What Do Developer-Repaired Flaky Tests Tell Us About the Effectiveness of Automated Flaky Test Detection?

Owain Parry
University of Sheffield
UK

Gregory M. Kapfhammer
Allegheny College
USA

Michael Hilton
Carnegie Mellon University
USA

Phil McMinn
University of Sheffield
UK

ABSTRACT

Because they pass or fail without code changes, flaky tests cause serious problems such as spuriously failing builds and the eroding of developers' trust in tests. Many previous evaluations of automated flaky test detection techniques do not accurately assess their usefulness for the developers who identify the flaky tests to repair. This is because researchers evaluate detection techniques against baselines that are not derived from past developer behavior or against no baselines at all. To study the effectiveness of an automated test rerunning technique, a common baseline for other approaches to detection, this paper uses 75 commits — authored by human software developers — that repair test flakiness in 31 real-world Python projects. Surprisingly, automated rerunning detects the developer-repaired flaky tests in only 40% of the studied commits. This result suggests that automated rerunning does not often find those flaky tests that developers fix, implying that it makes an unsuitable baseline for assessing a detection technique's usefulness for developers.

CCS CONCEPTS

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

KEYWORDS

Software Testing; Flaky Tests; Automated Detection.

ACM Reference Format:
Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2022. What Do Developer-Repaired Flaky Tests Tell Us About the Effectiveness of Automated Flaky Test Detection?. In *IEEE/ACM 3rd International Conference on Automation of Software Test (AST '22), May 17–18, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524481.3527227

1 INTRODUCTION

Flaky tests — test cases that pass and fail without code changes [31] — are a major burden to software developers [17, 30]. When surveyed, 79% of developer participants characterized flaky tests as a moderate or serious problem, citing time wasted debugging spurious failures and an erosion of trust in test case outcomes [15].

Developers report that repairing flaky tests is cumbersome [15, 33]. Despite this, many methodologies for evaluating automated flaky test detection techniques do not accurately assess their usefulness for the developers who decide which flaky tests to repair. To ensure that flaky test detectors enhance developer productivity, it is imperative to evaluate any technique from their perspective.

One common methodology is to calculate the recall of a detection technique against a baseline of flaky tests identified by *automated rerunning*, the most straightforward method for detecting flaky tests [9, 11, 34]. In this context, recall is the percentage of flaky tests in the baseline that are detected by the technique under evaluation. We refer to this as a *rerun-based* methodology. However, this approach is limited because no prior work has studied how well this baseline assesses a technique's usefulness for software developers.

Another methodology is to simply present the number of detected flaky tests [26, 35, 40]. Yet, this is limited because there is no baseline to assess recall against [40]. Following Harman and O'Hearn position that it is safer to assume that all tests are flaky [23], evaluating a technique this way gives no indication of its usefulness.

Given these limitations, we performed a study to demonstrate the value of a *developer-based* methodology. It features a baseline of developer-repaired flaky tests that is more suitable for assessing a technique's usefulness for developers. This is because developers allocated time to repair these flaky tests, implying they were of interest. This is not to say that flaky tests of a baseline derived from rerunning would not be of interest to developers, but rerunning alone cannot evidence this property as past developer behavior can.

Leveraging a baseline of 75 flakiness-repairing commits that originated from 31 open-source Python projects hosted on GitHub, this paper addresses the following two research questions:

RQ1: What is the recall of automated rerunning against our baseline? We developed our own automated rerunning framework and applied it to the flaky tests in each commit just before the fix. We found that it detected the repaired flaky tests in only 40% of the commits. Such a low recall suggests that automated rerunning is not that helpful for developers. This implies that a rerun-based methodology is unsuitable for assessing developer usefulness, highlighting the value of supplementing it with a developer-based methodology.

RQ2: What causes the flaky tests in our baseline and how did developers repair them? For a deeper understanding of our baseline, we manually classified the causes of the flaky tests and the

AST '22, May 17–18, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9286-0/22/05…$15.00
https://doi.org/10.1145/3524481.3527227

developers' repairs in the commits. We found that randomness was the most common cause and that widening the range of acceptable states and outputs in assertions was the most common repair.

