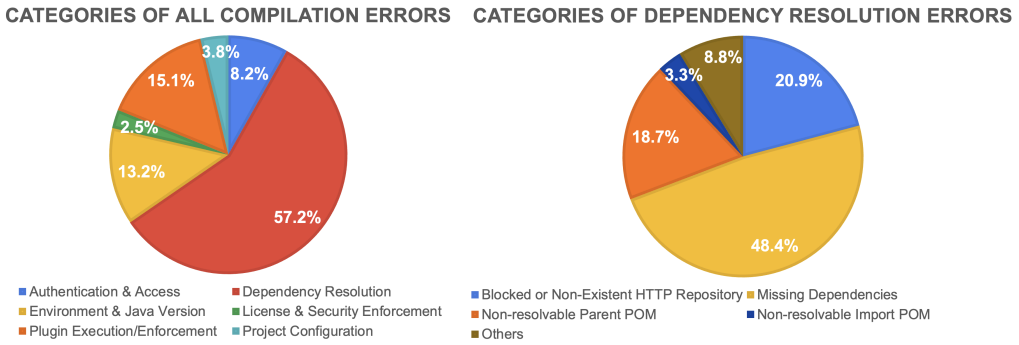


Fig. 3. Categorization of compilation errors in our dataset: compilation-error categories (left) and dependency-resolution error categories (right).

possible that our reproduction steps ultimately could still have led to reproducing the flaky-test failures, but we missed specific necessary conditions, e.g., the right order of unordered collections for ID tests, right test order for OD tests, or right amount or location of delay for TD tests.

Compilation Errors. We encountered problems with compiling the project code and tests on some project versions, even though the project version is known to have flaky tests. Specifically, our data collection excluded 159 tests due to compilation errors. To understand their causes, we performed a categorization on the compilation errors. These errors may not be due to our experimental design, and instead can come from the projects' own outdated or unreachable dependencies. We randomly sampled 100 build logs and manually assigned 15 fine-grained labels, each representing a distinct compilation error category corresponding to a recurring root cause (e.g., missing dependencies, Java version mismatches). We used keywords from error messages in build logs to construct regex rules and to build a few-shot prompt for GPT-4o-mini used for quickly categorizing other compilation errors. We categorized other logs using a regex-first approach; logs that did not match any regex rule were then categorized by GPT-4o-mini using our prompt. We observe that 104 (65.4%) of the categorizations are done deterministically via regex rules, so GPT-4o-mini was used for only the remaining 34.6% of cases (with temperature set to zero for determinism). The resulting compilation error categories were then grouped into higher-level categories based on shared failure causes (e.g., Dependency Resolution errors, Plugin Execution errors). We re-examined remaining uncategorized logs, leading to refined rules or new labels, and this process was repeated until all logs were covered and the categories stabilized. Our compilation error categorization resulted in six high-level categories shown in Figure 3: (1) Authentication & Access errors, which involve missing credentials, keys, or permissions; (2) Dependency Resolution errors, which involve missing or unavailable dependencies; (3) Environment & Java Version errors, which involve issues with Java versions; (4) License & Security Enforcement errors, which involve failed license or vulnerability checks; (5) Plugin Execution/Enforcement errors, which occur when a Maven plugin stops the build because a required tool or plugin dependency is missing, or because a configured rule is violated, such as the project including the Checkstyle [2] plugin, which detects a coding-style violation; and (6) Project Configuration errors, which involve invalid build settings, such as a POM file referencing a module directory that does not exist.

Dependency resolution errors account for the substantial majority of compilation errors: 80.0% of the compilation errors (20 out of 25) from the Jira data source and 53.0% of the compilation errors (71 out of 134) from the IDoFT data source. Due to the prominence of these errors, we further categorize them. Among the dependency resolution errors across tests from both data sources, Missing Dependencies is the most common issue (48.4%), occurring when Maven cannot locate

required artifacts, even though the dependency is specified correctly. Blocked or Non-Existent HTTP Repository errors (20.9%) arise when Maven cannot access an outdated, insecure, or removed repository. Non-resolvable Parent POM errors (18.7%) occur when Maven cannot resolve the parent POM file declared for a project. Non-resolvable Import POM errors (3.3%) occur when Maven cannot find an imported POM file, which provides version information for dependencies. Together, these four categories account for 91.2% of dependency resolution errors. The remaining errors include missing dependency version specifications in the POM, failed dependency resolution, and other less frequent dependency issues.

