International Journal of Computational and Experimental Science and ENgineering (IJCESEN)

Vol. 11-No.3 (2025) pp. 5887-5897
http://www.ijcesen.com

ISSN: 2149-9144

Research Article

Flaky Test Automation and Mitigating Test Crashes in Agile Releases

Savi Grover[1*], Sanjay Kumar Das[2]

[1]Software Quality Engineer, Independent Researcher, New Jersey, USA
* **Corresponding Author Email:** savig447@gmail.com - **ORCID:** 0009-0001-6928-1512

[2]Quality Engineering Associate Manager, Independent Researcher, North Carolina, United States
Email: sanjoo.das@gmail.com - **ORCID:** 0009-0007-6472-3186

Article Info:

DOI: 10.22399/ijcesen.3644
Received : 19 June 2025
Accepted : 06 August 2025

Keywords

Flaky Tests,
non-deterministic tests;
test bugs,
software testing,
Systemic Flakiness,
flake associated test crashes.

Abstract:

Flaky tests—tests that fail non-deterministically without any changes in code—pose a serious threat to the reliability and efficiency of automated testing pipelines. They lead to wasted engineering hours, reduced confidence in CI/CD systems, and increased costs. This paper provides an in-depth analysis of the causes of flaky test automation and presents actionable strategies to detect, diagnose, and mitigate test flakiness and crashes. We also explore emerging tools, AI-assisted techniques, and empirical studies that highlight industry practices for maintaining robust and deterministic test suites. In risk management and software testing industry, prevention aims to stop an event from happening in the first place, while mitigation focuses on reducing the severity or impact of an event that has already occurred or is unavoidable. Prevention is proactive, trying to stop a problem before it starts, while mitigation is reactive, managing the consequences of a problem. In this research, we are broadly describing techniques of mitigation of occurred and detected flaky test failures and automation crashes.

1. Introduction

In the modern DevOps and Agile landscape, automated testing is a critical component of the software development lifecycle (SDLC). However, the prevalence of flaky tests undermines the effectiveness of continuous integration (CI) systems. These intermittent failures may falsely indicate defects, disrupt release cycles, and degrade team morale. Understanding the nature, causes, and remedies of flaky test automation is crucial for engineering quality at scale.

About test flake: Automated tests are expected to be deterministic and produce the same result (pass or fail) on every run of the same code input. Flakes happen when a correct test can fail on consecutive runs. There are often a lot of variables at play that may cause this to happen for tests that are not a simple I/O challenge, but a complex logic flow with mission-critical timing aspects (i.e. especially in integration tests.) Often what happens in practice is that the test will pass frequently on local runs, enough to be merged into the development base branch. This unreliable test will then make its way into other PRs and other unrelated work that will see the test fail. This causes confusion on the developer and testers side as the test suite is failing, and checks prevent merging of code that (often) did not cause any regressions on its own.

At this point, as a team we choose to override what our test suite is telling us is a failure to push code through the boundaries that are set up to protect us. There is a lot of on us to then distinguish between the nuance of an “acceptable break” and an “unacceptable break.” In Agile, we want the power to be able to know exactly when regressions happen from our test suite’s insight into the validity of features and functionality. Until we solve the inevitable problem of flake, we cannot fully rely on our automated testing practices to identify real application bugs when they happen. Lot of large-scale organizations have tackled flakes with in-depth solutions and metrics to give engineers the knowledge on how to mitigate the risk of these tests failing the suite continuously. Both Facebook and Spotify have published talks/docs on the subject area [1]-[2]

1.1 Study Background and previous works

According to study conducted on a team of Mozilla developers and their flaky test analysis by [3] University of Zurich, Switzerland in 2019, there was a detailed account of understanding, definition

and investigation of causes of repetitive flaky tests. It was done through qualitative and quantitative analysis of developer's contributions in fixing these tests, and by their answers to a few predefined sets of questions like – how developers categorize these flakes, how problematic are they perceived and main challenges while dealing with these interruptions. This research brought out an interesting approach of developer's perspective in understanding them and how costly flaky tests resolutions are and issues being still unpredictable and reproducible.

This research was later expanded in [5] in 2020 in direction of life cycle of flaky tests so that – prevalence, reproducibility, reoccurrence, execution time and time-before-fix determination of the flake tests – so that a preemptive impact and possible solution can be found out. This research was done on many open sources and six proprietary projects in Microsoft and a Flakiness and Time Balancer (FaTB) solution was concluded which removes the negative impact of Async Wait tests along with CloudBuild- an incremental and distributed system for building code and running tests which when receives a build Pull request with a change, it identifies all modules that are impacted by the change and executes the tests only in those impacted modules, and skips the remaining modules tests, since none of their dependencies changed, hence saving time for retry and flake tests identification.