In summary, the contributions offered by this paper's study are:
1. Evaluation. Ours is the first study to evaluate automated rerunning under a developer-based methodology. Central to this is our baseline of 75 flakiness-repairing commits from 31 Python projects. We make this available in this paper's replication package [5].
2. Implications. This paper's results can both help developers decide whether or not to adopt automated flaky test rerunning and aid researchers who want to effectively evaluate new detection techniques. Our evaluation was also facilitated by an automated rerunning framework, available in the replication package [5].

2 BACKGROUND

Causes of Flakiness. Luo et al. [29] performed one of the earliest empirical studies of flaky tests. They categorized the cause of flakiness-repairing commits using the following ten categories:
1. Async. Wait. Test case makes an asynchronous call but does not adequately wait for it to finish, leading to intermittent failures.
2. Concurrency. Test case invokes multiple threads that interact in an unsafe or unanticipated manner, for example, race conditions.
3. Floating Point. Test case uses floating point operations and is flaky due to discrepancies such as non-associative addition.
4. Input/Output. Test case interacts with the filesystem and is flaky due to specific intermittent issues such as naming conflicts.
5. Network. Test case depends on the availability of a network and is flaky when the network or resource is unavailable or busy.
6. Order Dependency. Test case depends on a shared value or resource that is modified by other test cases as a side-effect.
7. Randomness. Test case uses (or covers code that uses) random number generators and is flaky due to either not setting seeds or not anticipating the full range of the generator's potential outputs.
8. Resource Leak. Test case does not release acquired resources, such as a database connection, inducing intermittent failures for itself or for other test cases that require the same resources.
9. Time. Test case relies on measurements of date and/or time. Flakiness is caused by, for instance, discrepancies in precision and representation of time across libraries and platforms.
10. Unordered Collection. Test case assumes a deterministic iteration order of an unordered collection type object, such as a `set`, leading to intermittent failures when the assumption does not hold.

Automated Rerunning. The most straight-forward flaky test detection technique is automated rerunning, which involves repeatedly executing test cases to observe inconsistent outcomes [31]. To detect more flaky tests, rerunning frameworks can introduce noise into the execution environment [37]. Examples of this include shuffling the test run order [40], randomizing standard library implementations [35], and introducing artificial CPU stress [36]. More sophisticated detection techniques may involve rerunning as a subcomponent [13, 18, 26] or as a means to train machine learning models [9, 12, 32, 34]. Rerunning can also be used by developers as a mitigation technique. For example, `flaky` [1] is a plugin for the `pytest` testing framework that will rerun a failing flaky test. In this paper, when we refer to rerunning, we refer to the detection technique as opposed to the mitigation technique.

3 METHODOLOGY

Baseline. To build our baseline of developer-repaired flaky tests, we searched for commits using the query "`flaky OR flakey OR flakiness OR flakyness OR flakeyness OR intermittent`" in the top-1,000 Python repositories on GitHub by number of stars. We decided to target Python since many previous studies tend to focus overwhelmingly on Java [10, 20, 27]. Upon finding matching commits, we inspected their commit messages and code diffs to determine if they repaired at least one flaky test. We considered up to six commits per project, improving the generalisability of our results by enabling the examination of more projects. For each commit, we cloned the repository at the state of its preceding commit, known as its *parent* commit, just before the repair. We then attempted to build it in a virtual environment [7] and execute its test suite. If we could not, then we were unable to use it. For older commits, this could be due to a dependency on an old version of a package no longer hosted on the Python Package Index (PyPI) [3]. To promote the reproducibility of this paper's results, we stipulated that we must be able to ensure a deterministic build. To that end, projects could only depend on packages available on PyPI. This allows us to capture and reproduce the exact environment of a build by recording the names and exact versions of each package as reported via a call to `pip freeze` [4]. Following this process, we arrived at 75 commits from 31 projects. Table 1 lists the names of each repository along with the number of its commits we used.

RQ1: What is the recall of automated rerunning against our baseline? To address this question, we developed our own automated rerunning framework called ShowFlakes [6]. It can introduce four types of noise into the execution environment during reruns. The first is to randomly deprioritize threads to expose concurrency bugs [28] and manifest flaky tests of the Async. Wait and Concurrency categories. The second is to randomly execute other test cases from the test suite and shuffle the order to identify flaky tests of the Order Dependency category. The third is to limit the upload and download speed of the network to a random value between 1 kilobyte/s and 1 megabyte/s to increase the probability of detecting Network flaky tests. The fourth is to vary the minor version of the Python interpreter between reruns (from versions 3.5 up to 3.9) to identify flaky tests that may only fail on one specific platform [15]. For each commit, we manually identified the Python versions that the project was declared to support by its developers and restricted ShowFlakes to just those. To find this information, we checked the project's `tox` configuration file [8], its setup script, and any continuous integration configuration files and build scripts.