Our in-depth study of compilation errors suggests a few changes that are likely to help research or dataset work involving legacy projects. To address Authentication & Access, License & Security Enforcement, and Plugin Execution/Enforcement errors, we suggest running Maven with multiple skip flags to avoid unnecessary checks, such as `-Drat.skip`, which audits software releases, or `-Dgpg.skip`, which skips the signing of a Maven artifact, where the key to sign the artifact is often not publicly available. Skipping these checks should not affect the correctness of running tests. To address Environment & Java Version errors, we suggest switching between a few Java versions, such as switching between Java 8, 11, and 17 to resolve version issues. To address Dependency Resolution errors, we suggest replacing SNAPSHOT dependencies with the closest stable versions from Maven Central in all POM files, and updating repository URLs from http to https to overcome Blocked or Non-Existent HTTP Repository errors. We leave the application of our suggestions to address these compilation errors and the use of automated compilation-repair tools, such as BuildMedic [52], as future work to enhance the ReproFlake dataset.

Open Research Challenge - *Improving Compilation of Legacy Projects*
Our categorization shows that legacy projects used in flaky-test research often fail to compile due to dependency resolution issues. The most common problems are missing dependency versions, missing dependencies, inaccessible repositories, and unresolvable import POMs. From our analysis and categorization of 159 compilation errors, we identified various suggestions to address compilation problems of legacy projects. Existing tools [52] might also be able to repair some compilation errors. Future work could evaluate our suggestions and these existing tools for compiling legacy flaky-test projects to further enhance the ReproFlake dataset.

5 Empirical Study

We address the following research questions to characterize the flaky tests in our dataset, both when the tests are in their flaky version of code and when the tests are in their fixed version of code. We examine the distribution of flaky-test categories across the tests, the size of changes between the flaky and fixed versions, and how often those changes are to test code, code-under-test (CUT), or both. We also analyze the differences in the code coverage of flaky tests between flaky and fixed versions. Our findings characterize flaky tests and their fixes, providing directions for future flaky-test analysis and repair research.

- **RQ1**: What are the categories of the flaky tests in our dataset?
- **RQ2**: Where are the code changes to fix flaky tests located and what is the size of these code changes?
- **RQ3**: How deterministic is the code coverage of flaky tests and how do flaky-test fixes affect code coverage?

Table 3. Code changes in test code and code under test (CUT) across flaky-test categories.

Category	Tests	Number of tests with changes in			Number of lines	
		Test Code	CUT	Both	Test Code	CUT
ID	819	584 (71.3%)	288 (35.2%)	53 (6.5%)	112.35	121.05
NIO	125	125 (100.0%)	1 (0.8%)	1 (0.8%)	52.18	0.13
OD	126	126 (100.0%)	90 (71.4%)	90 (71.4%)	11.06	1.59
TD	36	34 (94.4%)	17 (47.2%)	15 (41.7%)	39.36	18.97
UD	9	8 (88.9%)	4 (44.4%)	3 (33.3%)	46.00	11.67
Sum x 4 | Avg x 2	**1115**	**877 (78.7%)**	**400 (35.9%)**	**162 (14.5%)**	**91.26**	**89.81**

5.1 RQ1: Categories of Flaky Tests

Our dataset includes different categories of flaky tests, and the distributions of these categories are dependent on the data source from where we obtained them. For tests from Jira issue reports, we can reproduce the flaky-test failures for ID, OD, TD, and even for a few UD tests. For flaky tests from IDoFT, we can reproduce the flaky-test failures for ID, NIO, and OD tests. Table 2 shows the breakdown of the number of flaky tests per category. The majority of the dataset consists of ID flaky tests from IDoFT, whereas from Jira we obtain mostly TD flaky tests. Overall, the ReproFlake dataset consists of 73.5% ID, 11.2% NIO, 11.3% OD, 3.2% TD, and 0.8% UD tests, where the percentages are calculated across the entire set of 1115 reproducible flaky tests from both Jira issue reports and IDoFT.

5.2 RQ2: Location and Size of Flaky-Test Fixes

We analyze where flaky-test fixes occur, either in test code, code-under-test, or both. We also track how large those fixes are, measured as the summation of the number of added and deleted lines of code to identify repair location patterns, which can guide future automated repair techniques. By seeing patterns across different flaky-test categories, developers can better predict the effort needed and quickly focus on the right parts of code.