In a continuous study by [6] University of Luxembourg in 2021, detects the characteristics of flake by data collecting from 14 industry practitioners by qualitative questions, to find out flake reduction infrastructure and coding, automation guide for best practice framework for preventing flakiness. [6] in 2022 combines and summarizes – all aspects from causes, detection, implications and simple mitigating steps like- quarantine, re-fixing, skipping and remove testcases and how remove/ignore strategy is the principle behind the industrial flake mitigation tool like - Probabilistic Flakiness Score (Facebook), Spotify has Flakybot which is designed to help developers determine if their tests are flaky before merging their code to the master branch. The tool can be self-invoked by a developer in a pull request, which will exercise all tests and provide a report of their pass/fail and possible flakiness. Similarly quarantining the test case is adopted by Google, Flexport and Dropboxs.

1.2 Recent ML advancements

In a recent revolutionizing study in 2023 [7] by a few independent researchers in detecting flakiness by LLM based techniques and quantitative analysis behind the detection, and re-run to reoccurrence ratio and many code evolution and test historical data flakiness detection. And a 2025 [8] – an analysis of co-occurring flaky test failures was done by mix method approaches like – Clustering, Prediction and Manual Inspection in Python automation tests. Both of this research are based on AI-ML techniques- but prove to be really time consuming and expensive for software industrial purposes, both involve a dataset of 10,000 test suite runs, which is somewhat unrealistic for real-world web based, customer-facing, b2b and enterprise application associated with a single company and its technical scenarios.

1.3 Focus of this research

The study behind this research lies on discussing - simple, affordable, easily implementable and adopted techniques to mitigate flaky tests. Note that- prevention of flake tests means avoiding them to occur at all – which practically is non avoidable in agile, but mitigation procedures should be still inexpensive for small to medium technical applications comprising of target users of definite number along with developer friendly programming support which can help to integrate them with the existing test automation setup. These are efficient and easy get arounds to work in speedy release planning, countering with predominant flakes in agile sprints, require less time and less effort to understand and bring in common code optimization practices.

This research also covers some real case studies where flaky test minimization solutions were discovered and implemented by different organization and their reliability and success score.

2. Characteristics and Causes of Flaky Tests

As discussed in many different previous works, flaky tests are typically characterized by- non-deterministic behavior, where unchanged code may or may not result in pass tests for identical environments, may be caused by lack of consistent reproduction steps, loss of processing storage or resources, dependent conditions, often limited correlation with actual defects, page/application load issues, parallel execution, sudden network and API failures- and API edge cases not implemented. Combining all sources and causes, from many respective studies the causes can be better understood by flaky test categorization- concurrency, dependent test cases, network latency, randomness of external factors, time, async waits, external state/behavior, hardware [6]. Flaky tests

are not easy to catch and the frequency of flakiness is always a struggle to deal with, some can happen frequently, and others are so rare that they go undetected. Moreover, tests can be flaky as a direct result of the testing environment they are run in. Indeed, they can find their origin from multiple sources, some of which can help you find bugs you would never have without their flakiness. Some poor infrastructures or test environment designs can then be unveiled [9].

Many categories and causes seen and observed in the software technical industry can be classified by following -

1. Environmental Factors- Hardware variability, network latency, resource constraints, CPU lacking, memory lacking, underlying infrastructure and system load, resource deadlocks, different browsers behavior in same application, low device compatibility issues.
2. Timing and Synchronization Issues- Async Operations (AJAX requests) can be particularly prone to timing issues if not handled correctly, invocations left un-synchronized, or poorly synchronized**.** Race conditions, shared resources, improper waits, indefinite loops, sleep times in automation /code design.
3. Test Dependencies- Test code issues, uninitialized variables, improper data, improper cleanup of pre-requisite conditions, low garbage collection, absent exception handling, Third-party services, databases, APIs with inconsistent states.
4. Inconsistent Test Data- Dynamic data sources, real time data changes, accidental or intended data modification within tests, hardcoded values, too restrictive test range – not all expected outcomes/options are considered while test writing.
5. Uncertain application behavior- randomness, API edge cases not implemented, lack of application knowledge, incorrect assumptions, insufficient assertions. Randomness is a broad category of test flake reasons, especially when the traits of algorithms in various machine learning applications, including inherent randomness, probabilistic specifications, and the lack of solid test oracles [10], pose significant challenges for testing these applications.
6. Poor UI and test design- complex UI, unclear happy path, too many warning messages, too many alerts and popups, random web sleep instance, CTA action, lack of debugging processes, lack of timely bug fixes, unknown app issues and pre-conditions, too many pre-requisites which are hard to maintain and produce, lack of isolation and orphan code.
7. External Factors- CI misconfiguration, parallel execution problems, poor test design, test framework and code non optimized code structure, lack of application knowledge.

