

FLAKEFIGHTERS: A Pytest Plugin to Detect Flaky Test Failures

Michael Foster[✉]
m.foster@sheffield.ac.uk
University of Sheffield
Sheffield, UK

Owain Parry
oparry2@ed.ac.uk
University of Edinburgh
Edinburgh, UK

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

Gregory M. Kapfhammer
gkapfham@allegheny.edu
Allegheny College
Meadville, USA

Phil McMinn[✉]
p.mcminn@sheffield.ac.uk
University of Sheffield
Sheffield, UK

Abstract

Flaky tests — tests which non-deterministically pass and fail against the same version of code — pose a well-established painful issue for developers due to the unpredictability of failures. The current state of practice to detect and work around flaky tests is to simply rerun them. However, this can erode confidence in failing test outcomes and represents a significant waste of time and compute resources. A better solution is to detect flaky tests based on static data (collectable without running any tests) and/or execution data (collected by running the test suite) to determine if a test failure has occurred due to flaky behaviour or a genuine bug. Several such techniques have appeared in recent literature, but these exist in isolation as prototype research tools, and typically only support Java codebases. This paper presents FLAKEFIGHTERS, an extensible pytest extension that implements a “Swiss army knife” of flaky test classification tools to help developers classify the underlying cause of failing tests as genuine bugs or flakiness in Python codebases. We ran pytest with the FLAKEFIGHTERS extension on 23 failing workflows from the home-assistant/core repository and found that it was able to identify all of the failing flaky tests. A video demonstration of FLAKEFIGHTERS can be found at <https://youtu.be/3J4odtgzQX0>.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; *Software maintenance tools*.

Keywords

Software testing, Flaky tests, Pytest extension

ACM Reference Format:

Michael Foster, Owain Parry, Michael Hilton, Gregory M. Kapfhammer, and Phil McMinn. 2026. FLAKEFIGHTERS: A Pytest Plugin to Detect Flaky Test Failures. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*, October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3832783.3834641>

1 Introduction

Tests that non-deterministically pass and fail on the same version of a codebase are known as *flaky tests*, and occur frequently in many projects [19, 21]. For example, the open-source home-assistant project [1] enables modular integration with various home-automation products such as smart speakers, security, and temperature control. The core repository has almost 87,000 stars, 5,000 contributors, and 219,000 lines of Python code. Much of the functionality involves interaction with sensors over networks, leading to frequent flaky behaviour and a dedicated flapping-test tag for such issues.

Listing 1: Simplified example of a real flaky test taken from the home-assistant/core repository.

```
1  async def test_update_connection_error(  
2      home_assistant,  
3      mock_concord232,  
4      caplog,  
5  ) -> None:  
6      await async_setup_component(home_assistant,  
7          ALARM_DOMAIN, CONFIG)  
8      await home_assistant.async_block_till_done()  
9      mock_concord232.side_effect = ConnectionError(  
10         "Connection failed")  
11  
12      freezer.tick(10)  
13      async_fire_time_changed(home_assistant)  
14      await home_assistant.async_block_till_done()  
15  
16      assert "Unable to connect" in caplog.text
```

The code in Listing 1 shows a slightly simplified version of a real flaky test taken from the home-assistant/core repository. It is a test for the concord232 integration, which allows automated interaction with certain types of burglar alarm. The test sets up a mocked version of the component and triggers a “time changed” event, which is expected to elicit a (mocked) connection error. Yet, even though the test calls `async_block_till_done`, there is a race condition whereby the assertion is sometimes evaluated before the “Unable to connect” message is logged. This illustrates that flaky behaviour can still remain even when developers use best practices like mocking physical components and network connections.

