Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2022, 9(6):73-78

Research Article ISSN: 2394 - 658X

Handling Flaky Tests in Java and JavaScript Using Retry Mechanisms and Smart Test Selection

Praveen Kumar Koppanati

praveen.koppanati@gmail.com

ABSTRACT

Flaky tests are tests that exhibit inconsistent results despite no changes in the codebase. They have become a significant challenge in modern software development, particularly in languages like Java and JavaScript, which are widely used in enterprise and web applications. This paper explores strategies for managing flaky tests through retry mechanisms and smart test selection approaches. We investigate common causes of flaky tests in these two environments and propose solutions that mitigate their impact on Continuous Integration/Continuous Deployment (CI/CD) pipelines. We also review existing frameworks and tools that support automated flaky test detection and remediation. By leveraging retry mechanisms and adaptive test selection, development teams can optimize their testing pipelines, ensuring reliability and stability in production environments.

Keywords: Flaky tests, Java, JavaScript, Retry mechanisms, Smart test selection, CI/CD, Software testing, Test reliability.

INTRODUCTION

Software testing is an integral part of the software development lifecycle, especially in agile and Continuous Integration/Continuous Deployment (CI/CD) environments. As the demand for quicker releases grows, so does the reliance on automated testing to ensure code quality. However, the tests that produce different results when run multiple times without changes in the underlying code also known as Flaky tests undermine this process, leading to wasted developer time and raise doubts in the test suite.

In this paper, we investigate two approaches to handle flaky tests in Java and JavaScript environments: retry mechanisms and smart test selection. To simplify, Retry mechanisms allow rerunning failed tests to account for nondeterministic failures, while smart test selection optimizes which tests to rerun, thereby reducing unnecessary test executions. We examine these techniques in the context of both Java and JavaScript, highlighting their respective challenges and advantages.

The Growing Importance of Software Testing:

As software projects become larger and more complex, the importance of reliable automated tests has increased. Automated testing is crucial for catching bugs early in the development cycle and ensuring code quality as part of CI/CD processes. However, the presence of flaky tests undermines the trust in test results and slows down the entire development workflow.

Test flakiness refers to the behavior where tests produce inconsistent results without any changes in the code they test. A test might pass during one run and fail during another, even though the underlying code has not been altered. Flaky tests are a common problem, and they introduce several challenges:

• **Wasting developer time:** Developers need to spend considerable time diagnosing flaky tests, determining whether a test failure is due to an actual bug or simply a flaky behavior.

• **Slowing down CI/CD pipelines:** Flaky tests can interrupt continuous integration processes by introducing noise into the test suite, requiring unnecessary reruns or manual verification.

• **Reduced confidence in test suites:** Frequent flaky tests lead to a lack of confidence in automated test results, undermining the entire testing strategy.

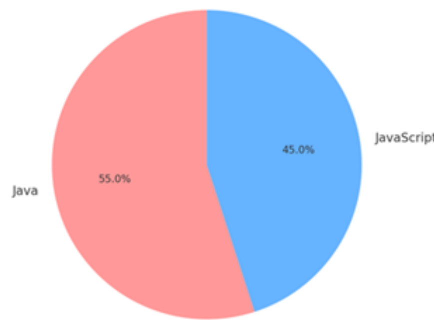

Fig. 1 *Proportion of software projects impacted by flaky tests in Java vs. JavaScript.*

This paper explores how retry mechanisms and smart test selection can be combined to minimize the impact of flaky tests in Java and JavaScript environments.

FLAKY TESTS: AN OVERVIEW

Definition of Flaky Tests:

Flaky tests are defined as tests that pass and fail intermittently on the same codebase due to nondeterministic factors. These factors can include environmental issues (e.g., network availability), concurrency, test order dependency, or improper test design. The unpredictability of flaky tests makes them hard to diagnose, leading to reduced developer confidence in the testing process.

Some of the common characteristics of flaky tests include:

- **Intermittent behavior**: A flaky test can pass during some runs and fail during others without any changes **in the code or environment.**
- **False negatives:** Flaky tests often generate false negatives, meaning they report a failure when there is no actual bug in the system.
- **Environmental sensitivity:** Many flaky tests are sensitive to external factors, such as timing issues, network conditions, or system resources.