For each commit, we used ShowFlakes to rerun the developer-repaired flaky tests at the state of the parent 1,000 times with no noise and 1,000 times with all four types of noise together. In each case, it counts a commit as "detected" and a true-positive if it can detect at least one of its flaky tests and a false-negative if it cannot. Recall is then the number of true-positives over the total number of commits. Because our baseline contains only positive examples, we cannot calculate true-negatives, false-positives, or precision.

RQ2: What causes the flaky tests in our baseline and how did developers repair them? We manually classified the causes of the test flakiness and the developers' repairs in the 75 commits. For the causes, we used the categories of Luo et al. [29] (see §2).

Table 1: The recall of automated rerunning against the baseline of flakiness-repairing commits. Each row shows the repository name, the number of commits we used, the number where at least one repaired flaky test was detected after 1,000 reruns with no noise, and the number detected after 1,000 reruns with all four types of noise together.

		Detected Commits	
GitHub Repository	**Commits**	**No Noise**	**Noise**
apache/airflow	4	2	3
HIPS/autograd	1	-	1
mahmoud/boltons	1	-	-
celery/celery	1	1	1
django/channels	1	-	1
quantumlib/Cirq	5	1	2
wandb/client	2	-	1
home-assistant/core	6	3	3
dask/dask	1	-	-
spesmilo/electrum	1	-	-
robinhood/faust	3	1	2
pallets/flask	1	-	-
pallets/flask-sqlalchemy	1	-	-
geopy/geopy	1	-	-
graphql-python/graphene	1	1	1
HypothesisWorks/hypothesis	6	1	2
dpkp/kafka-python	1	-	-
matplotlib/matplotlib	3	-	-
mitmproxy/mitmproxy	2	1	1
pandas-dev/pandas	6	1	2
zalando/patroni	1	-	-
OpenMined/PySyft	2	-	2
giampaolo/psutil	1	-	-
pytest-dev/pytest	4	-	-
saltstack/salt	1	-	-
scipy/scipy	4	-	2
searx/searx	1	-	1
matrix-org/synapse	2	1	1
tornadoweb/tornado	3	1	-
python-trio/trio	4	1	2
urllib3/urllib3	4	1	2
Total	75	16 (21%)	30 (40%)

Three authors of this paper categorized each commit independently, examining the code diff, any comments, and any linked issues. They then discussed their results and arrived at a consensus for the final categories, following a process of qualitative negotiation [38, 39]. To classify repairs without an initial set of categories, we followed a more exploratory methodology. Initially, one author inspected each commit and devised a minimal set of repair categories. Using these categories, and being allowed to create new ones, two other authors categorized each commit individually. These three authors then met to discuss the final category of each commit while agreeing on the final set of repair categories and their specific definitions.

Threats to Validity. We identified several threats to the validity and generalisability of this paper's findings and, where possible, took steps to mitigate them. First, the commits we used may have been unsuccessful at repairing the flakiness or may have only partially repaired it. Yet, we used these commits as a baseline to assess the usefulness of automated rerunning for developers identifying flaky tests to repair. Therefore, it does not matter how successful the repairs were, but rather that developers allocated time and resources in an attempt to do so. Second, the projects that we used may not be representative of open-source Python projects. To alleviate this threat, we used popular projects on GitHub, rather than from a specific organization, and limited the number of commits to six per project to limit any bias arising from using too few projects. Third, when calculating recall for **RQ1**, some of the undetected commits may have actually been detected had we set ShowFlakes to perform more reruns. There is little that we can do to mitigate this since there is no upper limit on the number of reruns required to manifest a flaky test [9]. It is also our judgement that 1,000 reruns is sufficient for assessing developer usefulness, since any more would likely be impractical [24]. Finally, our manual categorization for **RQ2** could have been impacted by our personal biases. To minimize this threat, three authors performed the task independently without communicating their opinions before the final discussion.