For large codebases such as home-assistant/core, flaky tests can cause a real impediment to Continuous Integration (CI) by failing for commits that do not introduce a regression, or even change any relevant code. This erodes trust in tests, and can lead developers to ignore or even delete them altogether [15, 22]. The most



This work is licensed under a Creative Commons Attribution 4.0 International License. ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
<https://doi.org/10.1145/3832783.3834641>

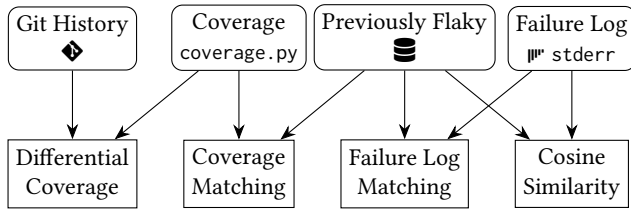


Figure 1: The input features taken by each flakefighter.

common workaround is to simply rerun the test: if it passes then it was flaky and the workflow can continue unimpeded. There are several pytest extensions that support this, for example `flaky` [5] and `rerunfailures` [8]. Related to this, `flakefinder` [7] provides the ability to *detect* flaky tests by repeatedly rerunning the suite.

However, simply rerunning a test provides no insight into *why* tests are flaky or how to repair them. In recent years, several more advanced flaky test detection techniques have appeared in the literature [11, 14, 17, 23] that can provide this insight. Unfortunately, these exist as standalone research tools rather than plugins for existing testing frameworks that developers can easily use.

This paper presents FLAKEFIGHTERS, a pytest extension that implements a “Swiss army knife” of flaky test detection tools to help developers classify test failures as being due to genuine bugs or flakiness, and to provide insights into the underlying causes of flakiness. Python developers can easily adopt FLAKEFIGHTERS to avoid the many time-consuming challenges that arise when extending existing open-source tools with methods from the research literature. To demonstrate how developers can use the presented tool, we executed it on a sample of flaky tests from the home-assistant/core GitHub repository [2]. We walk through the process in detail for a failing workflow and perform a preliminary experimental evaluation on 23 failing workflows. We found that our tool was able to correctly identify all the flaky tests in these workflows.

2 Flaky Test Classifiers

The FLAKEFIGHTERS plugin provides a production-ready unified implementation of several flaky test detection techniques from the literature. We refer to each individual technique implemented into the plugin as a “flakefighter”. As discussed in Section 1, the current state-of-practice for determining whether a test has failed due to flakiness or a genuine bug is to simply rerun it. Yet, there are several approaches in the literature that make this classification based on execution data, by examining different features of the test run. Figure 1 provides a visual summary of the inputs used by a flakefighter. This section outlines the techniques that we implemented in FLAKEFIGHTERS, and highlights their strengths and limitations.

2.1 Differential Coverage

DeFlaker, due to Bell et al. [14], classifies test failures as flaky or genuine based on *differential coverage*. The assumption is that if a test passed on a previous version of the codebase and failed on the latest version, but without executing any modified code, then the test is flaky. The main limitation is that only code files are considered; if a change has been made to a data or configuration file, genuine test failures may be falsely marked as flaky. Additionally,

the approach depends on the accuracy of code coverage measurement, which may be limited when using features such as reflection and annotations, which are often used in Python projects.

FLAKEFIGHTERS provides the first implementation of the DeFlaker algorithm for Python. DeFlaker is currently only available as a Maven build extension [4] for Java codebases tested with JUnit, and relies on various language-specific optimizations for code coverage measurement. While we do not claim to provide a direct translation of the Maven tool for Python, we have faithfully implemented the original algorithm from Bell et al. [14] (as have Magill et al. for Rust [20]), and included a translation of their flaky test example in our tool’s test suite. Additionally, the FLAKEFIGHTERS implementation of DeFlaker also classifies *passing* tests as flaky if they transition from failing to passing without any code changes.

2.2 Failure Log Matching and Cosine Similarity

Alshammari et al. [11] propose to classify flaky failures by looking for matches between the failure logs of newly-failing tests and the failure logs of known flaky tests. The intuition is that engineers familiar with a codebase may suspect that a new test failure is flaky if they have seen the same flaky failure “symptoms” before in the failure logs of other tests. Similar approaches have been implemented in the context of specific organizations [13].