Flaky tests are a serious problem in CI/CD pipelines because they interrupt the process of delivering reliable software. Developers waste time analyzing failures that aren't caused by bugs, and the entire release process may be delayed.

Causes of Flaky Tests in Java and JavaScript:

Understanding the root causes of flaky tests is essential to mitigating their impact. While the causes can vary depending on the language and testing framework, common factors include asynchronous behavior, concurrency issues, external dependencies, and improper test isolation.

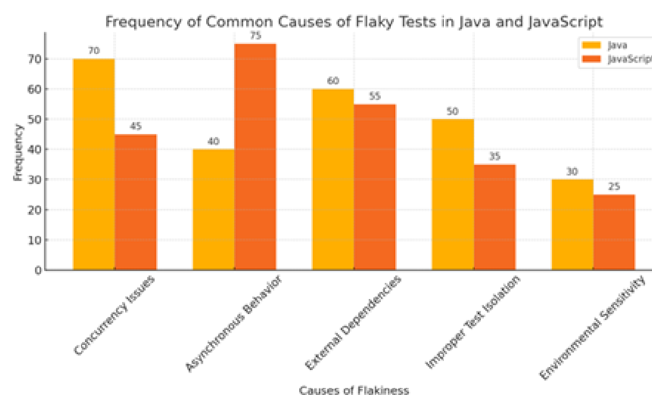

Fig. 2 *Frequency of common causes of flaky tests in both Java and JavaScript.*

Java:

In Java, many flaky tests arise due to concurrency issues, improper test isolation, and reliance on static variables. Java's multithreaded architecture, while powerful, introduces the potential for race conditions, where tests fail due to thread scheduling variability rather than actual bugs.

- **Concurrency issues:** Tests that involve multithreading are prone to flakiness because thread execution order can vary between test runs, leading to inconsistent outcomes.

• **Static variables:** Java's use of static variables can introduce state-sharing between tests, which may cause side effects and lead to flaky behavior.
• **Dependency on external systems:** Tests that rely on external systems, such as databases or external APIs, are vulnerable to network-related flakiness. Even slight network delays or unavailability can result in test failures that are unrelated to the functionality being tested.

JavaScript:

JavaScript's asynchronous nature introduces a different set of challenges for test reliability. JavaScript's asynchronous behavior using callbacks, promises, and async/await patterns, often leads to race conditions and flaky tests if not managed properly.
• **Asynchronous execution:** Tests in JavaScript often rely on promises or callbacks, which can lead to flaky behavior if the code is not correctly synchronized. Failing to manage the order in which asynchronous operations complete can cause tests to pass or fail inconsistently.
• **DOM-related flakiness:** In web-based JavaScript applications, tests that interact with the DOM can fail due to timing issues. For example, tests may attempt to access elements that haven't yet been rendered, leading to intermittent failures.
• **External service dependencies:** As with Java, tests in JavaScript that depend on external APIs or network calls are susceptible to flaky behavior due to latency or other network issues.
Given the prevalence of these issues, retry mechanisms and test selection strategies are crucial tools for managing flaky tests.

RETRY MECHANISMS

Introduction to Retry Mechanisms:

Retry mechanisms involve rerunning tests that fail to differentiate between legitimate failures and nondeterministic flakiness. This approach helps reduce the noise introduced by flaky tests by allowing transient failures to be retried before marking the test as failed.
Retry mechanisms have gained popularity in CI/CD pipelines as a means of mitigating flaky tests. Rather than immediately failing a build when a test fails, retry mechanisms rerun the failed tests a specified number of times to determine if the failure was transient.

Retry Mechanisms in Java:

In Java, retry mechanisms are typically implemented using features provided by the testing framework (such as JUnit or TestNG) or by integrating external plugins in CI/CD tools like Jenkins. Some common approaches include:
• **TestNG Retry Analyzer:** TestNG provides built-in support for retrying failed tests through the "IRetryAnalyzer" interface. Developers can configure this interface to specify how many times a test should be retried before it is considered truly failed. By rerunning flaky tests, developers can reduce false negatives and focus on legitimate issues.
• **JUnit Custom Retry Logic:** In JUnit, retry mechanisms can be implemented through custom annotations or rules that allow failed tests to be automatically retried. For example, a retry rule can be created to rerun tests that fail due to known flaky behavior, such as network-related issues.
• **Jenkins Flaky Test Plugin:** Jenkins, a popular CI tool, offers plugins such as the "Flaky Test Handler" that rerun failed tests in Java projects. This plugin retries the tests multiple times and only reports a failure if the test consistently fails.
These retry mechanisms help mitigate the impact of flaky tests in Java projects by allowing transient failures to be resolved automatically, without requiring manual intervention from developers.