As shown in Table 3, changes to fix flaky tests are not mutually exclusive to either test code or CUT: a test that is fixed through changes to both locations is included in both columns, while the “Both" column reports their overlap. Flaky tests are more often fixed through changes to test code compared to CUT, across all flaky-test categories. Specifically, we find that 64.1% of flaky-test fixes involved changes to test code only. This finding is similar to prior studies [44, 51] on flaky-test fixes, which found that flaky-test fixes involved test code only 71-73% of the time. However, many fixes span both locations, particularly for OD tests, for which 71.4% change both test code and CUT. Similarly, 41.7% of TD test patches change both locations, although test code is modified more frequently overall. NIO test fixes are the most test-focused: all NIO fixes change test code, while only one also changes CUT. For ID tests, test-code changes are more frequent than CUT changes, although the average number of changed CUT lines is slightly larger than the average number of changed test-code lines. Overall, 93.5% of ID, 99.2% of NIO, 28.6% of OD, 58.3% of TD, and 66.7% of UD tests have fixes targeting only either the test code or the CUT. However, because we measure the complete difference in code changes between the flaky and fixed versions, these changes may include code changes unrelated to fixing the flaky test. Multiple flaky tests may also be fixed with the same patch, and such patches may be counted multiple times. Thus, the reported locations and sizes may be an overestimation for fixing a specific flaky test. Future repair techniques may use these patterns to prioritize likely code-change locations, while still considering that a fix may require changes to both.

Table 4. Code coverage analysis of flaky and fixed version.

Number of unique coverage sets	Number of tests		Number of tests unique		
	Flaky	Fixed	Flaky	Fixed	Both
1	801	799	2	0	799
2	11	12	1	2	10
3	2	3	1	2	1
4	1	0	1	0	0
5	0	1	0	1	0
8	2	2	0	0	2
9	1	1	0	0	1
Average x 2 | Sum x 3	**1.05**	**1.05**	**5**	**5**	**813**

Open Research Challenge - *Prioritizing Location of Fixes*
Our findings in RQ2 show that developers modify test code more frequently than CUT when fixing flaky tests. However, many patches span both locations, particularly for OD tests. These findings can help developers prioritize where to make fixes to flaky tests or to guide automatic repair tools to focus their fixes in the most likely locations first (test code or CUT). For example, repair techniques could focus on making changes to test code only for NIO tests, prioritize test code for ID, TD, and UD tests, and make changes to both equally when it comes to OD tests.

5.3 RQ3: Code Coverage Between Flaky and Fixed Versions

We demonstrate the usability of our framework for analyzing dynamic flaky-test executions by integrating JaCoCo [11] to collect code coverage from running the flaky and fixed versions of the flaky tests. Adding the JaCoCo tool to collect code coverage for our dataset required only a one-line change to the test-execution script used by all flaky tests. Because this script is used across the dataset, the change enables us to attempt code coverage collection for all 1115 tests without modifying each project individually. We also integrated Checkstyle [2], a static-analysis tool that checks Java coding rules, similarly with a one-line change to the shared script and collected results for all 1115 tests, whose results we share on our website [17].

For our coverage analysis, we executed each test independently 10 times in both versions and recorded the lines covered during each execution. The collected coverage includes executed lines from both test code and CUT, including test setup, teardown, and all methods invoked by the test during execution. We observed that for 297 tests, the process failed to generate the expected JaCoCo coverage files, preventing us from comparing their covered-line sets. We could not obtain JaCoCo execution data for 157 tests due to project-specific configurations (e.g., storing class files in unusual locations as defined by the project's POM file), 46 encountered dependency resolution errors after adding JaCoCo, and 9 had issues with Surefire/JVM process crashing when run using JaCoCo. The remaining 85 failures were caused by incompatibilities with our infrastructure, which we plan to address in future work. We excluded all these 297 tests, leaving 818 tests for this analysis. Specifically, we aim to answer the following questions:

(1) Does coverage change across the 10 repeated runs of the same flaky and fixed versions?
(2) Does the flaky version of the test cover the same lines of code as the fixed version?
(3) Do certain fixes of certain flaky-test categories affect coverage more than others?

Table 5. Code coverage analysis of flaky version compared with fixed version.

Category	Number of flaky tests	Number of tests with Same cov.	Different cov.
ID	556	127 (22.8%)	429 (77.2%)
NIO	122	5 (4.1%)	117 (95.9%)
OD	122	118 (96.7%)	4 (3.3%)
TD	13	2 (15.4%)	11 (84.6%)
UD	5	2 (40.0%)	3 (60.0%)
All	**818**	**254 (31.1%)**	**564 (68.9%)**

For (1), we represent each execution of the same flaky or fixed test by the set of covered lines during that execution, and we count how many distinct sets appear across the 10 runs for that version of the test. If a flaky or fixed version has only one distinct set, then that means that all 10 executions cover the same lines, while 10 sets means every execution has a different covered-line set. For (2), we take the union of lines covered across 10 executions for each version and check whether the two unions match. For (3), we group results by flaky-test category and compare how often flaky and fixed versions have identical coverage per category. Similar to (2), each covered-line set is the union across 10 executions.