Table 2: The results of our manual categorization of the 75 flakiness-repairing commits. We gave each commit a category describing the cause of the flakiness (rows) and another describing the repair applied by the developers (columns).

Cause	**Add Mock**	**Add/Adjust Wait**	**Guarantee Order**	**Isolate State**	**Manage Resource**	**Reduce Random.**	**Reduce Scope**	**Widen Assertion**	**Miscellaneous**	**Total**
Async. Wait	1	6	-	-	-	-	-	2	-	9
Concurrency	-	2	-	-	2	-	-	2	2	8
Floating Point	-	-	-	-	-	-	-	3	-	3
I/O	-	-	-	-	-	-	-	-	-	-
Network	3	3	-	-	1	-	-	-	1	8
Order Dependency	-	-	-	2	-	-	1	-	-	3
Randomness	-	-	-	-	-	6	-	4	1	11
Resource Leak	-	-	-	-	2	-	1	1	-	4
Time	5	-	-	-	-	-	1	1	2	9
Unordered Coll.	-	-	3	-	-	-	-	-	-	3
Miscellaneous	2	-	1	-	1	-	-	6	7	17
Total	11	11	4	2	6	6	3	19	13	75

4 RESULTS

RQ1: What is the recall of automated rerunning against our baseline? Table 1 shows for how many of the 75 commits could automated rerunning detect at least one of the flaky tests. For each commit, we applied rerunning to only the developer-repaired flaky tests at the state just before the fix and had it perform 1,000 runs with no noise and 1,000 with all types of noise. Without noise, rerunning detected the flaky tests in 21% of the commits. With noise, it performed better — but still only achieved a recall of 40%.

RQ2: What causes the flaky tests in our baseline and how did developers repair them? Table 2 shows the results of our manual categorization of the commits. Each cell shows the frequency of every pair of cause and repair categories. The overall frequencies

```
  def test_example_runs_quantum_teleportation():
      expected, teleported = examples.quantum_teleportation.main()
-     assert np.all(np.isclose(expected, teleported, rtol=1e-4))
+     assert np.all(np.isclose(expected, teleported, atol=1e-4))
```

Figure 1: A commit of the Randomness cause category and the Widen Assertion repair category. This flaky test from the Cirq project [2] was repaired by a developer who adjusted the type of tolerance used in the approximate assertion statement. The source of the test data, `examples.quantum_teleportation.main`, uses a random number generator, indicating that the assertion was previously too conservative to accommodate all possible output values.

for the causes and repairs can be found in the left-most column and the bottom-most row, respectively. For the repairs, these eight categories emerged as part of our exploratory methodology:

1. Add Mock. Replace a function with a mock [16, 41] that mimics the behavior of the original in a controlled way.

2. Add/Adjust Wait. Add or adjust a time delay (i.e., an explicit wait for an event or a set amount of time) in the test code.

3. Guarantee Order. Guarantee a deterministic iteration order for a collection object by ensuring that it is a type that maintains an order [20, 35] and that it is sorted before being used in assertions.

4. Isolate State. Eliminate shared program state between test cases [18, 21] by, for example, not using global variables.

5. Manage Resource. Ensure that external resources are properly managed by, for instance, closing files and sockets.

6. Reduce Randomness. Make a test case that involves randomness more deterministic by, for example, setting a generator seed.

7. Reduce Scope. Hone the set of behaviors and properties checked by a test case to potentially make it less brittle [25].

8. Widen Assertion. Adjust a test case's assertion statement to make it accept a broader range of values and states [14, 15].

The most common cause category was Randomness, which developers addressed in 15% of the commits. This was followed by Async. Wait and Time both at 12% each and then Concurrency and Network both at 11% each. The most common repair category was Widen Assertion, which developers applied in 25% of the commits. Add Mock and Add/Adjust Wait were tied for second place at 15% each, followed by Manage Resource and Reduce Randomness both at 8% each. See Figure 1 for an example of a commit of the Randomness cause and Widen Assertion repair categories.

5 IMPLICATIONS

Developers. For the flaky tests that were repaired by developers in the studied commits, we can say that they allocated time to do so and so they must have been of interest. The results for **RQ1** showed that automated rerunning with noise achieved a recall of only 40% against these commits. This is not to say that the flaky tests that could be detected by rerunning, but that are not part of our baseline, would not be of interest to developers. Yet, we cannot say anything about those flaky tests, since we do not know if developers would assign time to repair them. Still, we demonstrated that the intersection between the flaky tests detected by rerunning and the flaky tests that we *can* say were of interest to developers is small. This implies that, for developers, the usefulness of rerunning is limited — especially given its prohibitive runtime cost [9, 11, 26].

