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. Although many techniques have been proposed to detect, debug, and repair flaky tests, reproducing their failures remains a major challenge due to their inherent nondeterminism. Many datasets related to flaky tests exist to help researchers study them, but these datasets are often composed of disjoint sets of flaky tests, 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. In this work, we aim to create a reproducible dataset of flaky tests, which are curated from both developer issue reports and a popular dataset of flaky tests.

Compared to prior flaky test datasets, our dataset is the first to provide (1) a reproducible environment to compile the flaky tests, (2) scripts to run the tests to reproduce the failure, (3) scripts to automatically apply the flaky test fixes and ensure that the test is no longer flaky, and (4) execution logs of the flaky test passing and failing. We present ReproFlake, a dataset of 1115 reproducible flaky tests, spread across four different flaky test categories. We create guidelines to help others contribute to this reproducible dataset, as well as demonstrate how to use our dataset to understand the challenges with reproducing flaky test failures (i.e., challenges that researchers may face when using any of the prior flaky test datasets), the characteristics (e.g., location of the fix and its correlation with the flaky test category), as well as the difficulties researchers may face in using our dataset to collect additional information (e.g., code coverage) about flaky tests. Our findings show that error information helps identify flaky test categories and guide repairs, that unresolved compilation failures highlight challenges in building legacy projects, and that knowing typical fix locations helps prioritize repair efforts.

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. 1, 1 (May 2026), 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 testing is the presence of *flaky tests* [48, 68], which are tests that can both pass and fail when run

*Both authors contributed equally to this research.

Authors' Contact Information: 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.

ACM XXXX-XXXX/2026/5-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 introduced in their recent code changes, but this test could have flakily failed regardless of any changes. Many software development organizations have reported flaky tests as one of their biggest problems [9, 23, 27–29, 32, 34, 36, 38, 40, 50–54, 60, 61, 66, 69].

Prior work on developing techniques to mitigate flaky tests also resulted in various flaky test datasets [10, 12, 17, 18, 23, 31, 41, 48, 63]. 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, 41, 63], failure logs of flaky tests [12], or flaky tests reported by developers [10, 23, 48] vs. flaky tests found by automated tools [12, 18, 41, 63]). For example, one of the most popular flaky test datasets is the International Dataset of Flaky Tests (iDoFT) [42, 63]. 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 [25, 26, 43–46, 55, 57]. However, iDoFT does not have flaky tests reported by developers, nor the failure logs associated with a flaky failure, which other datasets have [10, 12, 23, 48]. In contrast, iDoFT also has pull request links to fixes, which other datasets may not have.

We propose ReproFlake, a new dataset of reproducible flaky tests obtained from developer issues reporting flaky tests 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 Maven Surefire 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, along with an automated script to run the test and reproduce the failure, (2) the code changes accepted by developers that fix the flaky test, (3) a developer-written issue report describing the flaky test (when available), and (4) execution logs of the flaky test passing and failing using our scripts and environment. 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 encompasses 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, Maven-based unit tests. For both Jira and iDoFT tests, we require clearly identifiable flaky and fixing commit SHAs and discard tests that fail to compile. We then reproduce flaky 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 failures, and a reproducible environment to compile and rerun flaky tests, we allow researchers to better study flaky test behavior. Similar to how datasets such as Defects4J [35] and Bugswarm [62] enable various automatic program repair techniques that leverages bug reports [16, 37, 47] 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 classification. Researchers building tools for flaky test fault localization, categorization, or automated test repair can use our artifacts to train, test, and evaluate their ideas on real-world

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 [63]	ID, OD, NIO, NOD, TD, TZD	✓	✓	✗	✗	✓	✗
FlakeFlagger [12]	Does not classify	✓	✗	✗	✓	✗	✗
FlakyFix [26]	ID	✓	✗	✗	✗	✗	✗
DeFlaker [18]	Does not classify	✓	✗	✗	✗	✗	✗
FlakyCat [10]	TD, IO, ID, Randomness, Network	✓	✗	✗	✗	✗	✗
Keila et al. [17]	TD, OD, Platform, Randomness, Resources, ID, Network	✗	✗	✗	✗	✗	✓
ReproFlake	TD, OD, ID, NIO	✓	✓	✓	✓	✓	✓

flaky tests, all in an environment that enables flaky failure reproducibility. ReproFlake enhances the current state-of-the-art flaky test datasets with bug reports, flaky and fixed code, Docker containers, failing/passing logs and test execution 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 challenges with reproducing flaky test failures (i.e., challenges that researchers may face when using any of the prior flaky test datasets), (2) what are the characteristics (e.g., location of the fix and its correlation with the flaky test category of flaky tests in ReproFlake, and (3) how difficult it is to use the dataset to collect additional information (e.g., code coverage) about flaky tests. Our findings are primarily relevant to researchers and developers working on flaky test reproduction and repair. For example, by analyzing compilation failures and fix locations, our results inform them how to set up reproducible environments more efficiently and where repair efforts should be prioritized (e.g., focusing on test code over code under test for certain categories of flaky tests). We also study the exception type of the flaky test failures and find that the majority of exceptions in ReproFlake are assertion errors, which we further categorize based on the exception message. We find that categorization of exception messages can help obtain the flaky test categories, which can be helpful for debugging [10] and automated fixing [20] 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 reproducible dataset containing 1115 flaky tests spread across four different flaky test categories of flaky tests.
- ReproFlake consists of a Docker environment to compile legacy flaky test projects and run our provided scripts to reproduce flaky failures 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 or not from issue reports and pull request history.
- We present an empirical evaluation of the challenges in constructing a reproducible dataset of flaky tests and study various characteristics of flaky tests, providing insights on how future flaky test research can better aid developer's debugging and fixing of flaky tests. Our artifacts are publicly available [8].

2 Background

Software testing is key to checking the correctness of software. When a developer runs tests on their system and observe failures, these failing tests should signal to developers that they have a fault in their code that needs to be fixed. However, often, some tests fail sporadically. Such tests, that can pass or fail when run on the same version of code, are *flaky tests* [23, 49]

Understanding and reproducing flaky tests requires both a clear categorization of their causes and familiarity with the tools that can detect, analyze, or reproduce them. In this section, we first

introduce the main categories of flaky tests and then present the tools we use in our study, spanning a range of detection and reproduction techniques.

Flaky test categories. Flaky test failures may be of different categories, and prior work has defined several categories observed from open-source and industry code.

