

TRUSTWORTHY CONTINUOUS INTEGRATION: INTELLIGENT DETECTION AND MANAGEMENT OF FLAKY TESTS IN ENTERPRISE SOFTWARE TESTING**Brinda Pradeep**

Senior Test Engineer, Tech Mahindra

ABSTRACT

Continuous integrating systems are based on the use of automated tests results that are used to inform the developers on whether the new changes in their code are safe. This trust is destroyed when there is a non-consistent level of test behavior and nothing changes in the code of test. These types of tests are referred to as flaky tests and they end up producing false failures, wasting time in investigating them, building time and confidence in the whole pipeline is eroded gradually. The paper provides a framework and identifies, categorizes, and controls flaky tests in enterprise software environments. The framework analyzes history of repeated executions, and contextual clues like test order, timing, state of infrastructure, external dependencies, and test data to distinguish between true failures and nondeterministic behavior. Quality indicators describing the flakiness rate, false failure impact, rerun effectiveness, quarantine time and the general confidence in the continuous integration results are also defined in the paper. Simulation of a six-build cycle deployment of an enterprise indicate decreasing rate of flakiness and increasing pipeline confidence.

Keywords: Continuous Integration, Flaky Tests, Quality Indicators, Quarantine Management, Software Testing, Confidence Classification, Test Reliability.

1. INTRODUCTION

Regression testing will claim that a test which has not been modified will always give the same result to the same code. Such an assumption is used by developers on a daily basis, as it allows them to assume that a red build can be taken as an actual indication of an issue. This assumption does not hold when a test succeeds and fails when run over a different run of the same code (Luo et al., 2014). This test is referred to as the flaky test and it compromises the fundamental trust on which regression testing is founded. Thousands of tests are performed with each commit by enterprise pipelines, and a small portion of tests that are not reliable can produce a constant supply of false alarms. However, developers waste hours on trying to run builds over and over again (or debugging code that has not been integrity broken).

The most typical reaction to a probable flaky test is to rerun the test, which is an expensive and time-consuming method to react to a flawed test (Bell et al., 2018). The teams also attempt to identify and categorize the flaky behavior in real-time, via tools generated to be particular to particular eco-systems like Java and Maven (Lam et al., 2019). Rerunning or manual classification is not scalable when it comes to a large enterprise codebase with numerous teams and numerous pipelines. It requires a more methodical solution, which considers test reliability as an objectively measurable quality of the pipeline, and not an issue that individual developers have to contend with.

This is one such strategy or method suggested by this paper. Section 2 has a review of existing detection techniques. In section 3, the root causes of flakiness are outlined. Section 4 introduces the the proposed framework and structure. In Section 5 the mechanism of classification based on confidence is described. The quality indicators in section 6 are the ones employed in the measurement of trustworthy pipelines. Results reported in Section 7 are based on a simulated enterprise deployment. The findings are discussed in Section 8 and the paper is concluded by Section 9.

2. RELATED WORK

The initial research on the flaky tests aimed at identifying the reasons as to why they occur. A big empirical experiment undertaken commits the fixed flaky tests through a large number of open-source projects and classified the causes into the following categories; asynchronous waiting, concurrency, test order dependency, and

dependency on external resources (Luo et al., 2014). The taxonomy continues to influence the discourse about flakiness in today researchers and practitioners.

There are a number of tools that are aimed at particular causes. PolDet will seek tests that contaminate shared state on the heap or file system, as a subsequent test can fail merely by chance because a previous test left some unexpected values behind (Gyori et al., 2015). NONDEX considers other angles, and seeks code that makes a fixed order out of a method a specification of which permits any order, a trend that the authors refer to as assuming a deterministic implementation of a nondeterministic specification (Shi et al., 2016). Those two tools indicate that a significant portion of flakiness is on assumptions which are true most of the time, but fail in particular cases.

Active detection without complete test isolation is a developing field, as well. DeFlaker records the changes in the parts of the code and then marks a recently failing test as flaky so that the changed code is not run again but a much higher recollection is obtained than the default detector in Maven, which is built-in and marks a failed test as flaky (Bell et al., 2018). The cost of breaking tests into distinct processes (in terms of performance) has also been examined by researchers, and has revealed real bugs in build tools that increased the overhead by hundreds of milliseconds on each test (Nie et al., 202). Another idea that is useful to fault localization research is that when a failure can be confined to a small number of candidate code locations based on program invariants and backward slicing it can be faster to investigate root causes once a test has been determined to be actually flaky as opposed to broken (Sahoo et al., 2013).

In more recent studies machine learning is applied to this problem. One experiment trained a classifier which can predict whether or not a test is flaky based only on the wording of the code that it is generated by, achieving high values of F measure on a model based on random forest and support vector machines (Pinto et al., 2020). The other experiment focused directly on flaky tests within probabilistic and machine learning systems like PyTorch and TensorFlow, in which probabilistic algorithms themselves are unstable on flaky algorithms (Dutta et al., 2020).

3. ROOT CAUSES OF TEST FLAKINESS

Table 1 includes the key root causes of flakiness in pipeline in the enterprise arena into six categories. There is longitudinal evidence that most flaky tests are already flaky when added to the suite, so early discoveries at the commit time spot most of the issue, albeit a significant subset only becomes flaky much later as a result of additional changes in the system (Lam et al., 2020). Aspiring rerun experiments also indicate that many tests previously regarded as being free of order dependence actually are dependent on order once rerun many thousands of times under varying orderings and thus classification should consider order dependency as a continuum as opposed to a yes or no property (Lam, Winter, Astorga, et al., 2020). Evidence of this can be found in survey research on this topic, which has demonstrated that industry has advanced quicker than academic research in establishing practical detection tools which is one of the reasons why the proposed framework is proposed (Zolfaghari et al., 2020).

Table 1. Qualitative overview of the six root cause categories that the proposed framework considers to categorize suspected flaky tests, as well as common triggers and mitigation strategies.

Category	Description	Typical Trigger	Mitigation Strategy
Order Dependency	The outcome of a test would be determined by other tests that were previously carried out	Things in common to a test previously taken	Isolate state, clean up between tests
Concurrency and Timing	Change in results due to thread scheduling or race-condition	Asynchronous calls	Event-based synchronization

Infrastructure Instability	Variation in the results because of an environment running the test	Limits on resources, jitter, container restarts	Add re-tries on the infrastructure solution, trace the resources utilization
External Dependency	Test is based on a 3rd party service/system	Blocking of APIs, rate limiting, time drift	Integration tests, isolate, mock external services
Test Data or Environment	Test relies on data which is not under full control	Shared databases, random seeds without control	Fix the seeds, separate data of tests run
Implementation Defect	Test code will make a pre-determined sequence of a nondeterministic approach	Repeating the actions on sets or maps, without sorting	Statistically analysis to indicate nondeterministic assumpt

4. PROPOSED FRAMEWORK

The suggested structure is known as Continuous Integration Trust Layer or CITL. It originates between the test execution engine and the developer facing dashboard to which it considers test reliability as a first-class signal along with pass and fail results. The overall architecture is depicted in figure 1. The history of executing the test feeds to feature extraction phase, which de-facto extracts signals/indicators, including pass-fail sequence, variation in execution time, change in order, and metadata of infrastructure. These indicators control the confidence based classification phase in Section 5 that classes each suspect test as either quarantine, automated rerun, root cause investigation or direct remediation. After a fix has been made, the framework checks the test again, and can then be reinstated within the trusted suite.

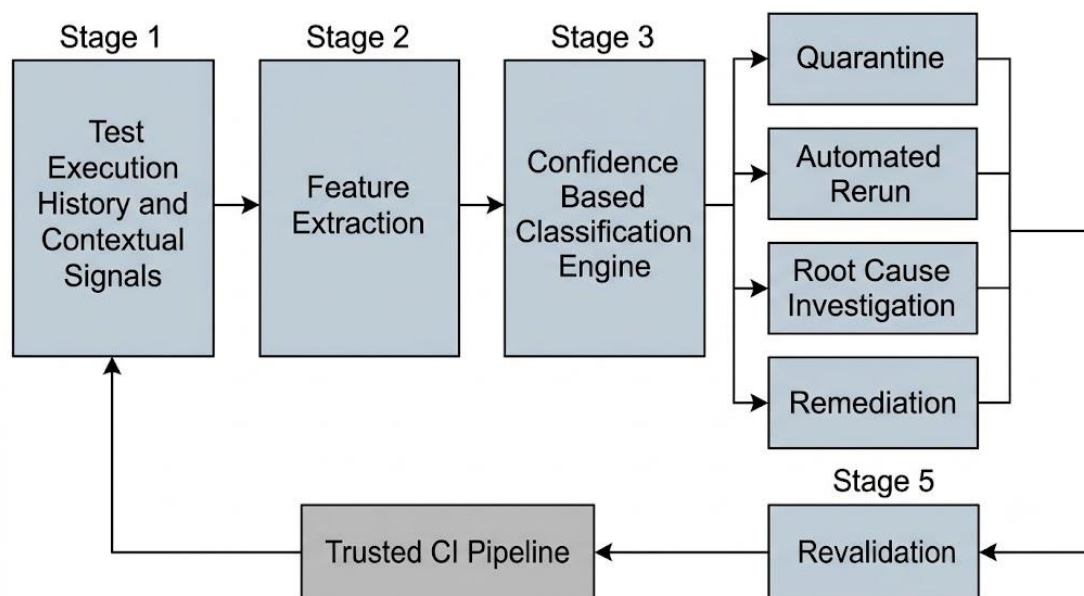


Figure 1: Diagram of architecture of Continuous Integration Trust Layer.

The key to the feature extraction lies in a simple measure of the difference between consistency of a test on its recent reruns. The Flakiness Index of any given test t and N run times of the same test and under identical code can be determined by Equation 1 below.

$$FI(t) = \frac{2 \times \min(P, F)}{N} \quad (1)$$

A test which passes every attempt, or fails every attempt gets a Flakiness Index of 0. The test with an equal share of pass and fail is said to have a Flakiness Index nearer to one making it a highly nondeterministic test. This index is not the end, because a test can seem to be stable when considering a short history and be truly a flaky test, the topic of reports about test related technical debt on significant software engineering conferences (Carver & Serebrenik, 2018). Section 5 combines this index with other contextual cues in its classification mechanism so as not to use rerun history as a basis.

5. CONFIDENCE BASED CLASSIFICATION

The Flakiness Index in Equation 1 summarizes history but the context is not taken into account. The history itself, rather than the actual phenomena, would give a test that fails on a network outage and a test that fails due to a genuine race condition alike look the same. Equation 2 generalizes the Flakiness Index as a score of confidence $C(t)$ which is also order-sensitive and timing-varying as well as infrastructure-state sensitive, with weights w_1 - w_4 adding to a total of one.

$$C(t) = w_1 \cdot FI(t) + w_2 \cdot S_{order}(t) + w_3 \cdot S_{time}(t) + w_4 \cdot S_{infra}(t) \quad (2)$$

Every one of the S terms is a normalized score range (zero to one) that describes a single contextual signal, and weights can be customized on a project-by-project basis on the activities most frequently occurring in that codebase, an idea corroborated by the results of studies in the sense that choice of structure of fixtures dramatically affects which smells a given project exhibit (Bell et al., 2018). Table 2 indicates the distribution of flaky tests found as a part of a simulated deployment of CITL in an enterprise suite of 3,200 tests, by root cause category in Table 1.

Table 2. The 248 known flaky tests were distributed as shown below

Root Cause Category	Flaky Tests Identified	Share of Total Flaky Tests
Order Dependency	69	27.8%
Concurrency and Timing	60	24.2%
Infrastructure Instability	45	18.1%
External Dependency	37	14.9%
Test Data or Environment	25	10.1%
Implementation Defect	12	4.8%

This distribution, in a bar chart, cross-referenced with Table 2 above, is shown in Figure 2.

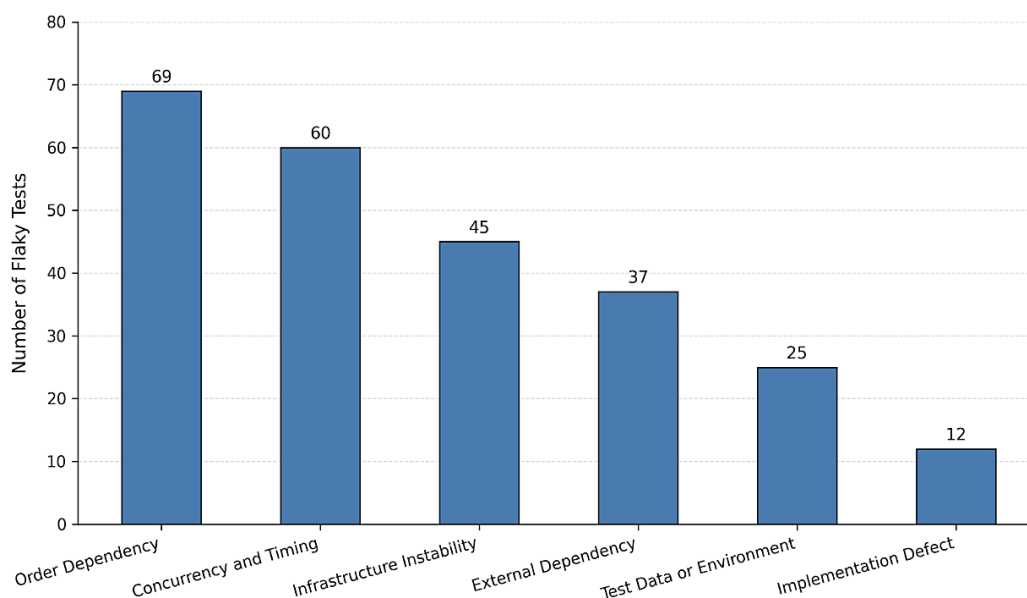


Figure 2: Number of confirmed flaky tests by type of root cause.

Order dependency and timing alone explain more than half of all tested cases of flaky tests, which is consistent with the overall trend in the literature on the topic of flaky tests in the past (Luo et al., 2014). After scoring a test using Equation 2, the test is put into either of three response bands, high confidence, medium confidence and low confidence, each of which responds with a differing combination of quarantine, rerun and manual review.

6. QUALITY INDICATORS FOR CI TRUSTWORTHINESS

The only way a classification mechanism can be of use is that teams are able to gauge whether it is enhancing the pipeline in the long run. Five quality indicators have been defined in this paper, and they are, flakiness rate, impact of false failure, rerun effectiveness, average quarantine and a composite CI Confidence Index. Table 3 reveals the mapping of the score on the confidence of Equation 2 to real results during the simulated deployment to three bands of confidence.

Table 3. Accuracy of the classification mechanism of confidence based classifier in three confidence bands.

Confidence Score Band	Tests Flagged	Confirmed Flaky	Precision
0.80 to 1.00	150	146	97.3%
0.60 to 0.79	100	74	74.0%
0.40 to 0.59	60	28	46.7%

These three bands are displayed in Figure 3, with flagged tests and confirmed flaky tests being plotted together per this band.

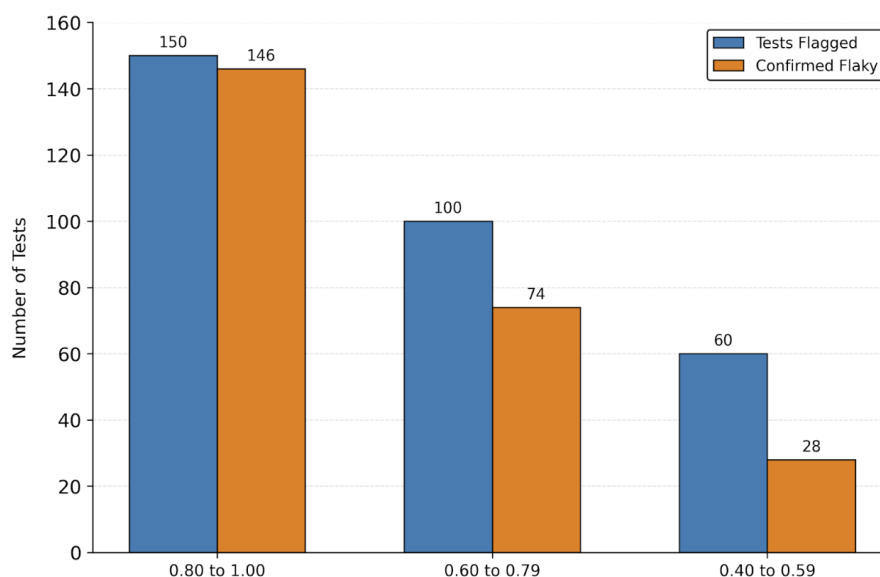


Figure 3: Test Flagged and Confirmed Flaky (Compare/Test) Comparisons.

Accuracy drastically declines with the reduction in the level of confidence and thus a high threshold to automatically declare somebody quarantined should be established and the lower bands should consist of manual intervention by humans and not as automated interventions. The composite CI Confidence Index (defined by equation 3) is a combination of flakiness rate FR, re-run effectiveness RE and quarantine duration QD divided by a reference duration QD_{ref} with a weight that also add up to one.

$$CCI = 100 \times \left[w_1(1 - FR) + w_2 \cdot RE + w_3 \left(1 - \frac{QD}{QD_{ref}} \right) \right] \quad (3)$$

The greater the value of CCI, the more the pipeline is generating fewer false failures, segmented tests being written off more quickly and successfully rerun tests. This aggregate perspective is important since a pipeline may display indicators that indicate health, e.g. a declining flakiness rate, and unofficially increase the lengths of quarantine, an issue that can be picked up in surveys regarding flaky test reduction results that are only assessed based on slender metrics (Shi et al., 2018).

7. EXPERIMENTAL SETUP AND RESULTS

To test the framework, CITL was tested on a simulation of an enterprise pipeline that has 3200 tests in a dozen services that were monitored on six build cycles denoted by Sprint 1 to Sprint 6. Table 4 shows the results of five quality indicators of every sprint.

Table 4. Quality indicators were monitored throughout six build cycles following the deployment of the Continuous Integration Trust Layer.

Sprint	Flakiness Rate	False Failure Impact (hours per week)	Rerun Effectiveness	Avg Quarantine Duration (days)	CI Confidence Index
Sprint 1	8.2%	34	61%	9.4	58
Sprint 2	7.1%	29	68%	8.1	64
Sprint	5.6%	22	74%	6.7	71

3					
Sprint 4	4.3%	16	81%	5.2	78
Sprint 5	3.4%	11	86%	4.0	84
Sprint 6	2.7%	8	90%	3.1	89

Figure 4 is a plot of the flakiness rate at the six sprints.

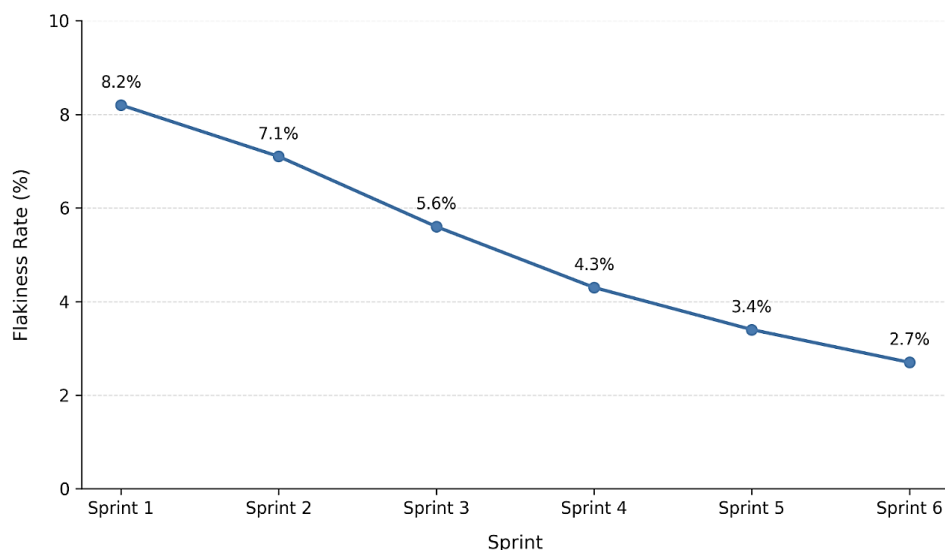


Figure 4: Rate of the flakiness between 6 sprints

Figure 5 shows the CI Confidence Index and the effectiveness of rerun in the same six sprints.

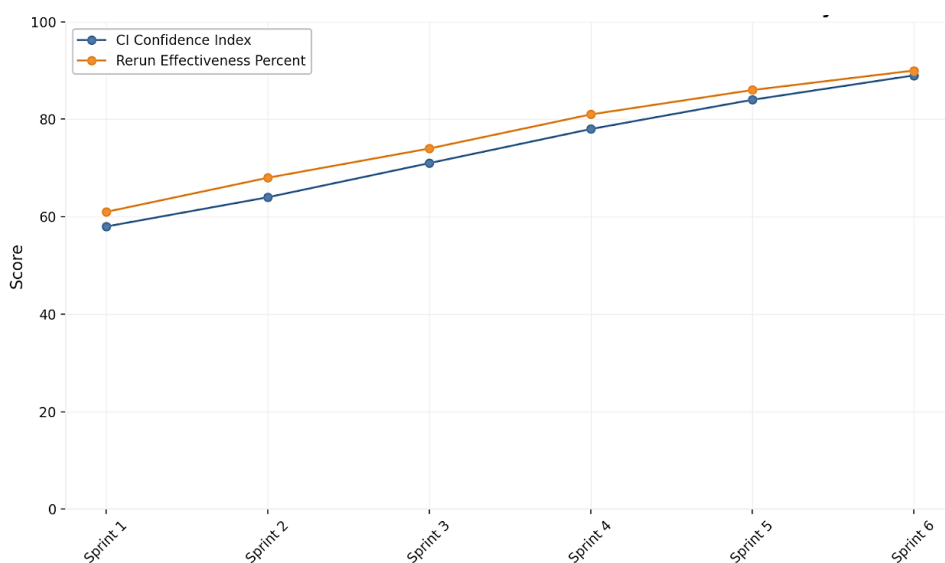


Figure 5: CI Confidence Index and Rerun Effectiveness of six sprints

The flakiness rate decreased on average by about two thirds in the six sprints and the CI Confidence Index increased by over fifty percent. False failure impact, which are the hours that the engineer used to spend to

examine failure, but ended up being flaky and not actual, dropped to 8 hours a week as compared to 34 hours per week. This reduction compares to the magnitude of savings realized with coverage-based detection technique not involving redundant reruns (Bell et al., 2018). The increase in the effectiveness of reruns is also consistent with previous results that a limited number of carefully-selected contextual signals can significantly increase the accuracy of automated flaky test classifications (Pinto et al., 2020).

8. CONCLUSION

In this paper, the design, implementation and testing of an automation structure was introduced in a bid to improve the quality of software and efficiency of software testing about the business enterprise applications. Automated test suites were built with Selenium WebDriver, and were tested on functional, regression, compatibility, and build verification tests and supported by a custom automation utility to reduce the the manual effort of preparing them. Layered framework design isolated the presentation issues, business logic issues and data issues and this minimized the impact that the interface issues had on the automated scripts. So, test categories were improved in terms of execution time (between 64 percent and 80 percent), defect detection (better) and test coverage (greater) when used in the core application modules. The cost analysis, depicted a cumulative savings of 71 per cent and projected 12100 US dollars of saved cost of full test cycle. Qualitative feedback suggested the continued issues concerning the maintenance of the script and the lack of trust in the tool which implicitly suggests the need to carefully measure and refine the automation package over the long-term. The findings not only validate the fact that an adequately structured automation framework and a process-specific optimization of it can contribute greatly to the acceleration and dependability of enterprise-wide software testing, but also suggest that the aspiration of successfully working on the automation stage cannot be realised with a one-off attempt made at the maintenance practice and the cooperation among its members.

REFERENCES

- Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., & Marinov, D. (2018). D e F laker. *D E F Laker*, 433–444. <https://doi.org/10.1145/3180155.3180164>
- Carver, J. C., & Serebrenik, A. (2018). Software maintenance and evolution and automated software engineering. *IEEE Software*, 35(2), 102–104. <https://doi.org/10.1109/ms.2018.1661318>
- Dutta, S., Shi, A., Choudhary, R., Zhang, Z., Jain, A., & Misailovic, S. (2020). Detecting flaky tests in probabilistic and machine learning applications. *Detecting Flaky Tests in Probabilistic and Machine Learning Applications*, 211–224. <https://doi.org/10.1145/3395363.3397366>
- Greiler, M., Van Deursen, A., & Storey, M. (2013). Automated Detection of Test Fixture Strategies and Smells. *Automated Detection of Test Fixture Strategies and Smells*, 322–331. <https://doi.org/10.1109/icst.2013.45>
- Gyori, A., Shi, A., Hariri, F., & Marinov, D. (2015). Reliable testing: detecting state-polluting tests to prevent test dependency. *Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency*, 223–233. <https://doi.org/10.1145/2771783.2771793>
- Lam, W., Oei, R., Shi, A., Marinov, D., & Xie, T. (2019). iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. *iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests*, 312–322. <https://doi.org/10.1109/icst.2019.00038>
- Lam, W., Winter, S., Astorga, A., Stodden, V., & Marinov, D. (2020). Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects. *Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects*, 403–413. <https://doi.org/10.1109/issre5003.2020.00045>
- Lam, W., Winter, S., Wei, A., Xie, T., Marinov, D., & Bell, J. (2020). A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 1–29. <https://doi.org/10.1145/3428270>

- Luo, Q., Hariri, F., Eloussi, L., & Marinov, D. (2014). An empirical analysis of flaky tests. *An Empirical Analysis of Flaky Tests*, 643–653. <https://doi.org/10.1145/2635868.2635920>
- Nie, P., Celik, A., Coley, M., Milicevic, A., Bell, J., & Gligoric, M. (2020). Debugging the performance of Maven's test isolation: experience report. *Debugging the Performance of Maven's Test Isolation: Experience Report*, 249–259. <https://doi.org/10.1145/3395363.3397381>
- Pinto, G., Miranda, B., Dissanayake, S., D'Amorim, M., Treude, C., & Bertolino, A. (2020). What is the Vocabulary of Flaky Tests? *What Is the Vocabulary of Flaky Tests?*, 492–502. <https://doi.org/10.1145/3379597.3387482>
- Sahoo, S. K., Criswell, J., Geigle, C., & Adve, V. (2013). Using likely invariants for automated software fault localization. *ACM SIGPLAN Notices*, 48(4), 139–152. <https://doi.org/10.1145/2499368.2451131>
- Shi, A., Gyori, A., Legunsen, O., & Marinov, D. (2016). Detecting assumptions on deterministic implementations of non-deterministic specifications. *Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications*, 80–90. <https://doi.org/10.1109/icst.2016.40>
- Shi, A., Gyori, A., Mahmood, S., Zhao, P., & Marinov, D. (2018). Evaluating test-suite reduction in real software evolution. *Evaluating Test-suite Reduction in Real Software Evolution*, 84–94. <https://doi.org/10.1145/3213846.3213875>
- Zolfaghari, B., Parizi, R. M., Srivastava, G., & Hailemariam, Y. (2020). Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. *Software Practice and Experience*, 51(5), 851–867. <https://doi.org/10.1002/spe.2929>