We found that rerunning with noise performed significantly better than without. This finding supports the results of Silva et al. [36], who found that flaky tests fail more often in noisy execution environments. It is also echoed by Habchi et al. [22], who found that developers expressed a need for reruns under diverse system configurations. Therefore, if developers do wish to use rerunning to detect flaky tests, we strongly recommend doing so with noise.

Researchers. It is common in previous studies [9, 11, 34] to evaluate detection techniques using the flaky tests identified by automated rerunning as a baseline. We found that rerunning did not perform well against the baseline of developer-repaired flaky tests. Transitively, this implies that a baseline provided by rerunning would be unsuitable for assessing developer usefulness, given that this property is what our baseline aims to assess. However, we appreciate the manual effort required to collect a set of flakiness-repairing commits and evaluate a technique against them. For these reasons, we would advise researchers to supplement their existing evaluation methodologies with a developer-based methodology.

6 RELATED WORK

Luo et al. [29] performed one of the most frequently cited empirical studies of flaky tests. They introduced ten cause categories to classify commits that repaired them. We used these categories in our study to answer **RQ2**. Eck et al. [15] used surveys to examine test flakiness from the view of software developers. They introduced four further categories, including Platform Dependency for flakiness stemming from assumptions about the execution platform. Gruber et al. [19] used automated rerunning to detect 7,571 flaky tests in Python projects. Of these, they randomly sampled 100 and classified their causes using the categories introduced by Luo et al. and Eck et al.. Like us, they found Randomness to very prevalent.

Researchers have proposed many techniques for detecting flaky tests. Bell et al. [11] introduced DeFlaker for detecting flaky tests whose outcome changes across consecutive versions of a project but that do not cover modified code. They found that it could detect 96% of the flaky tests they had previously identified using rerunning. This would be an example of a rerun-based evaluation methodology. Dutta et al. [13] presented the FLASH tool for identifying flaky tests specific to machine learning and probabilistic projects. As part of their evaluation, they assessed their technique's performance against a baseline of developer-repaired flaky tests. This would be an example of a developer-based methodology, though in comparison to this paper, the authors used a smaller baseline of 11 commits.

7 CONCLUSIONS

Given the challenges that flaky tests pose for developers, prior research has presented and evaluated many methods for finding them. This paper demonstrated the value of a developer-based methodology for evaluating automated flaky test detection techniques. It involves calculating the recall of a technique against a baseline of developer-repaired flaky tests. For this baseline, we collected 75 flakiness-repairing commits from 31 Python projects hosted on GitHub. We evaluated automated rerunning against our baseline and found that it attained a recall of only 40%. For developers, this indicates that the usefulness of rerunning is limited. For researchers, this implies that a baseline provided by rerunning is unsuitable for assessing the usefulness of a detection technique for developers.

REFERENCES

[1] 2022. *Box/Flaky: Plugin for Nose or Pytest That Automatically Reruns Flaky Tests.* https://github.com/box/flaky

[2] 2022. *Cirq, Commit Fixing Test Flakiness.* https://github.com/quantumlib/Cirq/commit/0d1eacc456ada78de0815446905e9c48254cee6b

[3] 2022. *Find, Install and Publish Python Packages with the Python Package Index.* https://pypi.org/

[4] 2022. *Pip Freeze.* https://pip.pypa.io/en/stable/cli/pip_freeze/

[5] 2022. *Replication Package.* https://github.com/flake-it/showflakes-framework

[6] 2022. *ShowFlakes.* https://github.com/flake-it/showflakes

[7] 2022. *Virtual Environments and Packages.* https://docs.python.org/3/tutorial/venv.html

[8] 2022. *Welcome to the Tox Automation Project.* https://tox.wiki/en/stable/

[9] A. Alshammari, C. Morris, M. Hilton, and J. Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *Proceedings of the International Conference on Software Engineering (ICSE).*

[10] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya. 2015. Efficient Dependency Detection for Safe Java Test Acceleration. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE).* 770–781.