3. Easy Identification and detection of Flake in Agile sprints

Need to fix flake in Agile Sprints- Since more developers are in a hurry to send across changes, finish bug fixing, or releasing feature launch and chasing deadlines, there are deadliest chances of producing test crashes, common files impact, recurring inconsistent application behavior and code smells. Ensuring quality code into production, if flaky tests are ignored, creates bugs or issues in code can be easily overlooked. There are several different factors that can cause flakiness, and all of them can slow the CI/CD pipelines and deployments or manifest issues of application's end user [13]

Detecting and identifying flakiness- Detecting and identifying flakiness in a test automation setup is an extremely frustrating process, it can be found easily in some situations by spotting repetitive varying results of same tests but in some complex applications, reasons of test break can be in a wide range. To broadly detect flaky tests, focus on identifying tests that exhibit inconsistent results across multiple runs. This can be achieved by rerunning tests, analyzing historical test data, and utilizing specialized tools. Additionally, consider testing in different environments and parallel execution to pinpoint potential issues. The need to detect and fix these tests- is the main reason which help us gather the consequence or impact of their presence in the test automation setup.

3.1 Heuristics-Based Detection

This is an efficient technique which can be adopted easily in Agile methodology. It is based on measuring the flake occurrence and quantifying flake reproducibility by Re-run tests mechanism. By setting a threshold to these tests we can outline the number of flake tests and isolate them to save test execution time.

Methodology

By running multiple tests N times in a test suite T where {t1, t2, t3…tn} are testcases and finding out F out of T where F=number of flake tests and a subset of T by marking tests failure variance over time.

- Re-run test multiple times (N times rule)

- Mark tests with high failure variance over time. (Finding both results = pass and fail for a testcase)

Method Description

Input:

- A test suite $T = \{t_1, t_2, ..., tn\}$
- A configurable re-run count N (typically N = 5–10) = 5 to 10 times

Output:

- A set of flaky tests $F \subseteq T$ identified by heuristic rules

Steps:

1. **Repeated Execution Strategy:** For each test $ti \in T$, execute the test N times in an identical environment.

   Record the results as a binary outcome: pass (✓) or fail (✗).

   Run(ti) = [✓, ✓, ✗, ✓, ✗, ..., ✓] → Result Vector Ri

   Flakiness Heuristic Rule: Define a test as flaky if the result vector Ri contains both pass and fail outcomes. That means a testcase ti is flaky if it has gotten both pass and fail in those N tests

   Formally:

   ti is flaky if $\exists$ j,k$\in$[1, N], $j \neq k$ such that $Run(ti)[j] \neq Run(ti)[k]$

   This simple "variation-based" heuristic is the most widely used.

2. **Assign a Stability Score Calculation to a testcase (Optional Enhancement):** Assign a **Flakiness Score (FS)** to each test as:

   A test can be prioritized for investigation if $0 < FS(ti) < 1$, indicating intermittent failure.

3. **Threshold Filtering:** Apply configurable thresholds to classify test outcomes:

   a. $FS(ti) = 0$: Stable Test

   b. $FS(ti) = 1$: Consistently Failing Test (likely a real bug)

   c. $0 < FS(ti) < 1$: Potentially Flaky Test (candidate for further triage)

4. **Environment Consistency Check:** Ensure that test environments (OS, browser, container state) remain consistent across runs to eliminate false positives due to environmental noise.

Example: Suppose test t1 is run 10 times with the following outcomes:

Run(t1) = [✓, ✓, ✗, ✓, ✓, ✗, ✓, ✓, ✓, ✓]
→ FS(t1) = 2/10 = 0.2 → Marked as Flaky

Advantages:

- Simple and easy to implement in any CI/CD pipeline
- Low computation cost
- Effective at identifying flakiness due to timing, concurrency, or nondeterministic behaviors

Limitations:

- May miss flakiness that appears infrequently (requires high N)
- Not effective if environment variability is not controlled
- Can result in overfitting (i.e., falsely classifying flaky tests due to a single failure)

3.2 Categorization by failure type to label the reproducibility of flake

- **Model**: Analyze the stack traces or error messages from failed tests and categorize them (e.g., assertion errors, timeouts, network failures, etc.).
- **Identification**: Recurring failures of the same type under different circumstances might suggest flakiness (e.g., repeated timeouts indicating a race condition).
- **Benefits**: Helps pinpoint the root cause of flakiness by providing insight into the type of failures occurring.

Advantages:

- Simple and easy to implement in any CI/CD pipeline
- Low computation cost
- Effective at identifying flakiness for even false positives

Limitations:

- May take extra developer's time in tagging/comment their tickets and GitHub submissions with proper labels and categories.