Coverage Variation Across Repeated Executions. Table 4 shows that coverage was highly stable across the 10 repeated executions. In the flaky versions, 801 of 818 tests (97.9%) produced only one distinct covered-line set across all 10 runs, while 17 tests (2.1%) produced multiple sets. Similarly, 799 fixed versions (97.7%) produced one distinct covered-line set across all 10 runs, while 19 (2.3%) produced multiple sets. Both versions produced an average of 1.05 distinct covered-line sets per test. Although it may seem that the fixed version of flaky tests would be the only version that will have deterministic code coverage, our findings suggest that flaky tests are actually very likely to have deterministic code coverage on both the flaky version and fixed version. Interestingly, we also find that five flaky tests had a different number of unique covered-line sets between the flaky and fixed versions. Four out of the five flaky tests actually increased in the number of unique covered-line sets when the tests became fixed, while only one decreased in the number of unique covered-line sets after the flakiness was fixed. This finding suggests that although fixes for flaky tests may make the outcome of the test deterministic, such fixes may make the coverage less deterministic.

Coverage Differences Between Flaky and Fixed Versions. As shown in Table 5, 254 tests (31.1%) covered the same lines in both versions, whereas 564 tests (68.9%) had different covered-line sets. Thus, although coverage was generally stable across repeated executions on the same version, it frequently changed between the flaky and fixed versions. On the one hand, fixed flaky tests executing different lines of code is likely expected as the fixes may have added or deleted lines that are executed by the test. On the other hand, it is also in the best interest of the fixed version to not deviate much in code coverage from the flaky version, as code coverage deviations imply that the fix changed the semantics of the test, and this changed semantics may no longer reflect the intention of the test.

Differences Across Flaky-Test Categories. Coverage differences are most common for NIO (95.9%), followed by TD (84.6%), ID (77.2%), and UD (60.0%) tests. In contrast, only four OD tests (3.3%) have different coverage. In one case that we inspected, the patch added setup and teardown code to the OD-relevant test's class, which was not executed when we collected coverage from running just the OD test in isolation. The patch also changed the code at the same file and line number. Therefore, our line-number based code coverage comparison treated it as the same covered

line in both versions. These factors illustrate why fixing an OD test can change test code or CUT without producing a different covered-line set in our analysis. Our results suggest that OD tests are more likely to change code not executed by the test or have fixes that modify existing lines, while other categories of flaky tests are more likely to have fixes that are line additions or deletions.

Open Research Challenge - *Comparing Flaky-Test Fixes*
Our analysis provides coverage data for flaky and fixed versions of flaky tests. These paired versions can serve as references for evaluating fixes produced by automated repair techniques. Future studies can compare generated and developer-written fixes using coverage, mutation score, failure rate, and lines of code changed. When multiple fixes are generated, coverage similarity to the flaky version could be used to select the best one among them.

5.4 Discussion on Flaky-Test Exceptions Across Flaky-Test Categories

To better understand the flaky-test failures in ReproFlake, we categorize the failures across different flaky-test categories. We followed a similar approach described in Section 4.3 for categorizing compilation errors to categorize flaky-test failures. Namely, we manually labeled 100 failure logs with fine-grained failure subcategories, derived regex rules from recurring log patterns, and used few-shot prompts with GPT-4o-mini to categorize logs that did not match any rule.

Across all 1115 flaky tests, Assertion errors are the most common (81.5%), although their prevalence varies. They account for 79.2% of ID-test failures, 99.2% of NIO-test failures, 78.6% of OD-test failures, 83.3% of TD-test failures, and 77.8% of UD-test failures. While Assertion errors dominate the failures for NIO tests, ID and OD tests have a wider variety of different categories of failures. The other ID-test failures consist of Runtime/Other errors (7.3%), Null Pointer errors (6.2%), and Parameter Resolution errors[2] (4.3%). The other OD-test failures consist of Class Initialization errors[3] (16.7%), Runtime/Other errors (4.0%), and one RPC error[4] (0.8%).