Text-based matching searches for exact string matches between the log of a newly-failing test and logs of previously-known flaky tests, classifying a test as flaky if a match is found. A slightly more advanced approach involves treating failure logs as “documents” and measuring their similarity using Term Frequency-Inverse Document Frequency (TF-IDF), rather than looking for exact matches. A test is then classified as flaky if the cosine similarity is above a specified threshold. In FLAKEFIGHTERS, this defaults to 1 (i.e., an exact match), but is configurable by the developer using the plugin.

As with the DeFlaker tool described in Section 2.1, FLAKEFIGHTERS provides the first implementation of these techniques in a pytest plugin for use by the general Python community, with prior implementations only being available as stand-alone prototype research tools [11, 13]. The main limitation of the FLAKEFIGHTERS implementation is the availability of previous failure logs, which may not always be stored long term. As discussed in Section 3, a major advantage of FLAKEFIGHTERS in this regard is that it automatically records all runs in a database to provide a set of known flaky tests.

2.3 Coverage Matching

Inspired by prior work that uses coverage as a signal to detect flaky test failures [12, 14, 17], we also present a new flakefighter based on comparing code coverage profiles between tests. New test failures are classified as flaky if the set of covered lines is sufficiently similar to enough tests from previous executions that have already been classified as flaky. To the best of our knowledge, this specific utilization of coverage is a novel flakiness detection technique.

3 FLAKEFIGHTERS PyTest Plugin

Our intention for FLAKEFIGHTERS is to provide a comprehensive suite of flaky test detection tools (i.e., flakefighters) to help practitioners discover and debug flaky tests, and to establish a standard platform that researchers can extend with new detection methods.

We implemented FLAKEFIGHTERS as a pytest plugin that integrates seamlessly with standard CI/CD workflows. It can be used by simply installing it and running pytest with the `--flakefighters` command-line option to use the default configuration. Test failures that FLAKEFIGHTERS identifies as flaky will then appear in pytest’s output as **FLAKY** rather than **FAILED**, highlighting that the failure may not be genuine. FLAKEFIGHTERS also provides an option to suppress flaky failures, meaning that CI/CD workflows with only flaky failures will not fail at the test running stage.

FLAKEFIGHTERS is hosted on PyPI [6] and can be installed by running a command such as `pip install pytest-flakefighters` or `uv add pytest-flakefighters --group dev`. The source code of this paper’s plugin is also available on GitHub [3]. Further documentation and tutorial material can be found on ReadTheDocs [9].

3.1 Implementation of FLAKEFIGHTERS

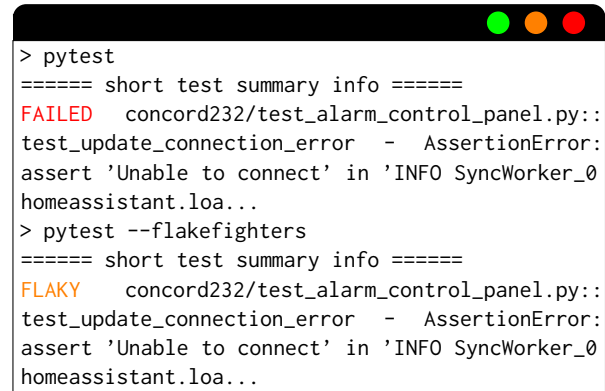
Software developers begin by specifying the configuration for the FLAKEFIGHTERS run. In particular, each flakefighter can be run either individually or in combination with other flakefighters, and in either “live” mode or “postprocessing” mode. Live flakefighters are run immediately after a test fails, which can be useful in determining whether to rerun a test, or terminate the workflow prematurely if a genuine failure has been observed, but may incur a slight penalty in terms of execution time. Flakefighters configured for postprocessing mode are run after the entire test suite has completed.

FLAKEFIGHTERS works by employing pytest hooks at four main points: initialization, test collection, test execution, and status reporting [10]. At the initialization phase, the pytest plugin is initialized by setting the project root directory (which defaults to the current working directory) and loading previous runs from the database (or initializing such a database if this is the first run). These previous runs are used by several of the flakefighters to make their classification, as previously discussed in Section 2.