Retry Mechanisms in JavaScript:

In JavaScript, retry mechanisms are supported by popular testing frameworks like Mocha, Jest, and Cypress, which allow failed tests to be automatically retried. Some of the key approaches include:
• **Mocha Retry:** Mocha, a widely used testing framework for Node.js applications, offers a --retries option that allows developers to specify how many times a test should be retried before it is marked as failed. This feature is particularly useful for tests that involve asynchronous operations or external dependencies.
• **Jest Retry:** Jest, a popular testing framework for JavaScript and React applications, provides built-in support for retrying flaky tests using the "retryTimes" option. This option allows developers to retry tests that fail due to asynchronous behavior or timing issues, improving the stability of the test suite.
• **Cypress Retry Logic:** Cypress, a testing framework designed for modern web applications, incorporates retry mechanisms within its architecture. By default, Cypress retries assertions and commands to account for asynchronous behavior in the browser. This retry logic helps mitigate flakiness in tests that interact with the DOM or make network requests.
By leveraging retry mechanisms in JavaScript, development teams can improve the reliability of their test suites and reduce the impact of flaky tests in web applications.

Pros and Cons of Retry Mechanisms:
Retry mechanisms offer several benefits in managing flaky tests, but they also have limitations.
Advantages:
• **Reduced false negatives:** By rerunning failed tests, retry mechanisms help distinguish between legitimate failures and transient, flaky behavior. This reduces the number of false negatives in the test suite, improving developer productivity.
• **Improved CI/CD stability:** Retry mechanisms help ensure that CI/CD pipelines are not interrupted by flaky tests, allowing the pipeline to continue even if transient failures occur.
Disadvantages:
• **Masking of underlying issues:** While retry mechanisms can mitigate the impact of flaky tests, they do not address the root cause of the flakiness. Over-relying on retries can mask real issues in the test suite, leading to technical debt.
• **Increased execution time:** Retrying tests adds additional execution time to the test suite, which can slow down the CI/CD pipeline, particularly in large projects with many tests.
Given these trade-offs, retry mechanisms should be used judiciously in conjunction with other strategies, such as smart test selection, to effectively manage flaky tests.

SMART TEST SELECTION

The Concept of Smart Test Selection:
Smart test selection is an optimization technique that involves selecting only the most relevant tests to run based on recent code changes. Rather than executing the entire test suite for every code change, smart test selection focuses on running tests that are most likely to be impacted by the changes.
This approach reduces the number of tests that need to be run, improving the efficiency of the testing process while maintaining test coverage. Techniques such as test impact analysis, dependency tracking, and historical test data can be used to inform smart test selection.