- **Order-Dependent (OD):** These tests' pass/fail outcomes depend on the order in which they are run [41, 68]. Other tests, known as **OD-relevant tests (ODR)** [55, 59], changes some shared state (e.g., static variables) between the ODR and OD test, thereby influencing the pass/fail outcome of the OD test.
- **Timing-Dependent (TD):** These tests can fail due to concurrency or waiting on asynchronous computations [56, 57]. Uncontrolled thread interleavings due to differences in timing per execution can result in different test outcomes.
- **Implementation-Dependent (ID):** These tests assume deterministic outcomes from APIs whose specifications are actually nondeterministic, such as iteration order over data structures like `HashMap`, whose order is not guaranteed [58].
- **Non-Idempotent-Outcome (NIO):** These tests fail when they are run in the same JVM more than once; the test passes when run the first time in the JVM but fails when run again [65]. 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.

Flaky Test Reproduction and Analysis Tools. We rely on several flaky test detection and reproduction tools to reproduce the failures of different categories of flaky tests.

- **iDFlakies [41] for OD tests:** To detect and reproduce OD tests, we use iDFlakies, a customized Maven plugin for running tests in different orders. iDFlakies runs tests in random orders, where a test that passes in one order but fails in another is flaky. We can easily reproduce a flaky test failure by rerunning a failing order. Given a failing and passing order, prior work [59] relied on delta-debugging [67] to systematically search for the ODR test corresponding to the OD test.
- **NonDex [30, 58] for ID tests:** NonDex identifies APIs with nondeterministic 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 [41] for NIO tests:** iDFlakies detects NIO tests by running the same test multiple times in the same JVM using a custom test runner; if the test passes on the first execution but fails on a subsequent run, it is classified as an NIO flaky test.
- **Docker:** To provide a reliable environment for reproducing flaky test failures, we utilize Docker [22] images. We can configure a Docker image to contain the relevant operating system as well as source code and dependencies needed for execution. We can then share this Docker image with others, who can use it to reproduce specific executions.

3 Example Flaky Test

We identified a Timing-Dependent (TD) flaky test for our dataset while examining issue report *COLLECTIONS-812* [14] 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 versus the standard JDK implementation, `java.util.Properties`.

In Figure 1a, the `save` method writes three elements to the provided `OutputStream` in Line 4 and `PrintStream` in Line 17: (1) a String `comments` (Line 2), (2) the current date and time (recorded to the nearest second), and (3) key-value entries on subsequent lines (with no entries for `EmptyProperties`).

```java
@Test public void testSave() throws IOException {
  final String comments = "Hello world!"; {
  try (ByteArrayOutputStream actual = new
        ByteArrayOutputStream()) {
    PropertiesFactory.EMPTY_PROPERTIES.save(actual, comments);
    try (ByteArrayOutputStream expected = new
          ByteArrayOutputStream()) {
      PropertiesFactory.INSTANCE.createProperties()
        .save(expected, comments);
      String expectedComment =
            getFirstLine(expected.toString("UTF-8"));
      String actualComment =
            getFirstLine(actual.toString("UTF-8"));
      assertEquals(expectedComment, actualComment, () ->
        String.format("Expected String '%s' with length '%s'",
          expectedComment, expectedComment.length()));
      expected.reset();
  // Added to simulate flakiness and ensure the test fails
  // try { Thread.sleep(2000); } catch (InterruptedException e)
        { }
      try (PrintStream out = new PrintStream(expected)) {
        new Properties().save(out, comments);
        assertArrayEquals(expected.toByteArray(),
              actual.toByteArray(), expected::toString);
    } catch (UnsupportedEncodingException e) {
          fail(e.getMessage(), e); }
  }
}
```

(a) Example of TD flaky test

```java
@Test public void testSave() throws IOException {
  final String comments = "Hello world!";
  try (ByteArrayOutputStream actual = new
        ByteArrayOutputStream(); ByteArrayOutputStream expected
        = new ByteArrayOutputStream()) {
    PropertiesFactory.EMPTY_PROPERTIES.store(actual, comments);
    PropertiesFactory.INSTANCE.createProperties().store(expected,
          comments);
    String expectedComment =
          getFirstLine(expected.toString("UTF-8"));
    String actualComment =
          getFirstLine(actual.toString("UTF-8"));
    assertEquals(expectedComment, actualComment, () ->
      String.format("Expected String '%s' with length '%s'",
        expectedComment, expectedComment.length()));
    expected.reset();
// Added to simulate flakiness in fixed version, but test should
      still pass
// try { Thread.sleep(2000); } catch (InterruptedException e) { }
    try (PrintStream out = new PrintStream(expected)) {
      new Properties().store(out, comments);
    }
    String[] expectedLines = expected.toString(
      StandardCharsets.UTF_8.displayName()).split("\\n");
    String[] actualLines = actual.toString(
      StandardCharsets.UTF_8.displayName()).split("\\n");
    assertEquals(expectedLines.length, actualLines.length);
    assertEquals(expectedLines[0], actualLines[0]);
  }
}
```

(b) Developer fixed version of the flaky test

Fig. 1. Example of TD flaky and fixed test from apache commons-collections. The red-commented block was added to better reproduce the failure, and it successfully turns the intermittent failure into a deterministic one in the flaky version, while no test failures occur in the fixed version.

Assume that the `OutputStream` object `actual` at Line 3 is written at some time T_0 by the method `EMPTY_PROPERTIES.save(...)` on Line 4. Then, the `OutputStream` object `expected` is written at some other time T_1 by the `new Properties().save(...)` call on Line 17. If T_0 and T_1 happen within the same second, the test passes; if they happen in different seconds due to delays in the test code or the code under test (CUT), the assertion on Line 18 fails, because it compares the entire `OutputStream` rather than only the first line. The pass/fail outcome thus depends solely on the wall-clock timing of the `save` call.

We are unable to reliably reproduce the flaky test failure in every run of the test. However, if we insert a delay, `Thread.sleep(2000)` at Line 15, 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 one second. As such, the TD flaky test failure now fails 100% of the time, making the flaky failure now 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 20– 21). Now, even if we introduce a delay at the same location (the red-commented `Thread.sleep(2000)` on Line 12), the test no longer fails, regardless of the amount of time between the two points T_1 and T_2.