Although most failures are Assertion errors, some less common error categories may indicate particular flaky-test categories. In our dataset, nearly all Class Initialization errors occur in OD tests, Timeout errors occur only in TD tests, and Parameter Resolution and Network errors occur only in ID tests. Future work could further subcategorize Assertion errors to evaluate whether their finer-grained patterns may also indicate the likely flaky-test category.

Open Research Challenge - *Inferring Flaky-Test Categories From Errors*
A potential use of our error analysis is to predict the likely flaky-test category from a failure. High-level error categories alone may be insufficient because Assertion errors dominate every flaky-test category. However, some less common errors show stronger associations. Prior work, such as FlakyCat [20] and FlakyDoctor [28], showed that knowing the flaky-test category is useful for diagnosis and repair, but typically requires analyzing test code or observing behavior across multiple runs. Future work should evaluate whether errors can accurately identify flaky-test categories and improve diagnosis and repair. Future work could also subcategorize Assertion errors to better predict the most likely flaky-test category.

[2]Parameter Resolution errors occur when the required test inputs cannot be provided.
[3]Class Initialization errors occur when the JVM cannot load a class at runtime.
[4]RPC errors occur when a remote procedure call fails.

Open Research Challenge - *Providing New Information to Fix Flaky Tests*
Our findings about errors can be combined with the information provided by ReproFlake, such as issue-report descriptions, covered methods and lines, and flaky and fixed code, to construct new context information concerning flaky-test behaviors. This context could be provided to large-language models to generate candidate repairs, which can then be validated using ReproFlake's reproducible environments.

6 Threats to Validity

Our findings may not generalize to other flaky-test categories or to projects outside of the Java and Maven-based projects we evaluated. ReproFlake includes four main flaky-test categories, although prior studies [30, 51] reported ten or more categories. The included categories were determined by the flaky tests we could find using our systematic search, as well as those whose failures we could reliably reproduce. Our guidelines support adding more Java and Maven-based flaky tests that belong to the categories we studied and provide fallback steps for tests whose categories are unknown. Nevertheless, extending ReproFlake to additional categories may require new tools and steps for reproducing their failures. Our code-change analysis compares the complete patches between flaky and fixed versions. These patches may contain unrelated changes and may be shared by multiple flaky tests, making it difficult to identify the changes required for each individual test. Our code coverage analysis may not precisely capture the actual coverage achieved from running a flaky test due to its natural non-determinism. To reduce the coverage comparison's dependence on any single execution, we execute each version 10 times and compare the unions of covered lines. However, our line-based coverage comparison may treat modified code at the same line as unchanged coverage. This limitation affects only the code coverage analysis between flaky and fixed versions, and we may observe that a test has the same coverage between versions if all of the changes needed for the fix are to existing lines, i.e., there are no added or deleted lines. Our error categorization may contain mistakes from the use of regex rules and an LLM for categorization. To mitigate this threat, we first manually inspected 100 logs, which informed the design of our regex rules and prompt for the LLM. We also set the temperature of the LLM to zero to reduce the effect of non-determinism on our categorization. Lastly, because the numbers of tests differ across flaky-test categories, results for categories with few tests may be less stable. We therefore report the underlying test counts together with the percentages.

7 Related Work

Datasets for improving software engineering research have a long tradition. From manually-curated datasets for test adequacy criteria [38] in 1994 to more recent and general datasets for software testing and analysis, such as SIR [18], Defects4J [4], and BugSwarm [1]. There has also been an increasing number of domain-specific datasets for research, such as for test prioritization [31], Android apps [21], continuous integration [27], and bug prediction [29].

ReproFlake, like prior datasets, provides real-world software artifacts for evaluating testing techniques. However, unlike general-purpose datasets, such as Defects4J or BugSwarm, it focuses specifically on flaky tests and supports reproducibility by including execution environments, logs, and both flaky and fixed versions. These artifacts make ReproFlake well-suited for studying non-deterministic test behavior and repair. A popular flaky-test dataset is IDoFT [10], which consists of a list of flaky tests, their GitHub URL, flaky version, and pull request information, but it lacks reproducibility artifacts, such as error logs, issue reports, and execution environments. FlakeFlagger [23] includes a different set of flaky tests than IDoFT and includes information, such

as build and failure logs, but does not provide fixed versions or execution environments. Similarly, FlakyFix [33] and DeFlaker [26] include flaky tests but lack key artifacts, such as fixed versions, logs, and reproducibility artifacts. FlakyCat [20] categorizes flaky tests across multiple dimensions but does not include reproducibility artifacts. Barbosa et al. [25] provide issue reports and categorization, but lack flaky or fixed versions, or reproducible artifacts. As shown in Table 1, ReproFlake is the only dataset that provides both flaky and fixed versions, execution environments, detailed error logs, pull requests, and issue reports – enabling end-to-end reproducibility and deeper analysis of flaky-test behavior.