During the test execution phase, FLAKEFIGHTERS measures the coverage of each test individually using `Coverage.py`. Any flakefighters configured to run in live mode will then classify failures as flaky or genuine. Users can optionally enable test reruns if certain conditions are met. Several rerun criteria are supported, including failures classified as flaky, previously flaky tests (regardless of test outcome), or simply rerunning all tests a certain number of times (useful for researchers looking to discover/verify flaky behaviour). The flakefighters configured to run in postprocessing mode trigger after all the tests have run. The classifications of postprocessing flakefighters only serve as a marker. FLAKEFIGHTERS currently only supports rerunning tests based on the results of live flakefighters.

During the reporting stage, pytest would typically report test outcomes as pass/fail. FLAKEFIGHTERS adds **FLAKY**, a third option for flaky failures. The user may configure the plugin to suppress flaky failures, meaning that they are not recorded in the number of failing tests reported to CI/CD pipelines, so if the only failing tests are flaky failures, this will not block the workflow. The plugin is also designed to integrate with existing pytest extensions to save flaky test outcomes to a JSON, HTML, and/or JUnitXML file.

Finally, the results of the run are saved to the database, unless the user has passed the `--no-save` option to stop this from happening. For each run, the outcome and flaky classification of each test is



```
> pytest
===== short test summary info =====
FAILED concord232/test_alarm_control_panel.py::
test_update_connection_error - AssertionError:
assert 'Unable to connect' in 'INFO SyncWorker_0
homeassistant.loa...
> pytest --flakefighters
===== short test summary info =====
FLAKY concord232/test_alarm_control_panel.py::
test_update_connection_error - AssertionError:
assert 'Unable to connect' in 'INFO SyncWorker_0
homeassistant.loa...
```

Figure 2: pytest output with and without FLAKEFIGHTERS.

saved, along with the captured stdout, stderr, and failure log information for use by future FLAKEFIGHTERS runs. To help manage storage space, users may also configure the maximum number of runs to keep, or the earliest date from which to keep runs.

3.2 Example Execution

To illustrate the utility of FLAKEFIGHTERS, consider again the example flaky test in Listing 1, which can non-deterministically pass or fail due to a race condition. When a developer makes a change to the repository, such as implementing a new feature, this test could fail due to flakiness, even if the developer has not changed the relevant `concord232` component. As illustrated in the output of the first command in Figure 2, the test is simply reported as **FAILED**. This gives the developer no indication that the test is flaky. Furthermore, if they *did* make changes to the `concord232` component that were *unrelated* to the test, they may spend time on unproductive activities like debugging either the test or the functions that it calls.

When pytest is run with our tool’s default configuration through `--flakefighters`, FLAKEFIGHTERS correctly identifies the flaky behaviour since no code covered by the test was modified by the developer. The failure is still reported, but as **FLAKY**, as illustrated in the output of the second command in Figure 2. This makes the developer aware of the flakiness, and suggests that the test may pass if retried. Furthermore, even if the developer modifies the relevant component in the future, the test can still be flagged as flaky by the flakefighters based on matching failure logs and test coverage.

The utility of this becomes even clearer when applied to CI pipelines. In the scenario above, if the developer submits a pull request (PR) and FLAKEFIGHTERS is not part of the workflow, the PR will be blocked until the testing action is repeated. This is not only frustrating for the developer, but wastes compute resources [18]. When used with the `--suppress-flaky-failures-exit-code` option, FLAKEFIGHTERS will allow the workflow to succeed when the only failing tests are flaky. This means that PRs are not hindered by flaky tests, saving both developer time and compute resources.

4 Demonstration of Utility and Quality

The example in Section 3.2 illustrates the utility of FLAKEFIGHTERS to developers. To demonstrate its reliability, we applied it to a

sample of failing CI workflows from the home-assistant/core repository discussed in Section 1. The repository for this Python project has over 40,000 CI runs in total, almost 25,000 of which are failing. The complex multitude of components designed to interact with a network of sensors means that these CI runs are plagued with flaky tests, making it a suitable case study for our tool.