We include this flaky test in our dataset, along with all the code and information needed to help a user compile and run the flaky test, for both the flaky version where the test could fail as well as the fixed version where the test should no longer fail. To help with confirming the flakiness behavior as well as it being fixed, we also include the changes needed to reliably reproduce the failure, namely the inserted `Thread.sleep` delay needed to make this test fail. In other words, for each flaky test in our dataset, we include four code versions: Flaky, Fixed, FlakyCodeChange (e.g., Flaky

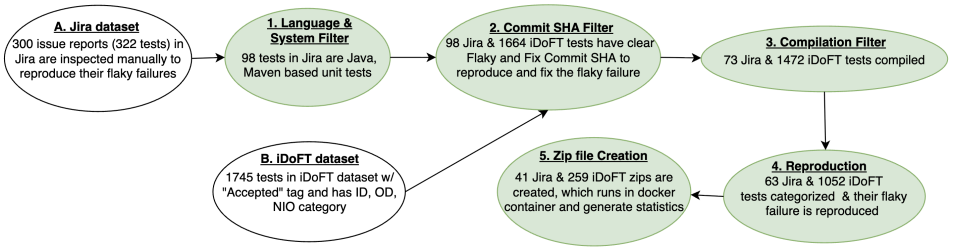

Fig. 2. High-level workflow of our methodology

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

4 ReproFlake Infrastructure and Artifact

We present ReproFlake [8], which is a dataset of reproducible flaky tests of multiple categories. We curate the dataset with flaky tests from a popular flaky test dataset, iDoFT [63], and issue reports from popular open-source projects posted in Jira. Our dataset consists of zip archives that provide dockerized environments to reproduce flaky-test failures across multiple categories, including Implementation-Dependent (ID), Order-Dependent (OD), Timing-Dependent (TD), and Non-Deterministic (NOD)tests.

4.1 Dataset Structure

For each Maven module[1] that contains a flaky test, ReproFlake has a zip archive for it for a particular code version. If a module has multiple flaky tests in it, only one zip archive is available for that, which includes the required data to reproduce all flaky tests from that module. The zip archive contains the following information to help reproduce the flaky test failures:

- Source code of the project at the version when the test/tests in the module were last found to be flaky so that we can work on the flaky version to reproduce the failure.
- The Maven dependency repository (i.e., the `.m2` directory) which contains all dependencies needed to compile and run the test for different code versions. We kept it in a single folder named `Flakym2` to reduce the data size, as there is often little to no difference in the dependencies among the different code versions due to their similarity.
- 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.
- The patch file, `FlakyCodeChange.patch`, which a user can apply to the existing flaky version, changing it into a version with higher rate of failure. This patch is only relevant for TD flaky tests as other categories have deterministic failure reproduction scripts.
- The patch file, `FixedCodeChange.patch`, which a user can apply to the fixed version of the code to see if it still fails or if the failure rate increases. (Only applicable for TD flaky tests.)
- An information file containing metadata for the flaky test, including the issue report link, the GitHub commit SHA, and the pull request link.

We provide a framework to automatically compile and run the test, reproducing the flaky failure deterministically. Our framework will use our zip archive and run the test in dockerized environment to generate a "results" folder containing the result of the test being run in different code versions, where each version has the test run logs, Maven Surefire reports, and a summary file that summarizes the results (number of failure/pass) of multiple test runs, as well as the coverage information for the test run.

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

Our work involves mining of two data sources: a) Jira Issue Report [13] and b) iDoFT dataset [63]. 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. In Figure 2, we can see that A and B represent the two datasets. We curate our reproducible dataset from these sources by following the steps outlined in the workflow diagram. The process starts with collecting data from Jira issue reports and iDoFT dataset, followed by filtering out tests based on language, relevant information availability, compilation, reproducibility, and zip creation process.

4.1.1 Jira Issue Report.

A. Jira Dataset (Input). 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" - either in the title, description, or comments. We used the Jira Python library [15] 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 in our artifact [8]. 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.

Inspection Methodology & 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, and updated the guideline as needed (in which case all reviewers were notified and updated their labeling accordingly). We make our guideline available online [6] so future work on flaky tests can follow it to create additional reproducible flaky test artifacts.

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 follow the guideline. In some cases, a single bubble may consist of multiple subsections of the guideline, but the section number always corresponds to the bubble number. For example, Language and System Filter in bubble 1 maps to Section 1.1 and 1.2 in our guideline.

Bubble 1 in the workflow diagram checks whether an issue involves a Java–Maven flaky unit test. Bubble 2 identifies the relevant commits and test information (i.e., module, fully qualified test name), Bubble 3 handles compilation, Bubble 4 classifies 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 unit test. Reviewers inspected the project repository to confirm the presence of a pom.xml file and Java source code, and analyzed the issue report, comments, and pull request details to determine whether the issue involves flaky unit-test behavior rather than integration or system-level flakiness. Issues failing this check were excluded from further analysis.

2. Commit SHA Filter. For valid Java, Maven-based flaky unit 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 first searched for the issue ID in the project's GitHub repository, examining 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. Compilation issues were mostly resolved by running Maven with multiple skip flags to avoid unnecessary checks, switching between Java 8, 11, and 17 to resolve version issues, replacing SNAPSHOT dependencies with the closest stable versions from Maven Central in

all pom.xml files, and updating repository URLs from http to https to ensure secure downloads. Issues that could not be compiled after these steps were excluded.

4. Reproduction. The main differences in reproducing various categories of flaky tests lies in the distinct methodologies used for their reproduction. More details about the flaky test category and tools to detect them are given in Section 2 of the paper. Below, we summarize the approaches used to reproduce each category of flaky test we came across while mining the Jira issue reports.

Timing-Dependent (TD) flaky tests: TD flaky tests are those that exhibit non-deterministic behavior due to variations in execution timing, such as thread scheduling, race conditions, or asynchronous operations. We tried to simulate the exact behavior that makes the test flaky by inserting a sleep either in the test code or the code under test. The major challenge here is where to add `Thread.sleep(delay)`. We performed manual analysis of the issue report along with its comment section, the pull request and commit in GitHub, the changed code to resolve the flakiness, the test method, code under test, and other associated code with this test, and then tried adding delay manually in different portions of the code and observed the test execution. Based on this analysis, we manually inserted Thread.sleep() near the code where timing or synchronization affects the test result, such as before assertions or around asynchronous operations.

If adding the delay caused a significant increase in the failure rate out of 10 runs compared to runs without the delay, we assumed the test to be a TD flaky test. We then increased the delay by 100 ms 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 did not label it as TD flaky test and considered it under other flakiness categories.