Logging efforts by test and product teams

4. Flake Mitigation Strategies in Agile Releases

4.1 Developer's Best Practices Guide

4.1.1 Employ deterministic inputs

Indeterminism in code testing arises from factors that lead to inconsistent test results despite identical test inputs and code. This makes tests unreliable and hinders the development and maintenance process. Here are several strategies to mitigate indeterminism

- Test properties of the output rather than the output itself: Instead of asserting an exact output, focus on verifying the properties that the output should possess. For example, if a dice roll is non-deterministic, you could verify that the rolled number is an integer, between 1 and 6, and not negative.
- Handle multiple valid outputs: If a piece of code can have several valid outputs, configure your tests to accept any of them. For instance, if a webpage lists products whose order might vary, you can prepare screenshots for both possible orders and verify if matches the current output.
- Output transformation: If the non-deterministic output has a predictable structure but varies in specific details, transform the output before testing. For example, if a program outputs "You rolled a 6!", "You rolled a 1!", etc., you can transform it to "You rolled an N!" and then test this generalized output.
- Mock external dependencies: Replace external services like databases or APIs with mock objects that provide predictable responses. Mocking API responses for integrated tests and developer's isolated unit tests before releasing code to upper environments. Implementing edge cases like resource not found/value not found with respect to UI.

4.1.2 Avoid code cycles/Performing Loop Testing

Many code cycles in application path leads to a greater number of flakes, as it increases indeterminism in tests. The "code cycle testing" refers to the specific technique of Loop Testing. This technique focuses on thoroughly testing loops within the code to ensure they function correctly under various conditions.

- Identify errors at loop boundaries: Loops are prone to errors at their entry and exit conditions, as well as during their first and last iterations. Loop testing aims to uncover these issues.
- Verify proper loop execution: It ensures that the loop executes the correct number of times and that the operations within the loop perform as expected.
- Detect structure of loops: Simple, nested, indefinite, incremental, conditional, concatenated and unstructured loops - to identify and prevent situations where a loop might run indefinitely, leading to program crashes or resource exhaustion.

4.1.3 Remediating the Root Cause of occurred Flaky Tes**ts**

To remediate/reproduce flaky tests, developers should know where to look for the root cause of flakiness, but this can prove to be challenging since it might require manually sifting through many lines of code. One way to identify flakiness is to re-try a test several times and document each time the test displays contradictory behaviors. [13]

4.1.4 Look Before You Leap

This represents clusters of flaky tests that could be repaired or at least mitigated by checking the status of some external system or resource. Examples include checking for the existence of a directory/file path and confirming that a server is running. Looking on pre-requisite conditions before triggering test execution. [8]

4.2 Automation Test Design Improvements

4.2.1 Isolate tests to ensure independence - Ensuring tests are isolated and independent is another key strategy. Adopt the independent test pattern, which means each test should be self-sufficient and not rely on external systems. This reduces the likelihood of tests failing due to external dependencies like APIs or databases [14].

4.2.2 Quarantining Flaky Tests- Dynamically quarantine flaky tests to remove them from the critical path, allowing development to proceed without being blocked by intermittent failures. This involves continuing to run the tests but preventing them from failing the building, creating a clear signal that they require attention. Automatically file bugs or tickets for quarantined tests to ensure they are addressed by the responsible team members.

4.2.3 Combining many small tests into one consistent happy path test- Instead of having many small verifications in each individual testcase, example – verification of elements of a page can be combined with submission of form elements data and verifying submit message. This technique reduces the number of tests in a test suite, thereby optimizing execution time, test reaction, and more coverage in a reduced number of tests.

4.2.4 Run at an appropriate time of the day- Sometimes, code behavior depends on load and network throughput during the day. This means test success may depend on the time the test runs. For example, we are running automation tests on test environments during the middle of the day vs scheduled tests during nightly background jobs when there are minimum network requests. Both these tests produce different results at different times. For better network availability, automation pipelines should be scheduled for nightly runs.

4.2.5 Timeout Strategies - Proper timeout, wait and async wait configuration is vital to avoid hanging tests. Set appropriate timeouts to ensure tests do not run indefinitely. For instance, if a test involves a network request, set a timeout to handle potential delays. Mocking simulates the behavior of external dependencies, ensuring your tests run smoothly without waiting for real systems. Tools like Mockito for Java or Sinon for JavaScript can help create effective mocks, leading to faster and more reliable tests. Rather than relying on fixed time intervals, fine-tune the wait conditions using explicit waits based on specific conditions. Implement dynamic waiting strategies to synchronize with the application's state changes.

4.2.6 Test Order dependency - A test might fail because of the test that runs before or after it. This happens because many tests use shared data, like state variables, inputs, and dependencies, simultaneously. We need to completely remove or minimize the dependencies among these tests to improve accuracy and reduce flakiness. Wherever your test depends on another module, use stubs and mocks. Stubs are objects with predefined responses to requests. Mocks (also called fakes) are objects that mimic the working representation, but not at 100 percent of production. Mocking and stubbing creates tests that run in isolation. [15]

4.2.7 Proper set up and tear down /clean up methods- Setting up proper cleaning mechanism for initializing start up and tear down methods to run at the beginning and termination of test suites. These methods contain common pieces of code/pre-steps for system start up and shut down and involve less chances of changes by automation developers while script writing. They also ensure easy reproducibility, a smaller number of lines, ease for retry mechanisms and quickly identify flakes.