Beyond datasets, prior work studies flaky-test detection [26, 45, 59], reproduction [57], and repair [28, 49, 58, 60]. ReproFlake supports these areas by providing categorized tests, execution environments, failure logs, and developer-written fixes. Alshammari et al. [22] studied whether failure messages and stack traces can distinguish flaky-test failures from regression-test failures using failure deduplication and machine-learning classifiers; ReproFlake's failure logs and execution environments can support further evaluation of such techniques.

8 Conclusion

In the past decade, many software development organizations have reported flaky tests as one of their biggest problems. To help with this issue, many research papers have studied the problems of flaky tests and proposed automated techniques to detect, debug, and fix flaky tests. To facilitate such research, various datasets of flaky tests have been proposed, yet reproducing the failures of flaky tests, especially for many categories of them, can still be challenging. Different datasets have also curated different types of information, e.g., error logs and issue reports, which are often useful for automated debugging and fixing techniques. In this paper, we presented ReproFlake, which is a dataset of 1115 Java and Maven-based flaky tests focused on providing the following for each flaky test: (1) an environment (in the form of Docker images) to aid in the reproducibility of the environment to observe the flaky-test failure, (2) automated scripts to run the flaky test and observe the flaky-test failure, (3) developer-written fix for the flaky-test failure, and (4) various flaky-test related information, such as an error log of the flaky-test failure and issue report information describing the flaky-test failure. Using this dataset, we studied the location and size of code changes for fixing flaky tests, how deterministic is the code coverage of flaky tests, and how do flaky-test fixes affect the code coverage of flaky tests. Our study revealed important open research challenges, which our dataset can help address: (A) compilation of legacy projects can focus on the prominent categories that were responsible for why some subjects in our study did not compile, (B) fixing of flaky tests can leverage the category of a flaky test to better predict where the fix is likely to be, (C) debugging of flaky tests can leverage the error category of a flaky-test failure to predict what category a flaky test is, (D) fixing of flaky tests can leverage a vast combination of information altogether (e.g., flaky-test category, error log, issue report) from our dataset to improve fixing, and (E) comparison of flaky-test fixes using traditional testing metrics (e.g., code coverage, mutation score) can be done easier with our dataset. We make our dataset publicly available [17] and provide guidelines on how we constructed the dataset and to enable future work to expand it as needed.

Data Availability

All relevant data, scripts, and results generated for this work are available on our website [17].

Acknowledgments

We thank Chuyan (Sissi) Wang, Md. Mahmudul Hasan Pious, and Ruochen Wang for their help on this work. This work is partially supported by the US National Science Foundation grant numbers CCF-2145774 and CCF-2338287, and the Jarmon Innovation Fund.