Order-dependent flaky test: Order-dependent flaky 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, which runs the tests in a test suite in random orders to expose flaky failures. Since iDFlakies relies on a customized test runner, which showed some incompatibilities with a few JUnit tests in our dataset, we instead made small changes to the Maven Surefire plugin, customizing it to allow running the tests in specific orders. We first gather all tests in a module and execute them in 20 randomized orders using the custom Maven Surefire plugin. If a test shows nondeterministic results (at least one pass and one fail across runs), we then rerun it in both a failing and 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 flaky 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 customized Maven Surefire plugin.

Implementation-Dependent (ID) Tests: ID tests are reproduced by exposing nondeterministic behavior in APIs or execution environments with tools such as NonDex [64], 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 reproduction step to guarantee deterministic reproduction.

If the category-specific tool fails, we run all other tools; if none succeed, we execute the test 100 times (Section 4.3 of the guideline). Tests that fail intermittently are labeled Unreliably Reproducible (with a presumed category when available, or Unclassified otherwise), while tests that never fail are marked Unreproducible.

5. Zip File Creation. To build zip archives for the different categories of flaky tests in Section 5 (bubble 5) of the guideline, 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 the more-flaky version for TD flaky tests only, and FixedCodeChange.patch for the developer-fix 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 statistics. We package the final setup into a ZIP archive and add it to our dataset for easy reproduction.

We add the generated zip archive that contains all the relevant information for our infrastructure by updating a central CSV configuration file. The CSV records key data for each flaky test, including its name, category, project module, corresponding ZIP and folder names, its OD-relevant test (for order-dependent cases), the number of iterations run, test configuration, and Java version used. Our infrastructure then uses the CSV and ZIP file to extract the project, place it in an appropriate Docker image for the test category, and run the tests (using tools/multiple reruns/code changes) on different code versions to gather results.

To ensure consistency across inspection, 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 category from the report, tools/code changes used, results from multiple runs (including 100-run statistics), logs and failure rates, final detected category, and supporting artifacts like patch files, NonDex seeds, test orders, and reproducibility status.

4.1.2 iDoFT Dataset.

B. iDoFT Dataset (Input). We also reproduced flaky tests from the International Dataset of Flaky Tests (iDoFT) [63], 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, Maven-based open-source projects. The methodology for reproducing flaky tests from iDoFT is illustrated in Figure 2, it starts from bubble B. Unlike the Jira dataset, the iDoFT dataset (i) already provides the flaky-test category through the *category* column, (ii) exclusively contains flaky unit tests from Java, Maven-based projects, allowing us to skip the language and system filtering step (bubble 1), (iii) includes only tests associated with accepted pull requests, and (iv) contains Non-Idempotent-Outcome (NIO) flaky tests.

We filtered 1745 flaky tests with *Accepted* status where the pull requests containing the fixes were accepted. After working with these flaky test we found 1052 flaky tests are reliably reproducible of categories ID, NIO, OD, and Unclassified (UD/NOD), and 693 flaky tests are excluded due to Compilation failure, Commit SHA not found, and Not reliably reproducible.

Inspection Methodology. For the iDoFT dataset, the commit SHA filtering step differed slightly. As the iDoFT dataset already contained accepted pull request link, we directly obtained the merged commit from the pull request as the fixed version and used its preceding commit as the flaky version, without requiring issue-based commit searches. 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. Also, for NIO tests, we used the custom testrunner provided by iDFlakies, configured to execute the same test twice within the same JVM. A test was classified as NIO if it passed on the first execution but failed on the second execution in the same JVM. All remaining steps

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

Outcome	Reason / Ctegory	# Tests
Jira Dataset		
Included	Timing-Dependent (TD)	36
Included	Implementation-Dependent (ID)	14
Included	Order-Dependent (OD)	4
Included	Unclassified	9
Included Total		**63**
Excluded	Non-Java/Maven project	66
Excluded	Not a flaky unit test	158
Excluded	Compilation failure	25
Excluded	Not reliably reproducible	10
Excluded Total		**259**
iDoFT Dataset		
Included	Implementation-Dependent (ID)	805
Included	Order-Dependent (OD)	122
Included	Non-Idempotent-Outcome (NIO)	125
Included Total		**1052**
Excluded	Commit SHA not found	81
Excluded	Compilation failure	191
Excluded	Not reliably reproducible	421
Excluded Total		**693**

- compilation filter, reproduction, zip file creation - followed the same workflow as for the Jira dataset.

4.2 Dataset Statistics

The dataset comprises 300 zip archive covering 1115 reproducible flaky unit tests across the categories ID, TD, OD, NIO and Unclassified. Each ZIP file may contain multiple flaky tests associated with a specific merged commit SHA and a particular module within the same project. Table 2 summarizes the composition of our final dataset, detailing how flaky tests from the Jira and iDoFT sources were filtered, reproduced, and included. It reports the number of tests per flaky-test category, along with the reasons for exclusion including compilation errors, missing reproduction metadata (e.g., commit or polluter information), and unreproducible failures. The unreproducible flaky tests mostly come from always passing or failing and not showing flakiness.

5 Experimental Setup and Evaluation

We address the following research questions to showcase the challenges with creating a flaky test dataset, provide insights into what future flaky test research should focus on and how they may do so, and the extensibility of our dataset.

RQ1: What are the challenges associated with reproducing and fixing of flaky-test failures?

RQ2: What are the characteristics of the reproducible and fixed flaky tests in our dataset?

- **RQ2.1**: What categories of flakiness are the reproducible and fixed tests?
- **RQ2.2**: What is the size and location of the code changes between flaky and fixed versions for different types of flaky tests?

- **RQ2.3**: What types of exceptions are most commonly observed from different types of flaky tests?

RQ3: What is the process to collect other information, e.g., test code coverage, with our dataset?

RQ1 is important because it highlights the challenges in creating a reproducible dataset, e.g., missing information or compilation issues, and offers guidance in how researchers should avoid such challenges. RQ2 explores what the reproducible and fixed flaky tests look like - what kind of flakiness they have, what code changes were made, what types of errors they cause, etc. These details will help researchers build better tools for detecting or fixing flaky tests. RQ3 looks at how useful our dataset is for collecting code coverage and running further analysis. By making it easier to gather this data, we hope to support future research and tool development around flaky tests. Our dataset opens several directions for future research on flaky tests detection and repair, which are described in later subsections.

Knowing the type of flakiness helps developers understand what causes it, e.g., timing, input, or order issues, and fix the flakiness correctly. Comparing code changes between flaky and fixed versions shows where and how much code was changed and whether the fix worked. The dataset offers a ready-to-use setup with Docker, and tools like JaCoCo to run tests, measure coverage, and compare versions. Our RQ results combine findings from both datasets.