4.2.8 Retry Mechanism- By automatically re-executing tests that fail, a retry mechanism can:

Bypass transient failures: Many flaky tests are caused by temporary issues. Retrying them can allow them to pass the second or third time, preventing unnecessary debugging.

Identify genuine failures: If a test consistently fails after multiple retries, it indicates a more serious underlying issue that requires attention.

Reduce false alarms: Retries can prevent a build from failing due to temporary glitches, reducing the number of false alarms and saving debugging time.

4.2.9 Automation tests standardization- Avoid stale elements, use relative xpaths, avoiding null pointer exceptions, use explicit implicit waits, avoid hard sleep times, page object files segregated with class runner files, not using hard coded values, use unique locators, careful iframe/window jumping, better output matching scenarios – not too restrictive range when outputs are more, using test reports and integrated CI-CD pipelines are some automation setup practices to reduce flake.

4.2.10 Regular Test Maintenance is crucial for flaky test mitigation. Start by scheduling routine reviews of your test suite. These reviews help identify and remove redundant or outdated tests. For example, tests that no longer align with current code or requirements should be discarded. This keeps your test suite lean and focused.

Next, address issues promptly. When a test fails, investigate and fix the problem right away. Proactive fixes prevent small issues from escalating into larger problems. Regular maintenance ensures your test suite remains reliable and up to date. [14]

4.3 Infrastructure Enhancements and CI-CD Process Improvements

- Environmental Consistency- Containerization tools such as Docker help to maintain consistent test environments. This can also be provided by external server and cloud platforms. By replicating the same environment for each test run, you eliminate variations that could cause flaky tests. This consistency is essential for reliable test results.
- Concurrency- Look for shared resources or critical sections that cause contention among the concurrent tests. To address the issue, implement a locking mechanism or other

concurrency control strategy to prevent interference and resource locking and ensure isolation. If data modification occurs before running another test, we need to run the tests in isolation, automation code should not comprise of reused variables.

- Constant network /hardware availability- Employ network and hardware services with tolerant systems, mock services for flaky external dependencies, along with device browsers and stable and compatible OS versions to run tests on cloud (Lambda test) or virtual containers. Maintain Resource capping and network shaping in test environments Checking for memory leaks in between execution and storage/resource conflict.

- Using visualization, error logging and exceptions to catch flake and debug them- Metrics-driven dashboards to highlight flaky tests like Datadog, Google GCP alerts help highlight job failures, CRUD issues, migration job failures, API failures and response codes. These can help debug flake without need of re-run.

5. Innovative Methodologies for improving Flaky tests Mitigation.

5.1 **Dynamic Test Skipping-** This method focuses on dynamically separating out flake tests and skipping them to exclude from current test runs. Reasons due – to known bugs, known defect categorizations or upcoming revamp/feature build up around flake tests, tests can be skipped/or removed from test run to overall avoid encountering any flake and save time. In the situation if developer is working on new feature development and begins to write tests that seem to pass locally but are not reliable in CI (Github)

Conditional skipping based on past results: Using historical test data or specialized tools, identifying tests with a high probability of flaking, and automatically skipping them in the current build. Quarantining flaky tests - isolating unreliable tests into a separate suite prevents them from blocking the main build while the root cause is investigated. Evaluating frameworks with dynamic skipping features include testing frameworks, such as Pytest or TestNG, offer mechanisms to conditionally skip tests during execution, which can be leveraged for dynamic test skipping.

5.2 **Test Result Decisive Algorithm for Frequent flakes** - Calculate the flake rate and PFS (Probabilistic Flakiness Score) for each test. Tests exceeding a pre-defined threshold for both metrics are flagged as "highly flaky" and passed on Task to fix with – measured impact, categorization, root cause and priority to fix.

Using this numeric, developers can measure and monitor the flake in regression in form of PFS each individual test – also done in Facebook to monitor changes in its reliability over time. “If we detect specific tests that became unreliable soon after they were created, we can direct engineers’ attention to repairing them.” [6]

Meta's approach uses Bayesian inference to quantify a test's flakiness based on past results. For example, running a test 10 times and finding the flake rate to be 2, the probability= 0.2 and assuming a set threshold of 0.5 or above. In this case

Since probability here is less than assumed threshold-> we consider this test non flaky.

- Track the impact of fixes on flakiness metrics. Monitor whether the flake rate and PFS decrease after implementing solutions.
- Continuously refine the algorithm by decreasing the threshold with every fix made in flake test.