We used the GitHub API to look for CI workflow runs associated with a pull request (PR) for which at least one test failed and then subsequently passed, submitting authenticated queries until we hit the rate limit (5000 per hour at the time of writing). This gave us a sample of 23 failing runs with a total of 30 failing tests and 26 distinct test cases that failed (due to several runs having multiple failing tests, and some tests failing in multiple runs). Each of these distinct failures may have been caused by either flakiness or a genuine defect in the program or test suite. We initially ran each failing test with FLAKEFIGHTERS on the source commit of the PR. That is, we ran our tool on the state of the repository before merging. To confirm flakiness, we repeated this up to 1,000 times, stopping early once a passing and failing outcome had been observed. Each of these runs was stored in the FLAKEFIGHTERS database.

We then ran FLAKEFIGHTERS on the target commit of the PR (i.e., the state of the repository after the proposed merge). Again, we repeated this up to 1,000 times, stopping early once a passing and failing outcome had been observed. We then recorded the classifications of the flakefighters detailed in Section 2 for the first observed failure of the target commit. The code and data for our experiment is available in our replication package at [16].

For both the 1,000 source runs and 1,000 target runs for the PR, the same 12 tests failed at least once: 6 of these were flaky, while the other 6 failed on every run, thus indicating that a genuine bug is likely. For the remaining 14 tests, we could not replicate the original failure, indicating that this may have been due to an error in the GitHub Actions workflow setup itself. Although we did not explicitly evaluate the influence of FLAKEFIGHTERS on the runtime of pytest, we did not observe a noticeable difference in performance during either its development or evaluation.

Table 1 shows the confusion matrix for each flakefighter detailed in Section 2. The Cosine Similarity flakefighter successfully identified all six flaky failures, with only two false positives. In this case, a false positive is a test being classified as flaky when all 1,000 of our runs failed, thereby providing no evidence of flaky behaviour. The Differential Coverage flakefighter was only able to identify two of the failures as flaky. The other failures were classified as genuine, highlighting its limitations in terms of changes to non-code files going undetected. The Failure Log Matching flakefighter did not identify any tests as flaky. The reasons for this are twofold: firstly, it depends on tests having been previously identified as flaky in a previous run. Secondly, the failure logs may not be exact matches between runs, making it hard to detect failures this way. The Coverage Matching flakefighter did not perform particularly well in this case study as it did not classify any of the flaky failures as such, and also produced two false positives. However, only one of these was the same as for Cosine Similarity flakefighter.

These results illustrate the power of using the different flakefighters *in combination*. Where certain flakefighters may fall short, others can make up the difference, leading to strong detection ability with a low rate of false positives. While the current implementation

Table 1: Flakefighter results for the 12 distinct failing tests. Flaky is the positive class. Coverage Matching is invalid when all runs have the exact same coverage profile.

Flakefighter	True Positives	False Positives	True Negatives	False Negatives	Invalid
Cosine Similarity	6	2	4	0	0
Differential Coverage	2	0	6	4	0
Failure Log Matching	0	0	6	6	0
Coverage Matching	0	2	2	5	3
<i>Combined</i>	6	3	3	0	0

of FLAKEFIGHTERS simply reports a test as flaky if any of the active flakefighters have classified it as such, a more intelligent combination of individual flakefighter classifications into a “suspiciousness score” would improve the overall performance of the tool.

5 Conclusion and Future Work

This paper presented FLAKEFIGHTERS, a suite of flaky test detection tools for pytest. This extension represents a significant step in flaky test detection for the Python community since advanced flaky test detection techniques from the literature have previously only been available as prototype tools for Java projects. We have demonstrated the utility of the tool with real flaky tests from the home-assistant/core repository. While we leave a more comprehensive evaluation for future work, this illustrates the power of using the flakefighters *in combination*. One promising area for future work would be to explore different ways to aggregate the individual flakefighter results into an overall “suspiciousness score” to prioritise the tests which are most likely to be flaky (e.g., flagged by multiple flakefighters) while reducing the number of false positives.