5.1 RQ1: Challenges faced while reproducing flaky tests

We faced several challenges while trying to reproduce flaky tests. We could overcome some challenges, others we could not.

There were 216 compilation errors that we could not fix, but for some, we were able to resolve them by running Maven with multiple skip flags to avoid unnecessary checks, switching between Java 8, 11, and 17 to resolve version issues, replacing SNAPSHOT dependencies with the closest stable versions from Maven Central in all pom.xml files, and updating repository URLs from http to https to ensure secure downloads.

Other challenges consist of not always finding the issue reports related to the PRs in iDoFT dataset. We addressed this by automatically linking pull requests to issue IDs and manually inspecting cases where automated mapping failed.

A key challenge we overcame was identifying the fixing commit on the main branch corresponding to each pull request. We manually inspected GitHub to locate merged commits when available. Another challenge we overcame was reproducing TD flaky tests.

Using tools like FlakeRake [56] appeared to be difficult due to insufficient documentation and version compatibility issues. The tool required a specific version of Java to run, and our projects were not always compatible with it. Even when they were compatible, using this tool was very expensive, took a long time to get the exact place to inject delays for one TD test. We overcame these issues by manually investigating the code and finding appropriate places to inject the delays.

Few of the challenges that we could not resolve were in some cases while trying to find the merged commit from a fixed PR, pull requests were closed or the changes were incorporated differently, making it impossible to identify an exact fixing commit. Issue reports could not be linked to some PRs because they were unavailable. Also, the 216 compilation issues we could not solve, but tried to analyze the error categories in the next paragraph to understand them better. We did not apply automated compilation-repair tools (e.g., Maven dependency repair tools), leaving their evaluation as future work. In many cases (431 tests), we could not reproduce the flaky behavior, even after running tools that usually catch these issues. For 157 OD tests, 259 ID tests, and 5 NIO tests, tools that run tests in random order certain times did not help, possibly because the flaky order was not triggered.

Compilation Error Categories: As the majority of failures are compilation-related, we performed a compilation error categorization on the combined iDoFT and Jira datasets. Compilation errors are not always experimental design problems, these errors come from the projects' own outdated or unreachable libraries and repositories. We randomly sampled 100 build logs and manually assigned 15 fine-grained labels, each representing a distinct compilation error type (i.e., subcategory) corresponding to a recurring root cause (e.g., missing dependencies, Java version mismatches). Characteristic error messages and keywords from each label were used to construct regex rules and to build a few-shot prompt for GPT-4o-mini.

New logs were classified using a regex-first approach; logs that did not match any rule were classified by GPT using the few-shot examples selected from the manually labeled sample. The resulting subcategories were then grouped into higher-level categories based on shared failure causes (e.g., Dependency Resolution Errors, Plugin Execution Errors). Logs that remained unclassified were re-examined, leading to refined rules or new labels, and this process was repeated until most logs were covered and the categories stabilized.

Our **Dependency Resolution Error** analysis shows what the most dominant category of compilation errors is in both datasets. Dependency resolution errors account for a big portion of the errors - **80.0%** of the compilation errors (20 out of 25) in the Jira dataset and **50.8%** of the compilation errors (98 out of 191) in the iDoFT dataset. These errors occur when Maven is unable to find, resolve, or download required dependencies or artifacts needed during the build. Some common issues are missing artifacts, wrong or missing version numbers, problems reaching repositories, or issues caused by missing or broken parent POMs.

Among the dependency resolution errors, *Blocked or non-existent HTTP repositories* encompass **31.6%** of the errors, which are blocked or outdated HTTP repositories that prevent artifact downloads.

A close second problem (30.8%) is *Missing Dependency Version in POM*, where dependencies are declared without explicit version information, leaving the build system unable to resolve the correct artifact. Other issues include *Missing Dependencies*, where required libraries cannot be found, failed dependency resolution from version mismatches or network issues, dependency management conflicts between library versions, and non-resolvable parent POMs that break inheritance.

Open Research Challenge - *Improving Compilation of Legacy Projects*
We classify compilation error categories, which can guide future research in understanding and addressing common build issues encountered when compiling legacy projects used for flaky test research. Our findings show that most issues are dependency resolution errors, which occur when required libraries or repositories cannot be found or accessed, often due to missing dependency version information in POM files, outdated or blocked HTTP repositories, missing dependencies, or broken parent POM links. These findings suggest that improving dependency availability and updating repository links should be prioritized to resolve these build failures.

5.2 RQ2: Characteristics of Reproducible and Fixed Flaky Tests

Our goal is to understand the characteristics of our reproducible flaky test dataset. We aim to explain the categories of flaky tests we could reproduce, the location and size of the changes needed to fix these flaky tests, and categories of flaky exceptions. Our findings aim to provide insights into what future flaky test research should focus on and what such research can do to address reproduction and fixing challenges.

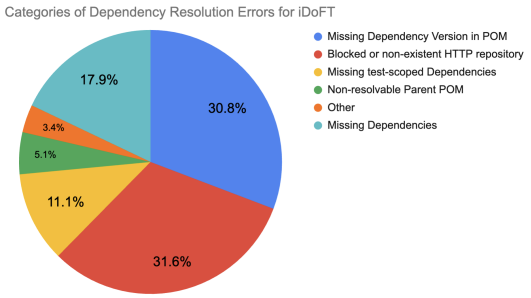

Fig. 3. Categories of Dependency Resolution Errors in our Dataset (N=216)

5.2.1 RQ2.1: Categories of Flakiness in Reproducible and Fixed Tests. Our reproducible dataset includes different categories of flaky tests - for iDoFT dataset, we reproduce ID, TD, OD, NIO flaky tests. For the Jira dataset, we are able to reproduce ID, OD, TD, and a few unclassified ones for Unknown reason. Table 2 has the breakdown of the type of flaky tests and the count for each category. The majority of the dataset consists of Implementation-Dependent flaky tests for the iDoFT dataset, whereas for Jira, it is mostly TD flaky tests. We reproduce **TD** (3.22%), **OD** (11.30%), **ID** (73.45%), **NIO** (11.21%), and **Unclassified** (0.81%) flaky tests, where the percentages are calculated across the entire set of 1115 reproducible tests from both iDoFT and Jira datasets.

5.2.2 RQ2.2: Size and Location of Code Changes Between Flaky and Fixed Versions. We analyze where fixes occur - test code versus code under test - and how large those fixes are across different flakiness categories to identify repair location patterns and guide future automated repair. By seeing patterns across different flaky types, developers can better predict the effort needed and quickly focus on the right part of the code.