GitHub also adopted a similar metrics-based approach to determine the level of flakiness for each flaky test. An impact score is given to each flaky test based on how many times it changed its outcomes, as well as how many branches, developers, and deployments were affected by it. The higher the impact score, the more important the flaky test and thus the highest priority for fix is given to this test. [16]

This method is also called as Matrix Method/ Square Method- also utilized by Spotify Engineering - At Spotify, engineers use Odeneye, a system that visualizes an entire test suite running in the CI and can point out developers to tests with flaky outcomes as the results of different runs. Another tool used at Spotify is Flakybot5, which is designed to help developers determine if their tests are flaky before merging their code to the master/main branch. The tool can be self-invoked by a developer in a pull request, which will exercise all tests and provide a report of their success/failure and flakiness. [17]Looking at the image Figure 1 above, we can see individual tests vertically and horizontally the result of these tests running in CI. If we see a scattering of orange dots this usually means test flakiness. If we see a solid column of failures this usually represents infrastructure problems such as network failures and things of that nature. Graphs like this are a wonderful way to help you establish what is flaky and what is an infrastructure problem.

5.3 **Early Locator Verification-** This mechanism can be useful for simple static automation UI tests where a prior presence of all elements can be done by a common reusable method, before executing any functional statements in test. If the elements are

passed for their presence- we can assure – no issues with stale elements, page load or timeout /network issue. If the test failed after this verification, the test team may assume it be still "passed".

In situations of less time remaining in sprint releases and knowing the fact that code changes are not impacting other classes associated with flake tests.

Limitations of this technique- It is only useful for UI display operations, does not guarantee 100 percent accuracy/ or bug discovery in the functional analysis of page elements.

5.4 **Visual Regression** - Computer vision tools, like those offered by Applitools Eyes or Percy, leverage machine learning algorithms to compare screenshots or UI elements across different builds, devices, and browsers.

They detect visual regressions, layout shifts, font changes, and other subtle UI inconsistencies that might cause tests to fail unexpectedly. By proactively identifying these visual anomalies, computer vision helps to catch potential UI flakiness before it becomes a problem, ensuring a more stable and reliable testing suite.

5.5 **Self-healing AI scripts-** Traditional test automation often relies on static locators to identify UI elements, which are prone to breaking when the UI changes. Computer vision-powered tools can analyze the visual properties of UI elements, including their shape, size, color, and relative position, enabling more robust object identification. Some advanced tools even offer self-healing capabilities, where AI algorithms can dynamically identify and track UI elements even when their attributes or positions change, thereby automatically updating test scripts and reducing maintenance overhead.

5.6 **Code-less / script-less tools for flaky tests-**

- Simplified Test Creation and Maintenance: LC/NC tools utilize visual interfaces, drag-and-drop functionalities, and pre-built actions to create test cases. This simplifies the process, making it less prone to human error that can lead to flakiness. When application changes occur, these tools often offer features like self-healing locators or dynamic element detection, automatically adapting test scripts and reducing the need for manual updates that can introduce new flakiness.
- Reduced Reliance on Complex Code- Flaky tests often stem from intricate or poorly written code within test scripts. LC/NC platforms minimize the need for complex coding, reducing the potential for coding errors, synchronization issues, or timing-related problems that contribute to test flakiness.

5.7 **Exception Handling-**

- Managing Race Conditions and Asynchronous Operations:

In concurrent or asynchronous code, race conditions can lead to inconsistent test outcomes. Exception handling, combined with appropriate synchronization mechanisms (like try-catch-finally blocks around shared resources or await statements in asynchronous code), can help ensure tests handle these scenarios predictably, preventing flakiness.

- Preventing Test Failures due to Expected Edge Cases:

Sometimes, the system under test might throw exceptions for valid, albeit less common, scenarios. If a test is designed to verify the handling of such an edge case, exception handling within the test allows it to specifically catch that expected exception and assert its presence, rather than failing the test entirely.

5.8 **Flake fix procedure-** Treat flaky tests as bugs: Prioritize fixing flaky tests and incorporate their review into the regular code review/development process to foster a culture of quality. Identifying the urgence, severity, acceptance criteria, test scenarios of the defect and investigate root cause. Document the causes of flakiness and the strategies to address them. Sharing these learnings can help prevent similar issues and speed up fixing.

5.9 **Cut, Fold and Burry in same code version-** This techniques is used for test-cases that show flakiness when there is no new code deployment, that means we are re-running tests in the same code version, if we have a few positive's in such a scenario, then testcase are marked cut (or zipped) at the point of flake, folded and marked as passed and buried or dumped after the point of flake found. This method can be also called **Box, zip and dump method** which is simply used to mark flaked test-cases to "passed" if they have shown passed results previously in same code version.