References

[1] 2026. BugSwarm. https://www.bugswarm.org/dataset.
[2] 2026. Checkstyle. https://checkstyle.org.
[3] 2026. COLLECTIONS-812. https://issues.apache.org/jira/browse/COLLECTIONS-812.
[4] 2026. Defects4J. https://github.com/rjust/defects4j.
[5] 2026. Docker. https://www.docker.com.
[6] 2026. Expedia – Fixing Flaky Time Based Unit Tests. https://medium.com/expedia-group-tech/fixing-flaky-time-based-unit-tests-176accf5096e.
[7] 2026. Flakiness dashboard HOWTO. http://www.chromium.org/developers/testing/flakiness-dashboard.
[8] 2026. Flaky Tests (And How to Avoid Them). https://engineering.salesforce.com/flaky-tests-and-how-to-avoid-them-25b84b756f60.
[9] 2026. Guideline. https://docs.google.com/document/d/1VcORh2Otr7iYqbIXlwVW7ikPnlEV8bOF.
[10] 2026. IDoFT. https://github.com/TestingResearchIllinois/idoft.
[11] 2026. JaCoCo Java Code Coverage Library. https://www.jacoco.org/jacoco.
[12] 2026. Jira. https://issues.apache.org/jira.
[13] 2026. Jira Python Library. https://jira.readthedocs.io.
[14] 2026. A Machine Learning Solution for Detecting and Mitigating Flaky Tests. https://eng.fitbit.com/a-machine-learning-solution-for-detecting-and-mitigating-flaky-tests.
[15] 2026. Netflix Automation Talks - Test Automation at Scale. https://youtu.be/FrBN94gUn_I?t=764.
[16] 2026. NonDex. https://github.com/TestingResearchIllinois/NonDex.
[17] 2026. ReproFlake: A Reproducible Dataset of Flaky-Test Failures. https://sites.google.com/view/reproflake-dataset.
[18] 2026. SIR. https://sir.csc.ncsu.edu/portal/index.php.
[19] 2026. Test Verification. https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification.
[20] Amal Akli, Guillaume Haben, Sarra Habchi, Mike Papadakis, and Yves Le Traon. 2023. FlakyCat: Predicting Flaky Tests Categories Using Few-Shot Learning. In *International Conference on Automation of Software Test*. doi:10.1109/AST58925.2023.00018
[21] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Mining Software Repositories*. doi:10.1145/2901739.2903508
[22] Abdulrahman Alshammari, Paul Ammann, Michael Hilton, and Jonathan Bell. 2024. 230,439 Test Failures Later: An Empirical Evaluation of Flaky Failure Classifiers. In *International Conference on Software Testing, Verification, and Validation*. doi:10.1109/ICST60714.2024.00031
[23] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *International Conference on Software Engineering*. doi:10.1109/ICSE43902.2021.00140
[24] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* (2019). doi:10.1145/3360585
[25] Keila Barbosa, Ronivaldo Ferreira, Gustavo Pinto, Marcelo d'Amorim, and Breno Miranda. 2023. Test Flakiness Across Programming Languages. *IEEE Transactions on Software Engineering* (2023). doi:10.1109/TSE.2022.3208864
[26] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *International Conference on Software Engineering*. doi:10.1145/3180155.3180164
[27] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Mining Software Repositories*. doi:10.1109/MSR.2017.24
[28] Yang Chen and Reyhaneh Jabbarvand. 2024. Neurosymbolic Repair of Test Flakiness. In *International Symposium on Software Testing and Analysis*. doi:10.1145/3650212.3680369
[29] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An Extensive Comparison of Bug Prediction Approaches. In *Mining Software Repositories*. doi:10.1109/MSR.2010.5463279
[30] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. doi:10.1145/3338906.3338945
[31] Emad Fallahzadeh and Peter C Rigby. 2022. The Impact of Flaky Tests on Historical Test Prioritization on Chrome. In *International Conference on Software Engineering (Software Engineering in Practice Track)*. doi:10.1145/3510457.3513038
[32] Sakina Fatima, Taher A. Ghaleb, and Lionel Briand. 2023. Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests. *IEEE Transactions on Software Engineering* (2023). doi:10.1109/TSE.2022.3201209
[33] Sakina Fatima, Hadi Hemmati, and Lionel Briand. 2024. FlakyFix: Using Large Language Models for Predicting Flaky Test Fix Categories and Test Code Repair. *IEEE Transactions on Software Engineering (2024)*. doi:10.1109/TSE.2024.3472476
[34] Phil Gochenour and Rachel Andre. 2026. How to Deal with Flaky Java Tests. https://wiki.saucelabs.com/display/DOCS/How+to+Deal+with+Flaky+Java+Tests.

[35] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. 2016. NonDex: A Tool for Detecting and Debugging Wrong Assumptions on Java API Specifications. In *International Symposium on Foundations of Software Engineering (Tool Demonstrations Track)*. doi:10.1145/2950290.2983932

[36] Sarra Habchi, Guillaume Haben, Jeongju Sohn, Adriano Franci, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2022. What Made This Test Flake? Pinpointing Classes Responsible for Test Flakiness. In *International Conference on Software Maintenance and Evolution*. doi:10.48550/arXiv.2207.10143

[37] Brian Harry. 2026. How We Approach Testing VSTS to Enable Continuous Delivery. https://blogs.msdn.microsoft.com/bharry/2017/06/28/testing-in-a-cloud-delivery-cadence.