As shown in Table 3, for OD, TD, UD, and NIO tests, fixes primarily modify the test code, with relatively small changes to the code under test. These fixes typically involve a limited number of lines and are localized to the test code. For ID tests, however, fixes show more variation. On average, ID fixes involve larger changes to the code under test than to the test code, indicating that some ID cases require substantial production code modifications. At the same time, the median ID fix changes more test code than production code, indicating that most ID fixes are test-centric, while a smaller number of large production code changes substantially increase the average. Nevertheless, our results suggest that ID flakiness can stem from both test-level issues and deeper implementation-level problems. Changes are measured as summation of additions and deletions of lines. Overall, 93.53% of ID, 28.57% of OD, 58.33% of TD, 99.2% of NIO, and 66.67% of Unclassified tests changed only one location - either the test code or the production code. Future research on fixing flaky tests can benefit from first focusing on which part of the code the fix is likely to go in before attempting different patches.

Open Research Challenge - *Prioritizing Location of Fixes*
Our findings in RQ2.2 show that most flaky tests, regardless of category, fixes mostly occur in the test code rather than code under test (Table 3). These findings can help developers prioritize the location of fix or guide LLM-based repair systems or automation repair tools to focus their fixes in the most likely location first. For example, if a flaky test is categorized as TD, OD or NIO, the developer/model can start by suggesting fixes in the test code (e.g., waits or synchronization for TD). If it is Implementation-Dependent (ID), they can also inspect the code under test for shared-state or order-sensitive logic as well as the test code.

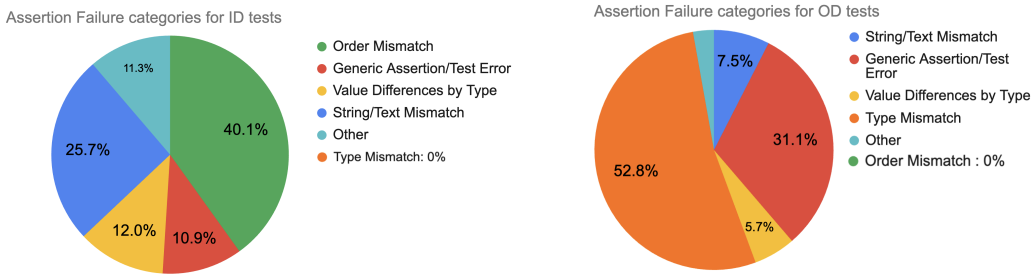

Fig. 4. Assertion error categories for ID (N=805) and OD (N=122) flaky tests in our dataset.

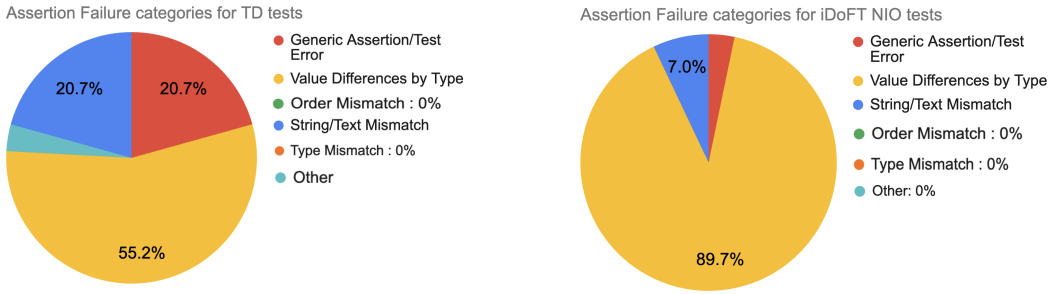

Fig. 5. Assertion error categories for TD (36) and NIO (125) flaky tests in our dataset.

Table 3. Average Change in Test Code and Code under Test across Flaky Types

Type	Avg Line Changes in Test Code	Avg Line Changes in Code under Test
ID	112.35	121.05
OD	11.06	1.59
NIO	52.18	0.13
TD	39.36	18.97
UD	46.00	11.67

5.2.3 RQ2.3: Categories of Flaky Exceptions across Flaky Categories. We categorize the flaky test failures for all types of flaky tests in our dataset. Knowing if certain error log types (e.g., Timeout, AssertionError, NullPointerException) consistently lead to certain fix strategies (e.g., adding waits, reordering tests, initializing state) can help guide what automated techniques should do to fix flaky tests using the error log. We followed the same approach described in Section 5.1 for categorizing compilation errors to classify the flaky 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 classify logs that did not match any rule.

The resulting subcategories were then systematically grouped into higher-level categories based on semantic similarity of failure causes (e.g., order mismatches, value mismatches, structural mismatches), and the classification was iteratively refined until the categories stabilized and covered most failure patterns.

In the iDoFT dataset, nearly all ID test failures result from *Assertion Errors* (99.4%). OD tests also fail mainly because of *Assertion Errors* (96.3%), with a few cases involving missing class definitions or RPC errors. A Remote Procedure Call (RPC) error occurs when a test calls a method on an external

system or service, but the call fails due to connection or setup issues. For NIO tests, *Assertion Errors* are also the primary cause (98.9%), with some errors due to global state conflicts or registry errors.

In the Jira dataset, TD tests mostly exhibit *Assertion Errors* (90.6%), with some *Timeouts* (6.3%) caused by delays in async operations and a few *NullPointer Exceptions* (3.1%). Both ID and OD tests show only *Assertion Errors*. In the *Unclassified* category, 66.7% failures are *Assertion Errors* and 33.3% are due to *Project Configuration* problems like broken build files or missing setup elements.

As most flaky errors are Assertion errors across all categories, we have further sub categorized the assertion errors. We can see in Figure 4 and 5, for **ID** tests, the most common issue is *order mismatch* (40.1%). In **OD** tests, *type mismatch* dominates (52.8%). For **TD** tests, *value difference by type* accounts for 55.2%, while in **NIO** tests, it is at 89.7%. Understanding these detailed assertion error types can help prioritize debugging and fix strategies.