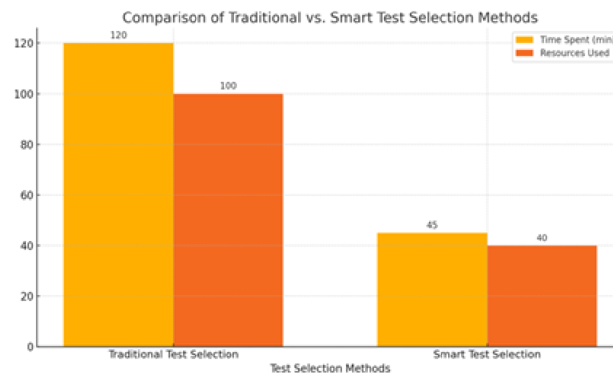

Fig. 3 *Comparison of Traditional vs. Smart Test Selection Methods*

Implementing Smart Test Selection in Java:
In Java, smart test selection can be implemented using a variety of tools and techniques:
• **Ekstazi:** Ekstazi is a tool that monitors dependencies between tests and the codebase, allowing developers to run only the tests that are affected by recent code changes. By avoiding rerunning tests that are not impacted, Ekstazi speeds up the testing process and reduces the likelihood of encountering flaky tests.
• **Surefire Plugin:** The Surefire plugin for Maven provides features for selective test execution based on historical test results and code coverage data. By analyzing which parts of the codebase have been modified, Surefire can run only the tests that are most likely to be affected by the changes.
• **Test Impact Analysis:** Test impact analysis is a technique that tracks the dependencies between code and tests, allowing developers to run only the relevant tests based on recent changes. This approach can be implemented using custom scripts or integrated with existing tools like Jenkins or GitLab CI.
By using smart test selection, Java projects can significantly reduce test execution time, allowing developers to focus on the most critical tests while minimizing the impact of flaky tests.
Implementing Smart Test Selection in JavaScript:
In JavaScript, smart test selection is supported by testing frameworks such as Jest and Mocha:
• **Jest Watch Mode:** Jest's "watch mode" monitors file changes in real-time and only reruns the tests that are directly affected by those changes. This feature reduces the number of unnecessary test executions, improving the efficiency of the testing process.

• **Mocha and Gulp Integration:** In Mocha, smart test selection can be implemented using tools like Gulp or custom scripts that track file changes and run only the relevant tests. By integrating Mocha with Gulp, developers can optimize the test execution process for large-scale JavaScript applications.
• **Selective Test Execution with ts-jest:** For projects using TypeScript, ts-jest provides features for selective test execution based on file changes. By focusing on the tests that are impacted by recent modifications, ts-jest improves the efficiency of the test suite and reduces the likelihood of encountering flaky tests.
Smart test selection is particularly useful in large JavaScript projects, where running the entire test suite for every code change can be time-consuming and inefficient.

Benefits of Smart Test Selection:

Smart test selection offers several advantages for managing flaky tests:
• **Reduced execution time:** By running only the relevant tests, smart test selection significantly reduces the time required to execute the test suite. This allows developers to iterate more quickly and respond to code changes faster.
• **Improved test reliability:** Smart test selection minimizes the likelihood of encountering flaky tests by focusing on the tests that are most relevant to recent code changes. This reduces the noise in the test suite and improves the overall reliability of the testing process.
• **Optimized CI/CD pipelines:** By reducing the number of tests that need to be executed, smart test selection helps optimize CI/CD pipelines, allowing teams to deliver code changes more quickly without sacrificing test coverage.

ADDRESSING FLAKY TESTS HOLISTICALLY

Combining Retry Mechanisms with Smart Test Selection:

While retry mechanisms and smart test selection are valuable techniques for managing flaky tests, they are most effective when used together. By combining these approaches, development teams can achieve the following benefits:
• **Reduced test flakiness:** Retry mechanisms help mitigate the impact of flaky tests by rerunning tests that fail due to transient issues. Smart test selection reduces the likelihood of encountering flaky tests by focusing on the most relevant tests.
• **Optimized test execution:** By reducing the number of unnecessary test executions and rerunning only the tests that fail intermittently, teams can optimize their test suites and improve the efficiency of their CI/CD pipelines.
Combining retry mechanisms with smart test selection provides a holistic approach to managing flaky tests, allowing teams to maintain reliable test suites while minimizing the impact of intermittent failures.

Proactive Flaky Test Detection:

In addition to retry mechanisms and smart test selection, proactive detection of flaky tests is essential for maintaining a reliable test suite. Tools and techniques for identifying flaky tests include:
• **FlakeFinder:** FlakeFinder is a tool for detecting flaky tests in Java projects. It runs tests multiple times and identifies those that produce inconsistent results, allowing developers to prioritize fixing flaky tests.
• **Jest Flaky Test Detection:** Jest, a popular testing framework for JavaScript, provides built-in features for detecting flaky tests. By running tests multiple times and tracking their consistency, Jest helps developers identify flaky tests and address them proactively.
Proactive detection of flaky tests allows teams to address the root causes of flakiness before they become a significant problem in the CI/CD pipeline.