[11] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *Proceedings of the International Conference on Software Engineering (ICSE).* 433–444.

[12] A. Bertolino, E. Cruciani, B. Miranda, and R. Verdecchia. 2021. Know Your Neighbor: Fast Static Prediction of Test Flakiness. *IEEE Access* 9 (2021), 76119–76134. Issue 4.

[13] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and Misailovic S. 2020. Detecting Flaky Tests in Probabilistic and Machine Learning Applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).* 211–224.

[14] S. Dutta, A. Shi, and Misailovic S. 2021. FLEX: Fixing Flaky Tests in Machine Learning Projects by Updating Assertion Bounds. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* 211–224.

[15] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE).* 830–840.

[16] M. Fazzini, A. Gorla, and A. Orso. 2020. A Framework for Automated Test Mocking of Mobile Apps. In *Proceedings of the International Conference on Automated Software Engineering (ASE).* 1204–1208.

[17] M. Fowler. 2011. Eradicating Non-Determinism in Tests, https://martinfowler.com/articles/nonDeterminism.html.

[18] A. Gambi, J. Bell, and A. Zeller. 2018. Practical Test Dependency Detection. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST).* 1–11.

[19] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser. 2021. An Empirical Study of Flaky Tests in Python. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST).*

[20] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov. 2015. NonDex: A Tool for Detecting and Debugging Wrong Assumptions on Java API Specification. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE).* 223–233.

[21] A. Gyori, A. Shi, F. Hariri, and D. Marinov. 2015. Reliable Testing: Detecting State-Polluting Tests to Prevent Test Dependency. In *Proceedings of the International Conference on Software Testing and Analysis (ISSTA).* 223–233.

[22] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. Le Traon. 2022. A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST).*

[23] M. Harman and P. O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM).* 1–23.

[24] K. Herzig. 2016. *Let's Assume We Have to Pay for Testing.* https://www.slideshare.net/kim.herzig/keynote-ast-2016

[25] C. Huo and J. Clause. 2014. Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE).* 621–631.

[26] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. 2019. IDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST).* 312–322.

[27] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov. 2020. Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects. In *Proceedings of the International Conference on Software Reliability Engineering (ISSRE).* 403–413.

[28] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye. 2019. Efficient Scalable Thread-Safety-Violation Detection: Finding Thousands of Concurrency Bugs during Testing. In *Proceedings of the Symposium on Operating Systems Principles (SOSP).* 162–180.

[29] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE).* 643–653.

[30] J. Micco. 2016. Flaky Tests at Google and How We Mitigate Them, https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html.

[31] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2021. A Survey of Flaky Tests. *Transactions on Software Engineering and Methodology* 31, 1 (2021), 1–74.

[32] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2022. Evaluating Features for Machine Learning Detection of Order- and Non-Order-Dependent Flaky Tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST).* to appear.

[33] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2022. Surveying the Developer Experience of Flaky Tests. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).*

[34] G. Pinto, B. Miranda, S. Dissanayake, M. D. Amorim, C. Treude, A. Bertolino, and M. D'amorim. 2020. What is the Vocabulary of Flaky Tests?. In *Proceedings of the International Conference on Mining Software Repositories (MSR).* 492–502.

[35] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-Deterministic Specifications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST).* 80–90.

[36] D. Silva, L. Teixeira, and M. D'Amorim. 2020. Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME).* 301–311.

[37] V. Terragni, P. Salza, and F. Ferrucci. 2020. A Container-based Infrastructure for Fuzzy-driven Root Causing of Flaky Tests. In *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER).* 69–72.

[38] C. S. Timperley, L. Herckis, C. Le Goues, and M. Hilton. 2021. Understanding and Improving Artifact Sharing in Software Engineering Research. *Empirical Software Engineering* 26, 4 (2021), 1–41.

[39] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu. 2019. A Conceptual Replication of Continuous Integration Pain Points in the Context of Travis CI. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE).* 647–658.

[40] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).* 385–396.

[41] H. Zhu, L. Wei, M. Wen, Y. Liu, S. C. Cheung, Q. Sheng, and C. Zhou. 2020. MockSniffer: Characterizing and Recommending Mocking Decisions for Unit Tests. In *Proceedings of the International Conference on Automated Software Engineering (ASE).* 436–447.