Some of the prevalent subcategories are described below:

- **Order Mismatch** - The expected and actual elements are correct, but they appear in a different order
- **Type Mismatch** - The expected and actual values are of different types or classes (e.g., different data types, object types, or exception types).
- **Value Difference by Type** - The type matches, but the actual value differs (e.g., 58 vs. 57)
- **Structure Mismatch** - The overall data structure differs (e.g., list vs. set, or tree shape changes)
- **String/Text Mismatch** The expected and actual strings contain the same data but differ in text format or order (e.g., JSON fields appear in a different order)
- **Generic Assertion/Test Error** - A broad failure with little detail (e.g., `assertTrue` failing without context). They can indicate missing context or weak assertions

Example (ID vs. OD): Order or formatting differences indicate ID flakiness, while unexpected exceptions or behavior indicate OD flakiness.
ID → `expected:<{"a":1,"b":2}>, was:<{"b":2,"a":1}>`
OD → `expected:<RuntimeException>, was:<AssertionError>`

Open Research Challenge - *Inferring Flaky Test Types from Error Categories*
A potential use of RQ2.3 analysis is to take an arbitrary error message and using the observed frequency of flaky test categories in real world projects [39, 48], estimate the likelihood that the failure belongs to a particular flaky test category. If one type of error is dominant in more than one category of flaky tests, we can calculate the likelihood according to the frequency of that flaky test category in practice, and estimate which category the failure is most likely to belong to. Prior work such as FlakyCat [10] and FlakyDoctor [20] shows that knowing the flaky test category is useful for diagnosis and repair, but typically requires analyzing test code or observing test behavior across multiple executions. Our results show that for a nontrivial number of flaky tests, the category can be inferred using error log messages alone, which can subsequently be used for fixing flaky tests.

Open Research Challenge - *Providing New Information to Fix Flaky Tests*
It is possible to combine the information from RQ2.2 and RQ2.3 and feed an LLM the flaky error message, with error category, suggested fix location for that category, combined with additional context such as covered methods and issue report descriptions to automatically generate a repair. Our work provides issue report information (if available), as well as covered methods and line numbers for flaky and fixed versions. Unlike prior tools such as FlakyFix [26], which predicts fix types from test code, or FlakyDoctor [20], which requires known OD/ID labels and static localization, users can use the error category to determine category of flaky test and consequently repair the test.

5.3 RQ3: Process for Extending the Dataset with Additional Tools

We want to assess the framework's usability by showing how easily we can collect code coverage using the infrastructure. We can run tests, collect code coverage, and compare different versions using the scripts in our framework. We added other tools like JaCoCo and assessed its usability. To add tools, we provide a script that automatically modifies the POM files of projects (configuration file for Maven-based projects).

We used JaCoCo to obtain code coverage for 36 tests out of 63 tests from Jira issue reports and 590 tests out of 1115 tests from iDoFT dataset.

The average line coverage of the flaky version divided by the fixed version by type of flaky test is: 0.992 for ID, 1.0 for OD, 0.992 for TD, 0.997 for NIO, and 1.042 for Unclassified. Values near 1.0 show that fixes mainly stabilized execution without changing which code ran. Ratios < 1.0 (ID, TD, NIO) mean the fixed version covered slightly more lines, suggesting the fix enabled additional or previously skipped paths. Ratios > 1.0 (unclassified) indicate that the flaky version ran extra or redundant code. Overall, OD remained close to 1.0, showing that their fixes improved determinism or isolation rather than coverage. The fact that flaky and fixed versions cover almost the same code means that most problems are not what code runs, but how or when it runs (like timing, shared state, or order issues).

Open Research Challenge - *Comparing Flaky Test Fixes*
Another research direction is to compare LLM-generated fixes with developer-written patches from our dataset. Future studies can analyze differences in coverage, mutation score, failure rate, and lines of code changed to assess repair quality. Since LLMs can often generate multiple potential fixes, coverage data from our dataset can be used to identify which generated fix best matches real developer fixes. Comparing coverage similarity can help determine which LLM-generated fix is closest to common developer fixes.

6 Threats to Validity

Flaky tests can produce nondeterministic results due to multiple categories. In our dataset, we follow a systematic guideline to identify the correct categories of a flaky test. The actual categories of a flaky test may still change over time (e.g., fixed flaky test can become flaky again on certain days of the week). To help mitigate this threat, we ran our reproducible environment multiple times for each flaky test to ensure that the failures are reliably reproducible and provide the error logs of the failure and scripts used to obtain the failure.

The findings of our study may not generalize to other flaky test categories. Our dataset composed of four main categories of flaky tests, but some studies [23, 48] have reported 10+ categories of flaky tests. The categories of flaky tests we included in our dataset and study was dictated by the flaky

tests that we can find and reliably reproduce, which was dictated by what categories of flaky tests have automated tools to help with failure reproduction. Our literature review revealed automated reproduction tools only for the four categories that we included in our study. To help mitigate this threat, we include a systematic guideline in our artifact so others can more easily replicate our work for other flaky tests of the categories we studied or even for categories that we did not study.

7 Related Work

Datasets for improving software engineering research have a long tradition. From manually-curated datasets for test adequacy criteria [33] in 1994 to more recent and general datasets for software testing and analysis, such as SIR [1], Defects4J [3], and BugSwarm [5]. There has also been an increasing number of domain specific datasets, such as for test prioritization research[24], research related to Android apps [11], continuous integration related research [19], and for bug prediction research [21].

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. This makes it well-suited for studying nondeterministic test behavior and repair. A widely used flaky test dataset is iDoFT [63], which consists of list of flaky tests, their GitHub URL, flaky commit, their pull request information, but lacks reproducibility artifacts such as error logs, issue reports, and executable environments (Table 1). FlakeFlagger [12] includes a different set of flaky tests than iDoFT and extends the information by including build and failure logs, but still does not provide fixed versions or reproducible environments (Table 1). Similarly, FlakyFix [26] and DeFlaker [18] include flaky tests but lack key artifacts such as fixed versions, logs, and reproducibility support. FlakyCat [10] categorizes flaky tests across multiple dimensions but does not include reproducibility artifacts or complete debugging information. Keila et al. [17] provide issue reports and categorization, but do not include flaky or fixed versions or reproducible setups.

Compared to all of these datasets, ReproFlake is the only dataset (Table 1) that simultaneously provides flaky and fixed versions, reproducible environments, detailed error logs, pull requests, and issue reports, enabling end-to-end reproducibility and deeper analysis of flaky test behavior.

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 present ReproFlake, which is a dataset of 1115 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 flaky test failure, (2) fix that was accepted by developers, (3) error log that details the message and stack trace of the flaky test failure, and (4) pull request and issue report information. Using this dataset, we studied the challenges in reproducing flaky test failures, the characteristics of flaky tests, and the difficulty in using our dataset to collect additional information. Our study revealed several 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 [8] and provide guidelines for transparency in how we constructed the dataset and to enable future work to expand it as needed.

9 Data Availability

All relevant data, scripts, and results generated for this work are available on our website [8].