Data Availability Statement

All code and data for our demonstration in Section 4 is provided as part of our replication package [16]. The source code for our FLAKEFIGHTERS plugin is available at [3].

Acknowledgements

This work was supported by the EPSRC grant “Test FLARE”, project number EP/X024539/1.

References

- [1] 2026. GitHub: Home Assistant, <https://github.com/home-assistant>.
- [2] 2026. GitHub: home-assistant/core, <https://github.com/home-assistant/core>.
- [3] 2026. GitHub: test-flare/pytest-flakefighters, <https://github.com/test-flare/pytest-flakefighters>.
- [4] 2026. Maven Repository: DeFlaker, <https://mvnrepository.com/artifact/org.deflaker/deflaker-maven-extension>.
- [5] 2026. PyPI: flaky, <https://pypi.org/project/flaky/>.
- [6] 2026. PyPI: pytest-flakefighters, <https://pypi.org/project/pytest-flakefighters/>.
- [7] 2026. PyPI: pytest-flakefinder, <https://pypi.org/project/pytest-flakefinder/>.
- [8] 2026. PyPI: pytest-rerunfailures, <https://pypi.org/project/pytest-rerunfailures/>.
- [9] 2026. Pytest Flakefighters, <https://pytest-flakefighters.readthedocs.io/en/latest/>.
- [10] 2026. Writing Hook functions, Pytest How-to Guides, https://docs.pytest.org/en/stable/how-to/writing_hook_functions.html.
- [11] A. Alshammari, P. Ammann, M. Hilton, and J. Bell. 2024. 230,439 Test Failures Later: An Empirical Evaluation of Flaky Failure Classifiers. In *IEEE Conference on Software Testing, Verification and Validation (ICST 2024)*. <https://doi.org/10.1109/ICST60714.2024.00031>
- [12] 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 2021)*. <https://doi.org/10.1109/ICSE43902.2021.00140>
- [13] G. An, J. Yoon, T. Bach, J. Hong, and S. Yoo. 2024. Just-in-Time Flaky Test Detection via Abstracted Failure Symptom Matching. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME 2024)*. <https://doi.org/10.1109/ICSME58944.2024.00078>
 - [14] 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 2018)*. <https://doi.org/10.1145/3180155.3180164>
 - [15] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. 2019. Understanding flaky tests: the developer's perspective. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. <https://doi.org/10.1145/3338906.3338945>
 - [16] Micahel Foster. 2026. Flakefighters: A Pytest Plugin to Detect Flaky Test Failures. <https://doi.org/10.15131/shef.data.32243409> Replication package.
 - [17] M. Gruber and G. Fraser. 2023. Debugging Flaky Tests using Spectrum-based Fault Localization. In *IEEE/ACM International Conference on Automation of Software Test (AST 2023)*. <https://doi.org/10.1109/AST58925.2023.00017>
 - [18] F. Leinen, D. Elsner, A. Pretschner, A. Stahlbauer, M. Sailer, and E. Jurgens. 2024. Cost of Flaky Tests in Continuous Integration: An Industrial Case Study. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST 2024)*. <https://doi.org/10.1109/ICST60714.2024.00037>
 - [19] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. <https://doi.org/10.1145/2635868.2635920>
 - [20] B. Magill and P. McMin. 2025. deflake.rs: Detect Flaky Tests in Rust Projects using Execution Data. *The Journal of Open Source Software* (2025). <https://doi.org/10.21105/joss.08757>
 - [21] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMin. 2021. A Survey of Flaky Tests. *ACM Transactions on Software Engineering and Methodology* 31, 1 (2021). <https://doi.org/10.1145/3476105>
 - [22] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMin. 2022. Surveying the Developer Experience of Flaky Tests. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2022)*. <https://doi.org/10.1145/3510457.3513037>
 - [23] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMin. 2023. Empirically Evaluating Flaky Test Detection Techniques Combining Test Case Rerunning and Machine Learning Models. *Empirical Software Engineering* 28, 3 (2023). <https://doi.org/10.1007/s10664-023-10307-w>

Received 2026-05-12; accepted 2026-06-19