[38] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In *International Conference on Software Engineering*. doi:10.1109/ICSE.1994.296778

[39] He Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. 2017. What Causes My Test Alarm? Automatic Cause Analysis for Test Alarms in System and Integration Testing. In *International Conference on Software Engineering*. doi:10.1109/ICSE.2017.71

[40] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *International Symposium on Software Testing and Analysis*. doi:10.1145/2610384.2628055

[41] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. 2020. Modeling and Ranking Flaky Tests at Apple. In *International Conference on Software Engineering (Software Engineering in Practice Track)*. doi:10.1145/3377813.3381370

[42] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug Report Driven Program Repair. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. doi:10.1145/3338906.3338935

[43] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *International Symposium on Software Testing and Analysis*. doi:10.1145/3293882.3330570

[44] Wing Lam, Kivanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A Study on the Lifecycle of Flaky Tests. In *International Conference on Software Engineering*. doi:10.1145/3377811.3381749

[45] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *International Conference on Software Testing, Verification, and Validation*. doi:10.1109/ICST.2019.00038

[46] Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D. Ernst, and Tao Xie. 2020. Dependent-Test-Aware Regression Testing Techniques. In *International Symposium on Software Testing and Analysis*. doi:10.1145/3395363.3397364

[47] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. 2020. Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects. In *International Symposium on Software Reliability Engineering*. doi:10.1109/ISSRE5003.2020.00045

[48] Chengpeng Li, M. Mahdi Khosravi, Wing Lam, and August Shi. 2023. Systematically Producing Test-orders to Detect Order-Dependent Flaky Tests. In *International Symposium on Software Testing and Analysis*. doi:10.1145/3597926.3598083

[49] Chengpeng Li, Chenguang Zhu, Wenxi Wang, and August Shi. 2022. Repairing Order-Dependent Flaky Tests via Test Generation. In *International Conference on Software Engineering*. doi:10.1145/3510003.3510173

[50] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Symposium on Principles of Programming Languages*. doi:10.1145/2837614.2837617

[51] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *International Symposium on Foundations of Software Engineering*. doi:10.1145/2635868.2635920

[52] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically Repairing Dependency-Related Build Breakage. In *International Conference on Software Analysis, Evolution and Reengineering*. doi:10.1109/SANER.2018.8330201

[53] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In *International Conference on Software Engineering (Software Engineering in Practice Track)*. doi:10.1109/ICSE-SEIP.2017.16

[54] John Micco. 2026. Continuous Integration at Google Scale. https://www.slideshare.net/JohnMicco1/2016-0425-continuous-integration-at-google-scale.

[55] Md Tajmilur Rahman and Peter C. Rigby. 2018. The Impact of Failing, Flaky, and High Failure Tests on the Number of Crash Reports Associated with Firefox Builds. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. doi:10.1145/3236024.3275529

[56] Shanto Rahman, Bala Naren Chanumolu, Suzzana Rafi, August Shi, and Wing Lam. 2025. Ranking Relevant Tests for Order-Dependent Flaky Tests. In *International Conference on Software Engineering*. doi:10.1109/ICSE55347.2025.00178

[57] Shanto Rahman, Aaron Massey, Wing Lam, August Shi, and Jonathan Bell. 2024. Automatically Reproducing Timing-Dependent Flaky-Test Failures. In *International Conference on Software Testing, Verification, and Validation*. doi:10.1109/ICST60714.2024.00032

[58] Shanto Rahman and August Shi. 2024. FlakeSync: Automatically Repairing Async Flaky Tests. In *International Conference on Software Engineering*. doi:10.1145/3597503.3639115

[59] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-Deterministic Specifications. In *International Conference on Software Testing, Verification, and Validation*. doi:10.1109/ICST.2016.40

[60] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A Framework for Automatically Fixing Order-Dependent Flaky Tests. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. doi:10.1145/3338906.3338925

[61] Pavan Sudarshan. 2026. No More Flaky Tests on the Go Team. http://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team.

[62] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes. In *International Conference on Software Engineering*. doi:10.1109/ICSE.2019.00048

[63] Anjiang Wei, Pu Yi, Zhengxi Li, Tao Xie, Darko Marinov, and Wing Lam. 2022. Preempting Flaky Tests via Non-Idempotent-Outcome Tests. In *International Conference on Software Engineering*. doi:10.1145/3510003.3510170

[64] Eric Wendelin. 2026. Introducing Flaky Test Mitigation Tools. https://blog.gradle.org/gradle-flaky-test-retry-plugin.

[65] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. doi:10.1145/318774.318946

[66] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *International Symposium on Software Testing and Analysis*. doi:10.1145/2610384.2610404

[67] Celal Ziftci and Jim Reardon. 2017. Who Broke the Build?: Automatically Identifying Changes That Induce Test Failures in Continuous Integration at Google Scale. In *International Conference on Software Engineering (Software Engineering in Practice Track)*. doi:10.1109/ICSE-SEIP.2017.13

Received 2026-01-30; accepted 2026-04-16