References

[1] 2005. SIR. https://sir.csc.ncsu.edu/portal/index.php. SIR Dataset.
[2] 2014. .
[3] 2014. Defects4J. https://github.com/rjust/defects4j. Defects4J Dataset.
[4] 2017. .
[5] 2019. BugSwarm. https://www.bugswarm.org/dataset. Bug Swarm Dataset.
[6] 2025. Guideline. https://docs.google.com/document/d/1VcORh2Otr7iYqbIXlwVW7ikPnlEV8bOF. Accessed: January 2026.
[7] 2026. . Accessed 2026.
[8] 2026. ReproFlake: A Reproducible Dataset of Flaky Tests. https://sites.google.com/view/reproflakedataset. Accessed: January 2026.
[9] A machine learning solution for detecting and mitigating flaky tests 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.
[10] 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*.
[11] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *MSR*.
[12] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting flakiness without rerunning tests. In *International Conference on Software Engineering*.
[13] Apache Software Foundation. 2025. Apache JIRA Issue Tracker. https://issues.apache.org/jira. Accessed: January 2026.
[14] Apache Software Foundation. 2025. COLLECTIONS-812. https://issues.apache.org/jira/browse/COLLECTIONS-812. Issue report.
[15] Atlassian. 2024. *Jira Python Library*. https://jira.readthedocs.io/ Accessed: January 2026.
[16] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.* (2019).
[17] Keila Barbosa, Ronivaldo Ferreira, Gustavo Pinto, Marcelo d'Amorim, and Breno Miranda. 2023. Test Flakiness Across Programming Languages. *IEEE Transactions on Software Engineering* 49, 4 (2023).
[18] 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*.
[19] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *MSR*.
[20] Yang Chen and Reyhaneh Jabbarvand. 2024. Neurosymbolic Repair of Test Flakiness. In *ISSTA 2024: Proceedings of the 2024 International Symposium on Software Testing and Analysis*.
[21] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An Extensive Comparison of Bug Prediction Approaches. In *MSR*.
[22] Docker 2026. Docker. https://www.docker.com.
[23] 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*.
[24] Emad Fallahzadeh and Peter C. Rigby. 2022. The impact of flaky tests on historical test prioritization on chrome.. In *ICSE SEIP*.
[25] Sakina Fatima, Taher A. Ghaleb, and Lionel Briand. 2023. Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests. *Transactions on Software Engineering* (2023).
[26] 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*.

[27] Flakiness Dashboard HOWTO 2026. Flakiness dashboard HOWTO. http://www.chromium.org/developers/testing/flakiness-dashboard.
[28] Flaky tests (and how to avoid them) 2026. Flaky tests (and how to avoid them). https://engineering.salesforce.com/flaky-tests-and-how-to-avoid-them-25b84b756f60.
[29] 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.
[30] 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)*.
[31] 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 *ICSME*.
[32] 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.
[33] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*.
[34] 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 *ICSE 2017: Proceedings of the 39th International Conference on Software Engineering*.
[35] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs, See [2].
[36] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. 2020. Modeling and ranking flaky tests at Apple. In *ICSE SEIP 2020: Proceedings of the 42nd International Conference on Software Engineering, Software Engineering in Practice Track*.
[37] 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 *ESEC/FSE 2019: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
[38] 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*.
[39] Wing Lam, Kivanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A Study on the Lifecycle of Flaky Tests. In *International Conference on Software Engineering*.
[40] Wing Lam, Kivanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *ICSE 2020: Proceedings of the 42nd International Conference on Software Engineering*.
[41] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST 2019: 12th International Conference on Software Testing, Verification and Validation*.
[42] 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*.
[43] 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*.
[44] 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*.
[45] Chengpeng Li, M. Mahdi Khosravi, Wing Lam, and August Shi. 2023. Systematically producing test-orders to detect order-dependent flaky tests. In *ISSTA 2023: Proceedings of the 2023 International Symposium on Software Testing and Analysis*.
[46] Chengpeng Li, Chenguang Zhu, Wenxi Wang, and August Shi. 2022. Repairing Order-Dependent Flaky Tests via Test Generation. In *ICSE 2022: Proceedings of the 44th International Conference on Software Engineering*.
[47] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. *SIGPLAN Not.* (2016).
[48] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE 2014: Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*.
[49] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *International Symposium on Foundations of Software Engineering*.
[50] MediumExpedia 2026. Expedia – Fixing flaky time based unit tests. https://medium.com/expedia-group-tech/fixing-flaky-time-based-unit-tests-176accf5096e.
[51] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing, See [4].
[52] John Micco. 2026. Continuous integration at Google scale. https://www.slideshare.net/JohnMicco1/2016-0425-continuous-integration-at-google-scale.

[53] Netflix automation talks - Test automation at scale 2026. Netflix automation talks - Test automation at scale. https://youtu.be/FrBN94gUn_I?t=764.
[54] 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 *ESEC/FSE 2018: Proceedings of the 2018 12th Joint Meeting on Foundations of Software Engineering*.
[55] Shanto Rahman, Bala Naren Chanumolu, Suzzana Rafi, August Shi, and Wing Lam. 2025. Ranking Relevant Tests for Order-Dependent Flaky Tests. In *ICSE 2025: 47th International Conference on Software Engineering*.
[56] 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*.
[57] Shanto Rahman and August Shi. 2024. FlakeSync: Automatically Repairing Async Flaky Tests. In *International Conference on Software Engineering*.
[58] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST 2016: 11th International Conference on Software Testing, Verification and Validation*.
[59] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *ESEC/FSE 2019: Proceedings of the 2019 13th Joint Meeting on Foundations of Software Engineering*.
[60] Pavan Sudarshan. 2026. No more flaky tests on the Go team. http://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team.
[61] Test verification 2026. Test verification. https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification.
[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 *ICSE*.
[63] University of Illinois at Urbana-Champaign. 2021. International Dataset of Flaky Tests (IDoFT). http://mir.cs.illinois.edu/flakytests. Accessed: January 2026.
[64] University of Illinois at Urbana-Champaign. 2022. NonDex. https://github.com/TestingResearchIllinois/NonDex. Accessed: January 2026.
[65] 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*.
[66] Eric Wendelin. 2026. Introducing flaky test mitigation tools. https://blog.gradle.org/gradle-flaky-test-retry-plugin.
[67] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why?. In *ESEC/FSE '99: Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*.
[68] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption, See [2].
[69] Celal Ziftci and Jim Reardon. 2017. Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at Google scale, See [4].