4. Conclusions

Flaky tests represent a persistent challenge in automated testing pipelines. Addressing them requires a blend of test design rigor, infrastructure improvements, intelligent tooling, and cultural investment. By building a culture of quality within a development team is crucial for effectively addressing and minimizing flaky tests. Integrating test quality into performance reviews and goals, include metrics related to flaky tests and test stability in individual and team performance evaluations to reinforce their significance and hence their mitigation.

Table 1. *Some Detection techniques for flaky and non-deterministic tests*

Detection Methods	**Ways to Implement**
Tests re-execution	A straightforward method is to repeatedly execute the same test suite under the same conditions. If some tests pass and others fail without any code changes, we've likely found a flaky test. Running the tests both sequentially and in parallel. If a test consistently fails only during parallel execution, it may indicate a race condition or test order dependency issue. Run tests in different environments (e.g., with different configurations or resources) to see if the results vary. Running the test in different environments to see if the failures are environment specific. This can help pinpoint environmental factors contributing to flakiness.[12]
Comparison of old and new test results with respect to current release changes/code files	Examine the CI/CD system's test results to identify patterns. Look for tests that fail frequently on specific branches or at times. Tools like Jenkins, GitHub Actions, or CircleCI can help with this. Leveraging CI/CD platforms with built-in flaky test detection features can save a lot of time. These platforms automatically identify tests that fail intermittently. For example, Semaphore CI and CircleCI offer flaky test detection tools. These tools analyze test results and flag tests that show inconsistent behavior. Automated detection not only saves time but also improves the reliability of your test suite, allowing you to focus on fixing the root causes.[11]
Targeted Testing around basic application flow (by differently purposed teams)	This can be non-re-run mechanism where after every deployment, the test team can request other supporting teams to run a quick smoke test analysis / or their daily tasks on the applications. Teams such as- test team, Ads team, design team, or sales team or HR team can just do necessary small checks or just their daily tasks and can report any discrepancies or unusual pattern they see. This can be a very unimportant measure which is not adopted across big companies but can save a ton of time in determining a flake or finding an indeterministic test around application.
Observing test execution history	Check the history of suspicious test- an alternating trail of pass/fail result may be a sign of flake.
Manual Test and maintain a Root Cause Tracker notebook (of old bugs)	If automated methods don't pinpoint all flaky tests, manual checks can be useful for identifying potential issues. Categorize test failures to understand the root causes and prioritize remediation efforts. Keep track of flaky tests using issue-tracking systems, spreadsheets, or specialized tools to monitor the cause of old, unrelated, related bugs, impact and debugging measure which were used if a flake used to be an actual old defect.
Catching GitHub pr's, pattern, local tests, change in config file, and file modification history	Early detection of code files where multiple people are working simultaneously. Track any config file changes, unit tests or even time clash of submitted pr's by two or more developers working on similar classes.
Log Monitoring and internal alerts	Examine test logs for patterns or inconsistencies in error messages. For example, timeouts or failed connections can be signs of a flaky test.
External tools	• Test Retry Plugins (e.g., Jest Retry Times, pytest-rerunfailures) • CI Dashboards with flake tracking features • Custom Scripts that flag inconsistent test outcomes • Prometheus and Grafana: These tools can monitor and visualize metrics, including test execution times and success rates, to help identify flaky tests.
Targeted testing of complex code files – before re-runs	Code changes which are ambiguous consist of complex code conditions, async time or concurrent condition, assertions or even indefinite, incremental loops- can be tested before hand in isolation or integration before running around whole test suite.
AI and Machine Learning Detection techniques	Heuristics-Based Detection, Statistical Methods, Prediction of flakiness using feature vectors, classification, Sophisticated Pattern recognition (code changes to bug patterns), Anomaly detection model, Time series analysis.