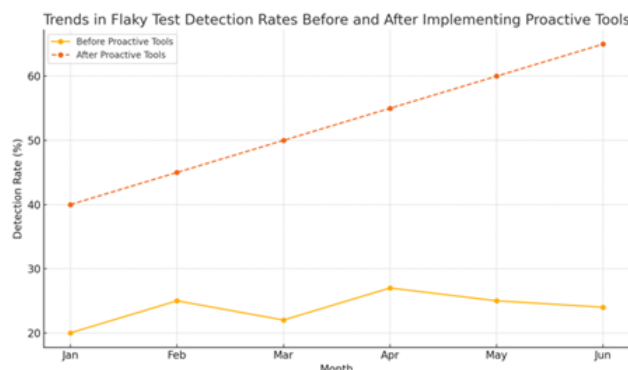

Fig. 4 *Trends in Flaky Test Detection Rates Before and After Implementing Proactive Tools*

Best Practices for Mitigating Flakiness:

To effectively manage flaky tests, development teams should follow best practices for test design and execution:

• **Isolation of tests:** Ensuring that tests are properly isolated from each other and from external systems can reduce the likelihood of flaky behavior. Tests should avoid relying on shared state or external dependencies whenever possible.
• **Proper management of timeouts:** Many flaky tests are caused by improper management of timeouts, particularly in JavaScript applications that involve asynchronous code. By carefully setting timeout values and ensuring that tests wait for asynchronous operations to complete, developers can reduce the likelihood of encountering flaky tests.
• **Systematic test design:** Following good test design practices, such as avoiding shared state and using dependency injection, can reduce the likelihood of flaky behavior. In Java, tests that involve multithreading should be carefully designed to avoid race conditions and other concurrency issues.

CONCLUSION

Flaky tests are a persistent challenge in software testing, especially in fast-paced CI/CD environments. By combining retry mechanisms with smart test selection, development teams can significantly reduce the negative impact of flaky tests while optimizing their test suites. Though these techniques do not prevent flakiness, they provide a practical approach for managing its effects. Moving forward, proactive identification and fixing of flaky tests, along with innovative test design principles, will be essential to maintaining robust and reliable test suites.
By addressing flaky tests through a combination of techniques, development teams can build more reliable software, reduce the time spent diagnosing test failures, and improve the efficiency of their CI/CD pipelines. The key to success lies in adopting a holistic approach that incorporates both preventive and reactive strategies for managing flaky tests.

REFERENCES

[1]. Luo, Q., et al. "An empirical analysis of flaky tests." Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14), 2014. Available: https://dl.acm.org/doi/abs/10.1145/2635868.2635920
[2]. Eck, M., Palomba, F., Castelluccio, M., & Bacchelli, 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
[3]. Retrying Tests in Jenkins. Flaky Test Retry Plugin for Jenkins Pipelines. Available at: https://plugins.jenkins.io/flaky-test-handler/
[4]. Retry Mechanisms in JUnit. JUnit Retry Annotations for Handling Flaky Tests. Available at: https://junit.org/junit5/docs/current/user-guide/#writing-tests-repeated-tests
[5]. Parry, O., Kapfhammer, G., Hilton, M., & McMinn, P. (2022). A Survey of Flaky Tests. ACM Trans. Softw. Eng. Methodol., 31, 17:1-17:74. https://doi.org/10.1145/3476105.
[6]. Romano, A., Song, Z., Grandhi, S., Yang, W., & Wang, W. (2021). An Empirical Analysis of UI-Based Flaky Tests. 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 1585-1597. https://doi.org/10.1109/ICSE43902.2021.00141
[7]. Zolfaghari, B., Parizi, R., Srivastava, G., & Hailemariam, Y. (2020). Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. Software: Practice and Experience, 51, 851 - 867. https://doi.org/10.1002/spe.2929
[8]. TestNG Documentation. Available: https://testng.org/doc/documentation-main.html
[9]. Cypress.io, "Best Practices for Writing Tests." Available: https://docs.cypress.io/guides/references/best-practices
[10]. Shi, A., Zhao, P., & Marinov, D. (2019). Understanding and Improving Regression Test Selection in Continuous Integration. 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), 228-238. https://doi.org/10.1109/ISSRE.2019.00031
[11]. Lam, W., Godefroid, P., Nath, S., Santhiar, A., & Thummalapenta, S. (2019). Root causing flaky tests in a large-scale industrial setting. Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. https://doi.org/10.1145/3293882.3330570