American Impact Review

Original Research

Flaky Tests as a Diagnostic Tool: A Conceptual Model for Early Detection of Systemic Problems in Infrastructure and Test Quality

Svetlana Ivanova*

Independent Researcher

* Corresponding author: ivsvetlana116@gmail.com

OPEN ACCESS

Citation:
Svetlana Ivanova (2026). Flaky Tests as a Diagnostic Tool: A Conceptual Model for Early Detection of Systemic Problems in Infrastructure and Test Quality. *Am. Impact Rev.*
10.66308/air.e2026059

Received: June 11, 2026
Accepted: July 7, 2026
Published: July 11, 2026

DOI:
10.66308/air.e2026059

ISSN: 3071-124X

Abstract

Background: Automated tests with unstable behavior are usually treated as a defect of test automation because they cause false positives, slow down releases, and reduce trust in the CI/CD pipeline. However, in complex distributed systems, flaky tests may also indicate hidden problems in the application, infrastructure, test data, or engineering process.

Methods: This article proposes a conceptual and applied model based on published research, CI/CD practices, and engineering experience in investigating flaky tests. The model treats flaky tests not only as operational noise, but also as diagnostic signals that can reveal systemic risks at an early stage.

Results: The proposed model distinguishes four diagnostic levels: test code, application, environment, and process. For each level, the article defines typical symptoms, probable sources of instability, relevant metrics, and initial analysis actions. It also introduces a prioritization matrix, a set of early detection indicators, and a flaky-signal handling cycle focused on identifying root causes rather than simply rerunning failed tests.

Conclusion: Flaky tests should not always be viewed only as automation defects. When analyzed systematically, they can become a practical tool for improving software reliability, increasing the diagnostic value of CI/CD data, and detecting risks before they lead to release delays, regression failures, or user-facing incidents.

Keywords: flaky test, CI/CD, test quality, test infrastructure, test automation, diagnosis of systemic problems

Introduction

Automated testing has long been one of the key mechanisms for managing software quality. As development cycles become shorter, teams increasingly depend on the results of the test pipeline: it is the pipeline that helps quickly determine whether the next code change is safe or contains a defect. Ideally, a test failure points to a problem in the product, while a successful run confirms that the changes are ready to move forward. In practice, this logic is disrupted by flaky tests.

A flaky test can produce different results during repeated runs on the same version of the code and with the same input data. The causes may vary: synchronization errors, dependence on execution order, unstable test data, external services, network delays, database behavior, or overloaded CI agents. The consequences are false positives, additional reruns, test quarantine, and a gradual decline in trust in automated verification results. Over time, the team begins to perceive red builds as routine noise rather than as a signal of a possible problem.

In practice, work with flaky tests most often comes down to eliminating symptoms: rewriting waits, avoiding

fixed delays, stabilizing locators, using mocks, or rerunning tests. These measures help reduce the number of false failures, but they rarely answer the more important question: why instability arises in this particular part of the system and why it recurs in specific environments, time intervals, or data sets.

A flaky test does not always indicate its own defect. In many cases, it points to an area of nondeterministic behavior that remains outside the team's control. The source of such instability may be in the test itself, the application, the infrastructure, the test data, or engineering processes. For this reason, flaky tests should be considered not only as an automation defect but also as a potential source of information about the state of the entire system.

Research Objective

The objective of this study is to propose a conceptual model in which flaky tests are considered not only as a source of noise but also as an early diagnostic signal. The model does not assume that every flaky test indicates a serious product problem. Its purpose is to help distinguish test-code defects, infrastructure instability, and signs of systemic degradation. This approach makes it possible to identify potential problems before they lead to release delays, mass regression-test failures, or user incidents.

The study considers the following questions:

- How can flaky tests be classified so that the classification helps identify root causes rather than merely record problematic scenarios?
- Which metrics make it possible to distinguish local test instability from systemic problems in the application or infrastructure?
- How can flaky-test analysis be integrated into quality management processes without significantly increasing operational costs?

Methodology

The article is conceptual and applied in nature. It is based on an analytical synthesis of published studies and engineering practices related to the analysis of flaky tests.

The methodology includes three interconnected stages.

1. Analysis of empirical studies of flaky tests. Despite differences between platforms and technologies, most studies identify similar causes of instability: asynchrony, dependence on execution order, lack of isolation, external services, network delays, work with time, and random data.
2. Generalization of CI/CD operation practices. Within the proposed approach, a flaky test is considered not only as a defect in a test scenario but also as an observability event. For such an event, the execution context, frequency of occurrence, links to other failures, analysis cost, and possible dependence on product or infrastructure changes are significant.
3. Construction of a diagnostic model. It does not depend on a specific programming language, automation tool, or application architecture and can be applied to all types of tests.

The proposed model is not based on the results of a single controlled experiment. Its contribution lies in forming a unified diagnostic framework that combines the results of published studies and practical experience

in operating automated testing. Empirical validation of the model requires an analysis of historical CI/CD data, including run results, environment characteristics, code changes, and information about the elimination of instability causes.

The Concept of a Diagnostic Signal

Traditionally, the result of a test run is interpreted in binary terms: the test either passes or fails. Flaky tests disrupt this model. A rerun can lead to the opposite result without any changes in code or configuration, so an individual failure can no longer automatically be considered proof of a defect.

However, this does not mean that such a failure has no value. On the contrary, it indicates the emergence of nondeterministic behavior in the chain: application - test - data - environment - process. The task of analysis is not only to restore a successful run but also to determine the source of this uncertainty.

The diagnostic potential of a flaky test is determined by several characteristics:

- symptom - error type, timeout, state loss, violation of the expected result, or infrastructure failure;
- context - commit, branch, CI agent, environment, container version, browser, database, run time, and load level;
- repeatability - frequency of occurrence, probability of recovery after a rerun, and distribution across run series;
- correlation - connection with other failing tests, infrastructure changes, service updates, or data migrations;
- cost - time spent on analysis, impact on the release process, and risk of missing a real defect.

Four-Level Diagnostic Model

The proposed model identifies four levels at which test instability can arise: test code, application, environment, and process. This division does not imply a rigid classification. The same symptom may have several causes at the same time. For example, a UI-test timeout may result from a poorly chosen wait, API degradation, CI-agent overload, or the absence of agreed response-time requirements between teams. Nevertheless, division by levels makes it possible to localize the source of the problem more quickly, identify the party responsible for eliminating it, and avoid a situation in which symptoms are fixed without searching for the root cause.

Table 1. Diagnostic classification of sources of flaky tests.

Level	Typical symptom	Probable source	Primary metric	Initial action
Test code	Failure of one scenario without connection to the environment	Brittle locator, sleep, shared state, execution order	Flaky rate by test	Fix test design and isolation
Application	Instability around one business process	Race condition, eventual consistency, implicit API contract	Share of failures with product defect	Check the behavior contract and state readiness
Environment	Simultaneous failures of different scenarios	CI agent, container, network, browser, resource limits	Environment skew and cluster failure index	Diagnose the test platform as a reliability incident
Process	Long quarantine and recurring ignored failures	No owner, SLA, triage, or return policy	Quarantine age and time-to-triage	Assign an owner, deadline, and escalation rule

4.1. Test-Code Level

This level covers cases in which the source of instability is in the test itself. The most common causes are well known: using fixed delays instead of waiting for state, unstable locators, dependence on scenario execution order, reuse of shared state, lack of test-data cleanup, work with random values, and dependence on locale, time zone, or implementation details of the user interface.

When analyzing such failures, it is important to answer not so much the question ”why did the test fail?” as the question ”to what extent does the test itself control the conditions of its execution?” For this purpose, it is usually sufficient to check:

- whether the scenario can run independently of the others;
- whether time, randomness, and external dependencies are controlled;
- whether the test verifies a business invariant rather than implementation details;
- whether brittle expectations are absent.

The most informative metrics remain the individual flaky rate, the average number of reruns before a successful result, the age of the flaky test, and the share of cases in which the problem was resolved by changing only the test code.

4.2. Application Level

If instability is related to application behavior, the flaky test ceases to be solely an automation problem. It becomes an indicator of nondeterminism in the system itself.

Possible causes include races between asynchronous handlers, eventual consistency, the absence of an explicit signal that an operation has completed, violation of idempotency, undefined data order, unstable API contracts, or hidden dependencies on cache and background processes.

Such cases are especially interesting because automated regression can detect a problem much earlier than it becomes visible to users. Thousands of daily test runs effectively turn CI into an early warning system.

When investigating such failures, it is useful to check:

- whether the application has a reliable signal that an asynchronous operation has completed;
- whether the data order used by the test is guaranteed;
- whether the test expectation matches the official product contract;
- whether a race arises between the UI, API, message queues, and database.

If the analysis confirms that instability is linked to application behavior, such a case should already be treated as a product reliability problem. In these situations, the fix is usually related not to the test but to the application itself: clarifying contracts, improving observability, stabilizing the API, or changing state-processing logic.

4.3. Environment and Infrastructure Level

At this level, instability is caused by characteristics of the test infrastructure: CI/CD, containers, browsers, databases, message queues, network delays, or resource limits.

Practice shows that infrastructure causes rarely manifest as isolated failures. Much more often, they cause a series of failures in several independent scenarios at once. This is why it is important to analyze not an individual flaky test but the overall picture.

During investigation, it makes sense to pay attention to several signs:

- whether the failures coincide in time;
- whether they are associated with a specific agent or test environment;
- whether the version of the container, browser, or base image changed.

Useful metrics include flaky rate by environment, distribution of errors across CI agents, test execution time, and the number of scenarios that failed simultaneously.

If several independent tests begin to fail synchronously, it is reasonable to treat the problem as a test-platform incident rather than as a set of separate defects.

4.4. Process Level

$$FR_t = \frac{F_t}{R_t},$$

$$FR_c = \frac{\sum_{t \in C} F_t}{\sum_{t \in C} R_t},$$

$$CFI_w = \frac{N_{\text{failed\_unique}}(w)}{N_{\text{executed\_unique}}(w)},$$

$$RRR = \frac{F_{\text{recovered}}}{F_{\text{rerun}}},$$

$$TTT = T_{\text{classified}} - T_{\text{first\_failure}}$$

$$QA_t = T_{\text{now}} - T_{\text{quarantined}}$$

$$ES = \max(FR_e) - \min(FR_e)\,,$$

$$SNR = \frac{F_{\text{actionable}}}{F_{\text{total}}}\,,$$

The final level of the model is related not to technical characteristics of the system but to the organization of the team's work. In practice, it is often the process that determines whether a flaky test becomes a one-off incident or a chronic problem.

The causes here are well known: lack of test owners, indefinite quarantine, absence of an SLA for fixes, manual management of test data, formal processing of regression results, and absence of stability requirements for new automated tests.

Such problems usually develop gradually. While the number of flaky tests is small, the team continues to trust CI results. But if flaky failures become routine, the attitude changes: a red build is no longer perceived as a signal requiring attention. As a result, the effectiveness of the entire quality-control process declines.

During analysis, it is worth paying attention to several questions.

- Does every flaky test have someone responsible for maintaining it?
- Is the time a test may remain in quarantine limited?
- Are the causes of instability analyzed, or is only the number of failures recorded?
- Are the results of such analysis used in technical-debt planning and retrospectives?
- Are there situations in which a release is blocked even after successful test reruns?

The most indicative metrics remain the lifetime of a flaky test, the share of tests in quarantine, time until an owner is assigned, and repeated failures after fixes have been introduced.

Prioritization Matrix

Not every flaky test requires the same response. The significance of a specific case is determined not only by how often it appears but also by how useful the information it provides about the state of the system is.

In the proposed model, each flaky signal is evaluated according to two criteria:

- diagnostic value - the probability that instability reflects a systemic problem;
- operational cost - the impact of the failure on the team's work and the delivery process.

The combination of these characteristics makes it possible to distinguish four priority classes.

Table 2. Flaky signal prioritization matrix.

Diagnostic value	Operational cost	Priority	Interpretation	Recommended response
High	High	P0/P1	Possible systemic degradation or critical product risk	Immediate cross-functional triage
High	Low	P2	Early signal that is not yet blocking delivery	Trend monitoring and planned investigation
Low	High	P2/P3	Noisy test with low protective value	Rewrite, simplify, or remove the scenario
Low	Low	P4	Accumulated test debt	Fix in batches as part of hygiene work

The matrix helps avoid two common mistakes: ignoring all flaky tests as inevitable noise and, conversely, conducting an equally deep investigation of every isolated failure regardless of its significance.

Early Detection Indicators

For flaky tests to truly become an early diagnostic tool, it is not enough to record individual failures. It is much more important to analyze their behavior over time, identifying stable patterns and relationships between events. For this purpose, a set of complementary indicators is proposed.

$$FR_t = \frac{F_t}{R_t},$$

where FR_t is the flaky rate of test t,

F_t is the number of flaky failures of test t during the observation period,

R_t is the total number of runs of test t during the same period.

Flaky rate by test shows how often a specific test changes its result without changes in code or execution conditions. Growth in the indicator usually points to degradation of the test itself or of the component associated with it.

$$FR_c = \frac{\sum_{t \in C} F_t}{\sum_{t \in C} R_t},$$

where C is the set of tests associated with a component or business process.

Flaky rate by component makes it possible to assess the stability not of an individual test but of a business component or service. Growth in the indicator more often points to application problems than to automation defects.

$$CFI_w = \frac{N_{\text{failed\_unique}}(w)}{N_{\text{executed\_unique}}(w)},$$

where w is a time window or environment configuration,

N_failed_unique(w) is the number of unique tests that failed in this window,

N_executed_unique(w) is the number of unique tests that were executed.

Cluster Failure Index reflects the share of tests that failed simultaneously within one time window or one environment configuration. High values usually indicate infrastructure instability.

$$RRR = \frac{F_{\text{recovered}}}{F_{\text{rerun}}},$$

where F_recovered is the number of failures that became green after a rerun,

F_rerun is the total number of failures for which a rerun was performed.

Rerun recovery ratio: the share of failures that disappear after a rerun. This indicator is useful but dangerous when isolated from other metrics: a high recovery ratio does not prove safety; it only shows nondeterminism.

$$TTT = T_{\text{classified}} - T_{\text{first\_failure}}$$

Time-to-triage: the time from the first flaky failure to classification of the cause. A long response time increases the likelihood that the team will begin to ignore the signal.

$$QA_t = T_{\text{now}} - T_{\text{quarantined}}$$

Quarantine age: the time a test remains in quarantine. Quarantine should be a temporary mechanism for protecting the pipeline, not a graveyard for useful scenarios.

$$ES = \max(FR_e) - \min(FR_e),$$

where FR_e is the flaky rate for a specific environment e.

Environment skew: the difference in flaky rate between agents, browsers, regions, containers, or test environments. This indicator is especially important for infrastructure diagnostics.

$$SNR = \frac{F_{\text{actionable}}}{F_{\text{total}}},$$

where F_actionable is the number of flaky failures that led to a confirmed fix or management decision,

F_total is the total number of registered flaky failures.

Signal-to-noise ratio: the share of failures that led to a useful fix in a test, application, or infrastructure relative to the total number of flaky failures. Growth in this ratio shows that the triage process is becoming more mature.

Table 3. Minimum set of metrics for an early-diagnosis dashboard.

Metric	Formula	Unit of analysis	Risk identified
FR_t	F_t / R_t	Individual test	Scenario degradation or brittle test design
FR_c	sum(F_t for t in C) / sum(R_t for t in C)	Component or business process	Product nondeterminism or weak contract
CFI_w	N_failed_unique(w) / N_executed_unique(w)	Time window or environment	Clustered infrastructure instability
RRR	F_recovered / F_rerun	Reruns	Nondeterminism hidden by the rerun policy
TTT	T_classified - T_first_failure	Triage process	Loss of manageability of the test signal
QA_t	T_now - T_quarantined	Test quarantine	Accumulation of invisible test debt
ES	max(FR_e) - min(FR_e)	Environments	Differences between agents, browsers, regions, or test environments
SNR	F_actionable / F_total	Flow of flaky failures	Share of useful signal relative to noise

Flaky Signal Handling Cycle

For flaky tests to work as a diagnostic tool, it is not enough simply to record the fact of a failure. It is important to build a clear handling cycle: from the first failure to verification that the cause has actually been eliminated.

The proposed cycle includes six steps.

1. Event registration. For each failure, the test, commit, branch, environment, CI agent, dependency versions, execution duration, log, screenshot or trace, and rerun result are saved.
2. Initial classification. The failure is assigned to one of the working categories: test code, application behavior, infrastructure, data, external dependency, or unknown cause.
3. Correlation analysis. At this stage, it is checked whether other tests failed nearby, whether the environment changed, whether execution time increased, whether dependent services were deployed, and whether there is a connection with a specific agent, browser, or test environment.
4. Prioritization. The signal is evaluated by diagnostic value and operational cost. Critical business scenarios, cluster failures, and failures associated with key components receive higher priority.
5. Fixing the root cause. The fix may be in the test, application, data, infrastructure, or process. It is important that the pull request record not only the change but also the verified hypothesis: why this specific change should eliminate nondeterminism.
6. Stability verification. After the fix, the test is monitored for a specified number of runs or days.

This cycle shifts work with instability from a ”rerun until green” mode to a mode of engineering diagnostics. A rerun becomes not a way to hide the problem but a source of additional information about its reproducibility.

Conclusion

The analysis shows that flaky tests should be considered not only as a weakness of automation but also as a source of information about the state of the software system. Under this approach, flaky tests cease to be purely operational noise and become an element of early diagnostics, making it possible to identify problems in test code, the application, infrastructure, and engineering processes before they turn into user incidents or release blockers.

The main result of the work is a four-level diagnostic model that combines a classification of instability sources, a system of metrics, a prioritization mechanism, and a flaky-signal handling cycle. Unlike the traditional approach, which is primarily focused on reducing the number of false positives, the proposed model treats instability as an independent object of engineering analysis.

The practical value of the model lies in the possibility of using existing CI/CD data to identify systemic risks without significantly changing the automated-testing process. This makes it possible to increase the informativeness of the test pipeline and make engineering decisions more evidence-based.

Further development of this work is associated with empirical validation of the model on data from industrial projects, comparison of the diagnostic effectiveness of individual metrics, and investigation of the relationship between characteristics of flaky tests and subsequent infrastructure or product incidents.

References

1. Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T. B., & Marinov, D. (2018). DeFlaker: Automatically detecting flaky tests. Proceedings of the 40th International Conference on Software Engineering. https://doi.org/10.1145/3180155.3180164

2. Berndt, A., Bach, T., & Baltes, S. (2026). Flaky tests in a large industrial database management system: An empirical study of fixed issue reports for SAP HANA. arXiv preprint. https://arxiv.org/abs/2602.03556

3. Berndt, A., Bach, T., Gemulla, R., Kessel, M., & Baltes, S. (2026). On the flakiness of LLM-generated tests for industrial and open-source database management systems. arXiv preprint. https://arxiv.org/abs/2601.08998

4. Berndt, A., Nochta, Z., & Bach, T. (2026). The vocabulary of flaky tests in the context of SAP HANA. arXiv preprint. https://arxiv.org/abs/2602.23957

5. Eck, M., Palomba, F., Castelluccio, M., Bacchelli, A., & De Lucia, A. (2019). Understanding flaky tests: The developer's perspective. Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. https://doi.org/10.1145/3338906.3338945

6. Fowler, M. (2011). Eradicating non-determinism in tests. https://martinfowler.com/articles/nonDeterminism.html

7. Gruber, M., Lukasczyk, S., Kroiß, F., & Fraser, G. (2021). An empirical study of flaky tests in Python. arXiv preprint. https://arxiv.org/abs/2101.09077

8. Hashemi, N., Tahir, A., & Rasheed, S. (2022). An empirical study of flaky tests in JavaScript. arXiv preprint. https://arxiv.org/abs/2207.01047

9. Lam, W., Oei, R., Shi, A., Marinov, D., & Xie, T. (2019). iDFlakies: A framework for detecting and partially classifying flaky tests. 2019 12th IEEE Conference on Software Testing, Validation and Verification. https://doi.org/10.1109/ICST.2019.00038

10. 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

11. Luo, Q., Hariri, F., Eloussi, L., & Marinov, D. (2014). An empirical analysis of flaky tests. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 643-653. https://doi.org/10.1145/2635868.2635920

12. Micco, J. (2016). Flaky tests at Google and how we mitigate them. Google Testing Blog. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

13. Parry, O., Kapfhammer, G., Hilton, M., & McMinn, P. (2025). Systemic flakiness: An empirical analysis of co-occurring flaky test failures. arXiv preprint. https://arxiv.org/abs/2504.16777

14. Romano, A., Song, Z., Grandhi, S., Yang, W., & Wang, W. (2021). An empirical analysis of UI-based flaky tests. arXiv preprint. https://arxiv.org/abs/2103.02669

15. Zhang, L., & Elbaum, S. (2014). Prioritizing tests for fault localization through ambiguity group reduction. Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering. https://doi.org/10.1145/2642937.2643008