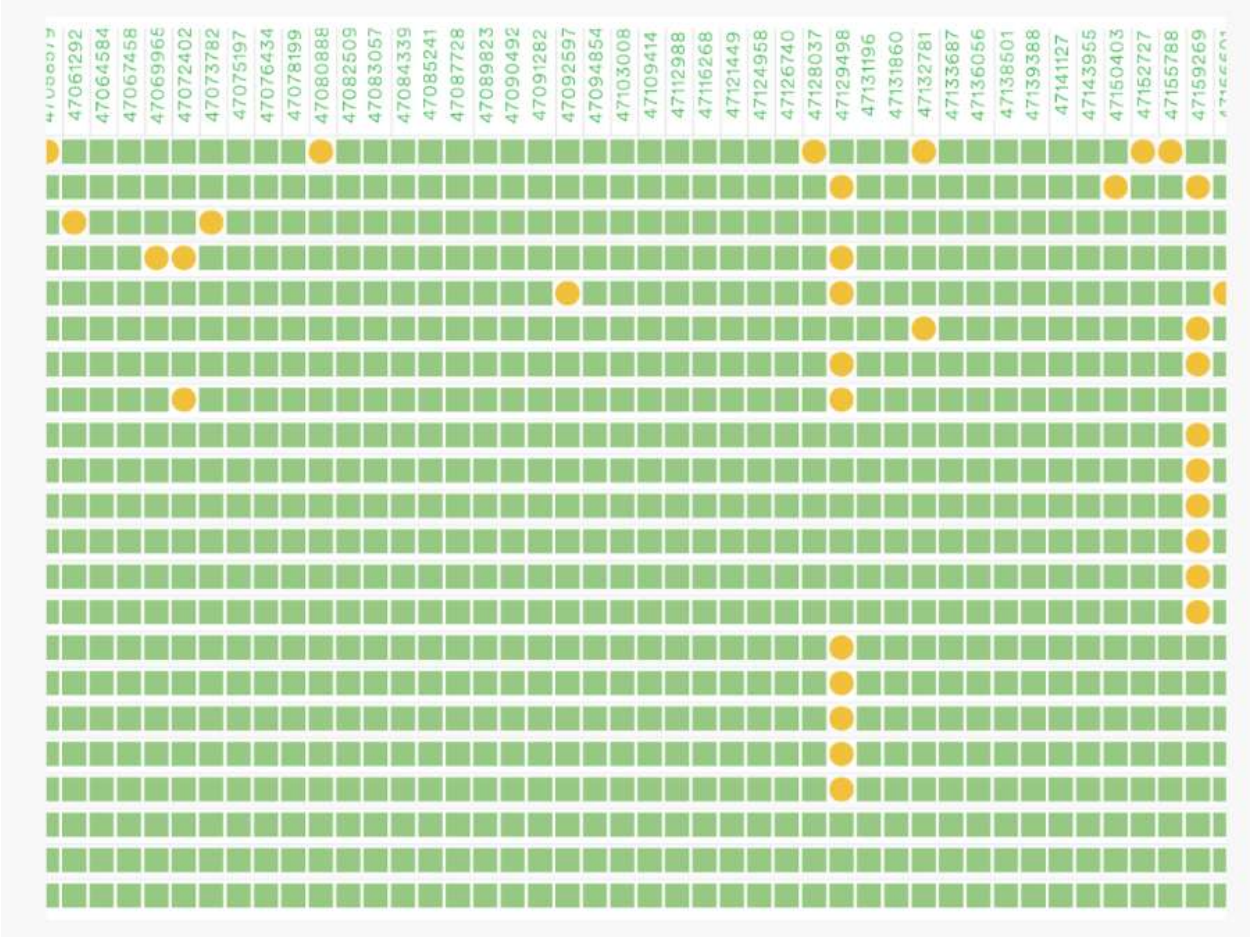

***Figure 1**- Tracking Flakiness at Spotify by Odeneye. Source [17]*

Author Statements:

- **Ethical approval:** The conducted research is not related to either human or animal use.
- **Conflict of interest:** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper
- **Acknowledgement:** The authors declare that they have nobody or no-company to acknowledge.
- **Author contributions:** The authors declare that they have equal right on this paper.
- **Funding information:** The authors declare that there is no funding to be acknowledged.
- **Data availability statement:** The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

References

[1] https://blog.mergify.com/flaky-tests-who-are-they-and-how-to-classify-them/

[2] Detecting Flaky Tests in Probabilistic and Machine Learning Applications. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020.

[3] Understanding Flaky Tests: The Developer's Perspective - University of Zurich Zurich, Switzerland, 2019

[4] A Study on the Lifecycle of Flaky Tests – Microsoft and University of Illinois at Urbana-Champaign Urbana, Illinois, USA, 2020

[5] A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests - University of Luxembourg, 2021

[6] Test Flakiness' Causes, Detection, Impact and Responses: A Multivocal Review, School of Mathematical and Computational Sciences, Massey University, New Zealand – 2022

[7] Practical Flaky Test Prediction using Common Code Evolution and Test History Data – 2023

[8] Systemic Flakiness: An Empirical Analysis of Co-Occurring Flaky Test Failures – 2025

[9] https://blog.mergify.com/flaky-tests-who-are-they-and-how-to-classify-them/

[10] Detecting Flaky Tests in Probabilistic and Machine Learning Applications. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020.

[11] https://trunk.io/learn/best-practices-for-finding-and-mitigating-flaky-tests#how-to-identify-flaky-tests
[12] https://www.testrail.com/blog/flaky-tests/
[13] https://www.datadoghq.com/knowledge-center/flaky-tests/
[14] https://trunk.io/learn/best-practices-for-finding-and-mitigating-flaky-tests#best-practices-to-mitigate-flaky-tests
[15] https://circleci.com/blog/reducing-flaky-test-failures/
[16] "Reducing flaky builds by 18x," Dec. 2020. [Online]. Available: https://github.blog/2020-12-16-reducing-flaky-builds-by-18x/
[17] https://engineering.atspotify.com/2019/11/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests