Identifying and Mitigating Flaky Tests in JavaScript

A thesis presented in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

Massey University
Palmerston North, New Zealand.

Negar Hashemi

2026

In memory of those who lost their lives in the pursuit of freedom in Iran.

Abstract

Flaky tests, tests that may pass in one execution and fail in another without changes to the test code or the code under test, pose a persistent challenge to software development by reducing confidence in test results and increasing maintenance costs. Although flaky tests have been studied in several programming ecosystems, their causes, manifestations, and mitigation strategies in JavaScript remain less well understood, despite the language's widespread use and highly heterogeneous execution environments.

This thesis investigates flaky tests in JavaScript through an empirical paradigm, focusing on three complementary objectives: understanding the main causes of flakiness in practice, detecting order-dependent flaky tests, and controlling flaky tests caused by environmental dependency. To address these objectives, the thesis combines large-scale mining of open-source repositories, systematic experimentation, and tool development.

First, the thesis presents an empirical study of flaky-test-related commits collected from popular JavaScript projects hosted on GitHub. By manually analysing commit messages, code changes, and associated issue discussions, the study identifies the dominant causes of flaky tests and examines how developers respond to them. The results show that flakiness in JavaScript is primarily driven by concurrency, asynchronous waiting, operating system dependencies, and network-related issues. Together, these causes account for more than 70% of the observed flaky-test commits. The study further shows that developers usually address flaky tests through direct fixes, while skipping, quarantining, or removing tests are used as pragmatic alternatives when immediate fixes are difficult.

Second, the thesis investigates order-dependent flakiness, where test outcomes depend on the execution order of tests. Building on the empirical findings, it introduces a systematic test reordering approach and implements it in a tool called JS-TOD for Jest-based projects. JS-TOD reorders tests at multiple structural levels and reruns them to expose order-dependent behaviour. An evaluation on real-world projects shows that order-dependent flakiness does occur in JavaScript, although it is less prevalent than reported in other ecosystems. The detected cases are primarily caused by shared state between tests, including shared files and shared mocking state, with the latter identified as an important and previously underreported source of order dependency in JavaScript.

Finally, the thesis examines environmental flakiness, which arises when tests fail under specific execution environments such as particular operating systems, Node.js versions, or browsers. Through systematic reruns across multiple environments, the study demonstrates

that environmental flakiness is common and diverse in JavaScript projects. To mitigate its impact, the thesis proposes a lightweight annotation-based approach, implemented as a tool `js-env-sanitizer`, that sanitizes environment-dependent tests by conditionally skipping and reporting them under known problematic configurations. An empirical evaluation shows that `js-env-sanitizer` can effectively control environmental flaky tests without introducing incorrect annotations.

Overall, this thesis provides an evidence-based understanding of flaky tests in JavaScript and demonstrates practical techniques for detecting and controlling two important classes of flakiness. The findings highlight the need for ecosystem-specific analysis and offer actionable guidance for improving the reliability of automated testing in JavaScript projects.

Acknowledgements

First and foremost, I would like to thank my primary supervisor, Dr. Amjed Tahir. I consider myself very fortunate to have had his guidance throughout my doctoral studies. He has consistently been generous with his time and support, and his advice and feedback have been instrumental in shaping both this research and my development as a researcher. I am sincerely grateful for his mentorship.

I am grateful to the members of my supervisory team, Dr. Shawn Rasheed and Dr. Rachel Blagojevic, for their valuable feedback, discussions, and support during the course of this research. I also thank Dr. August Shi for his collaboration and technical insights, which contributed to the development and evaluation of parts of this work.

I acknowledge the support of Massey University for providing the academic environment and resources necessary to conduct this research, as well as the financial support provided through the Massey University Doctoral Scholarship and associated research and travel grants.

Finally, I would like to express my deepest gratitude to my family. I am especially thankful to my mother and brothers for their constant love, patience, and encouragement, and to my late father, whose belief in the value of education continues to inspire me. I am profoundly grateful to my husband, Mohammad, for his unwavering support, understanding, and encouragement throughout this journey. I also thank my daughter, Mania, whose love, curiosity, and joy have been a constant source of motivation and strength during my doctoral studies.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

AST Abstract Syntax Tree

CD Continuous Deployment

CI Continuous Integration

CUT Code Under Test

FP False Positive

FN False Negative

GH GitHub

GHA GitHub Actions

JS JavaScript

JS-TOD JavaScript Test Order Dependency Detector

LOC Line of Code

npm Node Package Manager

OD Order-Dependent

ODT Order-Dependent Test

OS Operating System

RQ Research Question

TS Test Suite

UI User Interface

Chapter 1

Introduction

1.1 Overview

Society and individuals rely heavily on software for many of their activities. From the applications on our smartphones and the navigation systems in our vehicles to the medical devices that save lives, software powers countless aspects of society. As it becomes increasingly pervasive, the demand for high-quality, reliable, and safe software grows accordingly. In practice, software may contain bugs that can lead to catastrophic failures [1]. Examples include the 1996 Ariane 5 rocket explosion, caused by an arithmetic overflow that led to the rocket self-destructing only 36 seconds after launch, resulting in an estimated loss of 0.37 billion USD [2]. Another major incident was the 2012 Knight Capital trading glitch, where a software deployment error triggered errant trades and caused losses exceeding USD 440 million within an hour [3]. More recently, in September 2025, the Australian telecom firm Optus experienced a firewall upgrade failure that blocked emergency "000" calls for more than 12 hours; as a result, around 600 calls failed, and four deaths were reported among those unable to reach emergency services during the outage [4].

Software testing plays a key role in the broader process of software quality assurance. It involves executing software under controlled conditions to verify whether it behaves as expected, satisfies its requirements, and performs reliably in typical usage scenarios. Although developers use a range of techniques to ensure software quality, testing remains one of the primary means of identifying and diagnosing defects. Many testing approaches follow systematic procedures to validate software functionality, while others are exploratory or adaptive in nature. By detecting issues before deployment, testing helps reduce the likelihood of costly post-release failures, though it cannot completely prevent them. However,

testing can constitute a significant portion of software development costs, often ranging from 20 % to 40 % of the total budget [5].

Software testing encompasses a wide range of activities, including unit, integration, system, and acceptance testing, each targeting different aspects of software quality. Within this range, regression testing plays a central role. It involves rerunning existing tests after new code is added or changes are made to ensure that functionality is preserved. Regression testing is therefore a cornerstone of modern software development [6], particularly in environments practicing continuous integration and deployment, where software changes are frequent.

A key assumption of regression testing is that test outcomes are deterministic: when developers submit changes, rerunning the tests should reveal failures directly related to those changes. Unfortunately, this assumption does not always hold. When tests produce inconsistent outcomes without corresponding changes to the code, this behaviour is commonly referred to as test flakiness.

A flaky test is a test that may pass in one execution and fail in another, even though neither the test code nor the code under test has changed [7].

Figure 1.1 illustrates this behaviour. Test flakiness is particularly damaging in regression testing because regression testing relies on historical comparison and change attribution to identify newly introduced faults. Some tests exhibit non-deterministic outcomes; they may pass or fail intermittently when run multiple times on the same code. These so-called flaky tests produce failures unrelated to the actual changes, but rather to some underlying non-deterministic behaviour. This can arise from issues within the test itself (e.g., timing, concurrency, or improper setup) or in the code under test, where shared files or asynchronous execution can lead to inconsistent results. Such behaviour undermines developers' trust in their builds and often leads them to ignore test failures altogether [8]. However, ignoring flaky test failures during a build has been shown to increase the likelihood of crashes in deployed systems [9].

Flaky tests are widespread in practice. For example, Google engineers identified test flakiness as a major challenge in automated testing, highlighting its impact on developer productivity, release velocity, and confidence in test automation [10]. Meta (Facebook) has also introduced a Probabilistic Flakiness Score (PFS) to quantify and monitor flakiness across its vast CI ecosystem, helping engineers distinguish between genuine test failures and non-deterministic behavior [11]. This industrial focus reflects a growing trend among

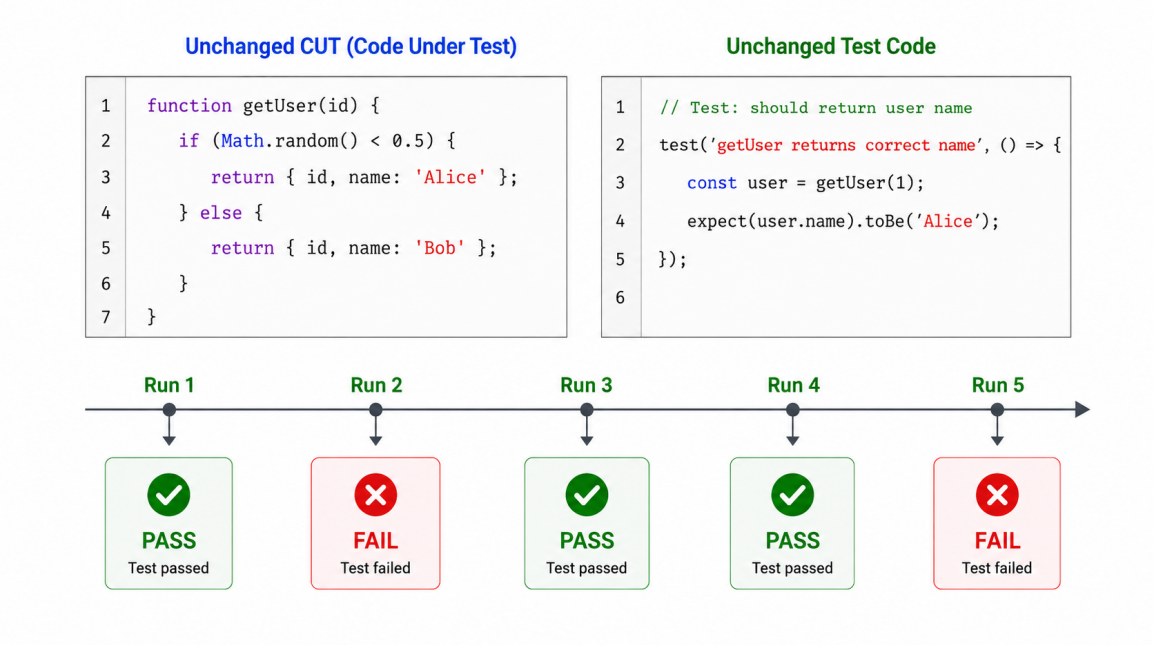

Figure 1.1: An illustration of a flaky test that passes in one execution and fails in another without any changes to the test code or the code under test.

large technology companies to treat flakiness as a measurable and controllable engineering problem rather than an incidental inconvenience. Recent industrial studies show that developers spend about 1.28% of their working time diagnosing and repairing flaky tests [12], which corresponds to an estimated monthly cost of around USD 2,250 per project affected by flakiness [13].

The consequences of flaky tests are significant. Flaky tests cause intermittent build breaks and uncertainty about whether a failing test signals a real fault. This ambiguity hinders corrective actions, negatively affects product quality, and delays delivery [10,14,15]. Developers may spend hours investigating failures that turn out to be false alarms, inflating project costs. For instance, in the Microsoft Dynamics product line in 2015, verifying whether a failure was due to flakiness was estimated to cost up to USD 7.2 million per year [16].

Beyond direct debugging overhead, flakiness also disrupts testing-related activities. It reduces the effectiveness of test suite minimization, a process that selects the smallest subset of tests needed to maintain coverage, because flaky results make it difficult to determine which tests are truly redundant [17]. Similarly, it interferes with test parallelization, where

tests are executed concurrently to speed up continuous integration (CI) pipelines, since unpredictable outcomes can cause false dependencies or masking effects [18]. Moreover, research techniques that rely on test results, such as fault localization, which identifies likely fault locations in source code [19], or automated program repair, which generates candidate patches based on failing tests [20], are also negatively affected by flaky tests.

1.2 Problem Statement

Flaky tests have become a major concern in both research and practice due to their significant impact on software quality and development productivity. They cause uncertainty in test results, waste developer time, and reduce confidence in automated testing and CI pipelines. As a result, a growing body of work has focused on understanding, detecting, and mitigating flaky tests. However, based on a review study we conducted (Chapter 2), most existing flaky test studies have concentrated on Java programs, with only a few examining other languages such as Python. This leaves a gap in understanding how flakiness manifests in JavaScript, one of the most widely used languages for web, mobile, and server-side applications. The language's dynamic and asynchronous characteristics, combined with its highly heterogeneous execution environments (e.g., browsers, Node.js, and CI pipelines), may lead to different patterns and causes of flakiness than those observed in statically typed languages. Addressing this problem is crucial to improving test reliability and CI stability in the JavaScript ecosystem, ultimately contributing to higher-quality, more dependable software systems.

To the best of our knowledge, only two studies have examined flaky tests across multiple programming languages, including JavaScript [21,22]. Despite its popularity and widespread use, flakiness in JavaScript remains underexplored. JavaScript has consistently ranked among the most widely used programming languages[1] and is applied across diverse domains, including web (both client and server-side), mobile, and the Internet of Things (IoT) [23, 24]. Its extensive reliance on asynchronous APIs and an event-driven execution model makes it especially susceptible to races. Asynchronous and concurrency issues are already known sources of flakiness in other languages, notably Java and Python [7,8]. Furthermore, JavaScript applications often run across different platforms (e.g., operating systems and browsers). Since platform dependencies have been identified as major causes of flakiness [8, 25], such issues are also expected in JavaScript programs.

Previous studies on test flakiness in other languages and its causes largely focus on

[1] `https://survey.stackoverflow.co/2024/`

specific sources, such as order-dependency [26], concurrency [27], or UI-related flakiness [21, 28] (discussed further in Section 2.2.1). However, little attention has been given to the role of environmental variations (e.g., operating systems and browsers [29]), even though we found these to be a major cause of flakiness in JavaScript projects (Chapter 4).

In addition, there is a lack of techniques and tools for addressing flaky tests in JavaScript. While several tools have been developed to detect and analyse flakiness in languages like Java and Python, similar support for JavaScript is limited. A survey of recent literature on flaky tests [30, 31] identified only two tools designed to manifest flakiness in JavaScript [32, 33], leaving the ecosystem under-supported compared to other languages. Beyond the lack of tooling, there's a broader need for systematic approaches and empirical insights to help developers identify, explain, and reduce flaky behaviour in JavaScript tests. Filling this gap is crucial for improving the reliability of automated testing and ensuring that continuous integration workflows in JavaScript projects remain dependable and efficient.

Given these gaps, this research investigated the causes of flaky tests in JavaScript by mining real-world open-source projects hosted on GitHub. It also examined the strategies developers use to respond to identified flaky tests. Building on these insights, the research aimed to develop and evaluate practical tools to detect and mitigate flakiness in JavaScript programs, with particular emphasis on causes that remain underexplored in the literature. The overall aim of this research is to improve the reliability of automated testing in JavaScript by deepening the understanding of test flakiness and developing practical techniques to address its most impactful and underexplored causes. Therefore, the objectives for this research are:

1. **Understanding flaky tests in JavaScript projects**: Investigate causes of flaky tests and developers' responses to flakiness in real-world JavaScript projects.

2. **Detecting flaky tests in JavaScript**: Based on the results of (1), we identify one of the causes of flakiness (i.e., test order dependency) and then develop a practical technique that aims to detect and manifest flaky tests.

3. **Controlling flaky tests in JavaScript**: Based on (1), we identify another major cause of test flakiness in JavaScript (i.e., environmental dependency) and then design a technique to sanitize potential flaky tests from regression testing.

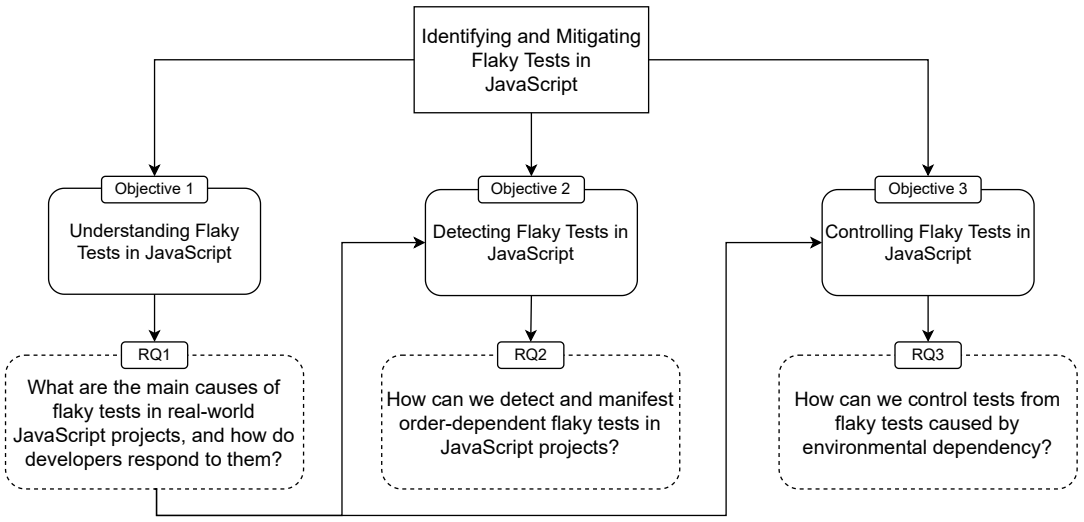

Figure 1.2: Research objectives and their corresponding research questions in this thesis

1.3 Research Questions

Building on the gaps and objectives identified in Section 1.2, this research is guided by the following three main research questions:

RQ1: What are the main causes of flaky tests in real-world JavaScript projects, and how do developers respond to them?

RQ2: How can we detect and manifest order-dependent flaky tests in JavaScript projects?

RQ3: How can we control tests from flaky tests caused by environmental dependency?

Figure 1.2 illustrates how the research objectives are connected with the research questions. The first objective (RQ1) provides an empirical foundation by uncovering the main causes of flaky tests and how developers address them. These insights shape the subsequent directions of this thesis. The second objective (RQ2) focuses on detecting and manifesting order-dependent flakiness. The third objective (RQ3) addresses environmental flakiness by designing techniques to sanitize affected tests. Together, these research questions ensure that the objectives are achieved in a structured and complementary manner.

Addressing RQ1: To investigate the main causes of flakiness in JavaScript programs, we conducted a large-scale empirical analysis of the top JavaScript projects on GitHub.

These projects provide a sufficiently large and diverse sample of JavaScript tests (in terms of commits). For each commit, we examined the commit message, the code changes, and any linked issues in the issue tracker. In addition to identifying the causes of flakiness, we analysed the strategies developers employed when dealing with flaky tests, such as fixing, skipping, or ignoring them. This provided deeper insight into how JavaScript developers perceive and respond to test flakiness.

Addressing RQ2: Building on the findings from RQ1, and comparing them with prior results from other languages such as Java and Python, we extended our study to focus on order-dependent tests in JavaScript. Specifically, we investigated the prevalence of order-dependency in JavaScript projects. We implemented a systematic approach to randomize the execution order of tests and observed changes in their outcomes.

Addressing RQ3: Motivated by the results of our empirical study, we conducted a systematic evaluation of the impact of environmental factors on test flakiness in JavaScript. We executed test suites across multiple environments to observe whether environmental changes trigger flaky behaviour. To mitigate such issues, we designed a lightweight annotation-based approach to sanitize environment-related flaky tests.

1.4 Research Contributions

The main contributions of this thesis are as follows:

- **Empirical analysis of flaky tests in JavaScript.** We identified the primary causes of flaky tests in real-world projects and documented the common strategies developers use to address them.
- **A systematic evaluation and detection of order-dependent tests.** We analysed the main causes of order-dependent tests in JavaScript and developed a systematic approach for detecting them by reordering tests and test suites.
- **Analysis and mitigation of environment-dependent tests.** We investigated the causes of environmental flakiness in JavaScript and designed a lightweight annotation-based approach to sanitize tests against environmental factors.

In addition to the empirical findings and methodological contributions described throughout this thesis, this work produced a set of reusable tools and datasets to support future research on flaky tests in JavaScript:

- **FlakyJS Dataset**: A curated dataset of JavaScript projects exhibiting test flakiness, collected from open-source repositories on GitHub. Link: `https://zenodo.org/records/6757825`
- **OD-Test Dataset**: A dataset of order-dependent tests identified from open-source JavaScript projects through systematic test reordering and rerunning. Link: `https://zenodo.org/records/13852085`
- **JS-TOD**: An automated tool for revealing test order dependency in Jest-based projects through controlled test reordering and repeated execution. Link: `https://github.com/Negar-Hashemi/JS-TOD`
- **js-env-sanitizer**: A Babel-based tool for mitigating environment-related flaky tests by conditionally skipping and reporting tests under known problematic configurations. Link: `https://github.com/Negar-Hashemi/js-env-sanitizer`
- **Env-Flaky Dataset**: A dataset of environment-dependent flaky tests in JavaScript projects, collected from open-source repositories and analysed across multiple execution environments. Link: `https://github.com/Negar-Hashemi/env-flakiness-js`

1.5 List of Publications

This research has resulted in several publications, some of which have been accepted and others are currently under review. At the time of this thesis submission, five papers have been published or remain under review across five different research venues, as follows:

- Chapter 4: **Hashemi, N.**, Tahir, A., & Rasheed, S. (2022, October). An empirical study of flaky tests in JavaScript. In 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE.
- Chapter 2: A. Tahir, S. Rasheed, J. Dietrich, **N. Hashemi**, and L. Zhang, "Test flakiness' causes, detection, impact and responses: A multivocal review", Journal of Systems and Software 206 (2023).

  I contributed to part of the data analysis and to the writing and reviewing of the manuscript. Parts of this work are included in the literature review in Chapter 2.
- Chapter 5: **Hashemi, N.**, Tahir, A., Rasheed, S., Shi, A., & Blagojevic, R. (2025, March). Detecting and evaluating order-dependent flaky tests in JavaScript. In 2025 IEEE Conference on Software Testing, Verification and Validation (ICST). IEEE

- Chapter 5: **Hashemi, N.**, Tahir, A., Rasheed, S., Shi, A., & Blagojevic, R. (2025). JS-TOD: Detecting Order-Dependent Flaky Tests in Jest. Science of Computer Programming, Elsevier. (tool paper - under review, second revision)
- Chapter 6: **Hashemi, N.**, Tahir, A., Shi, A., Rasheed, S. & Blagojevic, R. (2025). A Systematic Evaluation of Environmental Flakiness in JavaScript Tests. In 2026 IEEE Conference on Software Testing, Verification and Validation (ICST). (under review)

1.6 Thesis Outline

The rest of the thesis is organized as follows:

- **Chapter 2 Literature Review:** Presents the background for the research and provides an overview of related work on test flakiness, its causes, and existing detection and mitigation approaches.
- **Chapter 3 Research Methodology:** Describes the overall research design, data collection process, and experimental settings used in the studies.
- **Chapter 4 Understanding Flaky Tests in JavaScript:** Presents the methodology and the results received for the empirical study conducted to investigate the causes of and responses to test flakiness in JavaScript.
- **Chapter 5 Detecting and Evaluating Order-dependent Tests:** Presents the methodology and results for the study on order-dependent tests in JavaScript.
- **Chapter 6 Understanding Environment-dependent Tests:** Presents the methodology and results for the study on environmental flakiness in JavaScript.
- **Chapter 7 Discussion:** Interprets and synthesizes the results across all studies, highlights the broader implications for researchers and practitioners, and discusses the limitations of the work.
- **Chapter 8 Conclusion:** Summarizes the findings of the studies, potential future research, and concludes the thesis.

MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
DOCTORATE WITH PUBLICATIONS/MANUSCRIPTS

We, the student and the student's main supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the student's contribution as indicated below in the Statement of Originality.	
Student name:	Negar Hashemi
Name and title of main supervisor:	Associate Professor Amjed Tahir
In which chapter is the manuscript/published work?	Chapter 2 Literature Review (partial inclusion)
Describe the contribution that the student and members of the supervisory team have made to the manuscript/published work:[1] The student contributed to the data analysis, synthesis of results, and writing and reviewing of the paper. The supervisors and co-authors contributed to the overall design of the study, data collection, analysis, and writing of the paper. Student (Negar Hashemi): 20% Supervisors and co-authors: 80%	
Please select one of the following three options:	
◉	The manuscript/published work is published or in press Please provide the full reference of the research output: Tahir, A., Rasheed, S., Dietrich, J., Hashemi, N., & Zhang, L. (2023). Test flakiness' causes, detection, impact and responses: A multivocal review. Journal of Systems and Software, 206, 111837.
○	The manuscript is currently under review for publication Please provide the name of the journal:
○	It is intended that the manuscript will be published, but it has not yet been submitted to a journal

Student's signature:	Negar Hashemi Digitally signed by Negar Hashemi Date: 2026.01.26 23:05:49 +13'00'	Main supervisor's signature:	Amjed Tahir Amjed Tahir 2026.01.29 07:48:27 +03'00'

This form should be placed at the beginning of each relevant thesis chapter.

[1] Refer to the Massey University Publishing and Authorship guidelines (OneMassey for staff, Stream for students) and/ or Contributor Roles Taxonomy (CRediT) guidelines for guidance.

Chapter 2

Literature Review

Software testing is a fundamental activity in software engineering, aimed at assessing the correctness, reliability, and quality of software systems. Over time, a substantial body of research has examined different testing techniques, challenges, and failure modes that arise during testing. One such challenge is test flakiness, where tests exhibit non-deterministic outcomes despite no changes to the underlying code.

For starters, this chapter reviews prior work on software testing to establish the context in which flaky tests arise. It then focuses on flaky tests, examining how they are defined and characterised in the literature, as well as the commonly reported causes of flaky behaviour across software systems.

In addition to causes, this chapter surveys existing mechanisms for detecting flaky tests and the strategies developers use to respond to flakiness once it is identified. Finally, this chapter reviews prior work specifically related to flaky tests in JavaScript projects. This discussion highlights the extent to which existing findings generalise to JavaScript-based systems and identifies limitations in the current literature.

2.1 Software Testing

Software testing is a fundamental activity in the software development lifecycle, ensuring that systems behave as intended and meet user expectations [34–36]. As modern software grows in scale and complexity, testing provides a systematic approach to identifying defects, verifying functionality, and assuring reliability before deployment. Without effective testing, software products risk containing faults that can degrade user experience or lead to critical

system failures. Testing thus acts as both a quality assurance mechanism and a confidence-building process for developers, organizations, and end users.

Early defect detection through testing significantly lowers maintenance costs and improves overall software quality [37]. Beyond defect detection, testing contributes to software verification and validation, two complementary processes that ensure a system's correctness and fitness for purpose [38,39]. Verification confirms that the software correctly implements the required functionality, while validation ensures that it satisfies user needs and intended use cases [40]. Together, these processes reduce uncertainty in software behaviour and form the foundation for trustworthy and maintainable systems.

From an organizational perspective, software testing is not only a technical requirement but also a strategic investment. Effective testing practices contribute to lower maintenance effort, faster release cycles, and enhanced customer satisfaction [41, 42].

Testing is performed at different levels, each targeting a specific scope of functionality within the system. Unit testing focuses on individual components or functions to ensure that each behaves correctly in isolation [34, 39]. Integration testing verifies the interactions between components and checks whether combined units function together as expected [35]. System testing evaluates the complete software system against its requirements in a production-like environment [36], and acceptance testing validates that the system meets user and business needs [38].

Software testing can generally be done manually or automatically, each serving a distinct purpose within the software development process. In manual testing, human testers execute test cases without the use of automation tools. They observe the behaviour of the software and compare the actual results with the expected ones [34, 36]. Manual testing enables exploratory and usability evaluation, where testers' intuition and domain knowledge play an important role. It is particularly useful for assessing user interfaces and identifying issues that are difficult to capture through predefined scripts. However, manual testing is expensive, time-consuming, error-prone, and difficult to maintain as systems grow in size and complexity.

On the other hand, automated testing relies on tools or scripts that execute tests repeatedly and systematically without human involvement [35,39]. It has become an essential component of regression testing and continuous integration practices, where software is frequently updated and must be verified efficiently. Automated tests can run across different configurations and environments, making them highly suitable for maintaining the stability of large and evolving projects [41]. They provide rapid feedback to developers, facilitate early defect detection, and support long-term software quality assurance [43, 44]. In agile

and continuous integration workflows, due to its scalability, automation can ensure that frequent changes do not compromise stability and help sustain confidence in the development process.

Although testing is a key element of modern software development, its effectiveness relies on the reliability of the test outcomes. One of the most serious threats to this reliability is the presence of *flaky tests*, which, as defined in Chapter 1, behave inconsistently and produce different results across repeated executions even when the code or environment has not changed [7, 25, 45]. Such unpredictable behaviour reduces confidence in test results and makes it difficult for developers to determine whether a test failure indicates a genuine fault or a false alarm. This uncertainty disrupts continuous integration processes, increases debugging time, and can even lead to the neglect of failing tests. In the next section, we discuss flaky tests in detail, including their leading causes, impact, and available detection practices.

2.2 Flaky Tests

As defined in Chapter 1, a flaky test is a test that may pass in one execution and fail in another, even though neither the test code nor the program under test has changed. This section reviews how flaky tests have been characterised in previous studies and discusses their causes, impacts, detection mechanisms, and developer responses.

Previous studies, including those from industry, have shown that test flakiness is a widespread and costly problem. For example, a study on the maintenance of regression test suites in GitHub projects found that around 13% of test failures are caused by flakiness [46]. Similarly, Eck et al. [25] examined how developers perceive flaky tests by surveying contributors from 21 Mozilla projects. They analysed 200 flaky tests that the same developers fixed and found that most developers regard flakiness as a serious and recurring issue that negatively affects their productivity and trust in test results.

Flaky tests affect multiple aspects of software development, including automatic test generation [47], test maintenance [48], and overall product quality and reliability [25, 49]. Flaky tests can have a significant impact across different stages of the software lifecycle. Their effects can be broadly classified into three categories: *testing*, *product quality*, and *debugging and maintenance*, as described below.

Effects on testing: Test flakiness challenges the fundamental assumptions of testing, such as determinism and test independence. Automated testing tools can inadvertently generate flaky tests [50], resulting in unreliable feedback that reduces the accuracy of test generation

algorithms and may mask genuine defects. Similarly, test optimization techniques such as test suite reduction, prioritization, and parallelization rely on deterministic outcomes [18, 51, 52]. When these assumptions do not hold, these techniques can introduce or expose flakiness, resulting in misleading regression results and reduced test coverage.

Effects on product quality: The impact on product quality is equally significant. Flaky tests can cause build failures [53], delay continuous integration workflows [54], and weaken overall confidence in software reliability [49]. Because manually identifying flaky tests is time-consuming and often impractical in large projects, researchers have developed automated approaches to detect, classify, and predict flakiness before execution [55–57]. Beyond the technical implications, flaky tests affect developer confidence and perception, often leading to frustration, wasted effort, and resource misallocation [25]. These factors collectively reduce the effectiveness of testing and can result in lower product quality and slower release cycles.

Effects on debugging and maintenance: Flaky tests also disrupt debugging and maintenance activities that depend on stable and reproducible test outcomes. Techniques such as automated program repair, crash reproduction, and fault localization rely on reliable tests to identify and validate faults. When these tests behave inconsistently, their results can mislead automated systems or produce incorrect conclusions. Flaky tests can reduce the accuracy of automated program repair [20, 30], introduce noise into performance regression detection [58], and degrade the precision of fault localization approaches [59].

Over the past decade, the problem of test flakiness has received increasing research attention. Several studies [7, 8, 60] have investigated its causes in both open source and industrial projects, while others have proposed tools and techniques to detect, predict, and mitigate flaky behaviour. As this thesis focuses on the causes of and responses to flakiness, as well as methods for detecting and mitigating flaky tests in JavaScript, this chapter discusses flaky tests in the context of the following areas: (1) causes of test flakiness, (2) detection techniques, (3) developer responses to flaky tests, and (4) test flakiness in JavaScript projects.

2.2.1 Causes of Flaky Tests

Root causes of flaky tests vary in type and nature. Test flakiness can emerge from a wide range of factors related to the test itself, the code under test, or the supporting environment. Table 2.1 summarizes the main causes of flaky tests as identified in previous studies. The order of causes in the first column follows the ranking or relative prevalence reported by the corresponding study, with the most common causes listed first. These studies were

conducted across different programming languages and software domains, including large open-source projects, mobile applications, and machine learning frameworks. The table illustrates that although test flakiness appears in diverse contexts, similar patterns recur across ecosystems.

Luo et al. [7] first classified flaky test causes such as async wait, concurrency, and order dependency. Subsequent studies expanded and refined these categories, adding causes like dependency, UI, timeouts, and platform dependency [8, 25]. Other works highlighted domain-specific and new types of flakiness, such as algorithmic nondeterminism in machine learning [61] and infrastructure flakiness in Python [60].

Overall, previous research shows that flaky tests can arise from a variety of factors, including those related to the tests themselves as well as the code under test. The causes identified in prior studies are organized based on their similarities into the following major categories: *concurrency*, *test order dependency*, *resource leak*, *network*, *time*, *randomness*, *platform dependency*, *hardware*, and *other*. This classification provides a conceptual baseline for investigating flaky tests in JavaScript projects, and each cause is discussed in detail below.

Concurrency. This category is related to flakiness resulting from concurrency-related bugs such as race conditions, data races, atomicity violations, or deadlocks. The major cause of flakiness under concurrency is *Async-wait*. This occurs when an application or test makes an asynchronous call and does not correctly wait for the result to become available before using it. Async-wait is classified under concurrency in Throve et al. [8] and Luo et al. [7]. Similarly, Lam et al. [45] investigated the life cycle of flaky tests in six large-scale Microsoft projects and found that asynchronous calls were the leading cause of flakiness. To mitigate this issue, they proposed an automated technique called Flakiness and Time Balancer (FaTB), which reduces flaky-test failures due to asynchronous calls. Additionally, Luo et al. [7] introduced a further subcategory, termed bug in condition, where the guard condition determining which thread can access a code block is either overly restrictive or overly permissive.

Test order dependency. Tests are intended to be independent, producing the same results regardless of execution order. In practice, however, shared state in memory or external resources (e.g., files, databases) can create hidden dependencies, leading to order-dependent flakiness [18]. Luo et al. [7] found that 12% of flakiness in their study was caused by order dependency. This results from a shared state that can be in either memory or external sources (e.g., file system, database). Several studies have investigated test order dependency across multiple programming ecosystems and environments, including

Table 2.1: Summary of Flaky Test Causes Reported in Prior Studies

Cause of Flakiness	**Language(s)**	**Studies**
Async wait	Java, C++, Python	[7], [25], [60]
Concurrency	Java, Android, C++, Python, Probabilistic (Python, C++)	[7], [8], [25], [61], [60]
Test order dependency	Java, Python	[7], [60]
Resource leak	Java, C++, Python	[7], [25], [60]
Network	Java, Android, Python	[7], [8], [60]
Time	Java, Python	[7], [60]
I/O	Java, Python	[7], [60]
Randomness	Java, C++, Python	[7], [25], [60]
Floating point operations / precision	Java, C++, Probabilistic (Python, C++)	[7], [25], [61]
Unordered collections	Java, Python	[7], [60]
Platform dependency	C++	[25]
Test case timeout	C++, Python	[25], [60]
Dependency / Program logic	Android	[8]
UI	Android	[8]
Algorithmic nondeterminism	Probabilistic (Python, C++)	[61]
Hardware	Probabilistic (Python, C++)	[61]
Incorrect API usage	Probabilistic (Python, C++)	[61]
Infrastructure	Python	[60]

Java [52, 55, 62, 63], Android [64, 65], Python [66], and large-scale industrial settings [56].

Resource leak. Improper management of resources such as memory, database connections, or file handles can lead to nondeterministic failures. Several studies [7, 25, 27, 60] have reported resource leaks as a recurring cause of flakiness.

Network. Tests that rely on network communication are inherently prone to instability due to factors such as latency, connection loss, bandwidth variability, or unavailable remote resources [25, 30]. Gruber et al. [60] identified network issues as the third main cause of flakiness in Python. It is also reported that 14% of the flaky tests found in six large-scale Microsoft projects are due to network-related causes [67]. Flakiness can stem from both local issues (e.g., socket management or port conflicts) and external ones (e.g., remote server failures or API rate limits). Even minor timing differences in response handling can lead to nondeterministic results.

Time. Time-related issues are another common cause of test flakiness. These may arise from midnight transitions in the UTC time zone, daylight saving adjustments, or variations in time precision across different platforms. Lam et al. [55] identified time as one of the root causes of flaky tests, while Eck et al. [25] reported additional subcategories such as test case timeout and test suite timeout, based on feedback from developers. Berglund and Vateman [68] also noted that inconsistencies in time precision across operating systems, platforms, and time zones can contribute to nondeterministic behaviour. More recently, Rahman et al. [69] demonstrated that timing-dependent flakiness remains a major challenge and proposed FlakeRake, an automated approach to reproduce such failures by manipulating thread timing. Flakiness in this category may occur when tests fail to complete within the specified duration or are unable to acquire required resources in time, resulting in intermittent timeouts.

Randomness. Tests that use random inputs or depend on nondeterministic algorithms can fail unpredictably if their assertions do not cover all possible outcomes. This problem is particularly common in probabilistic and machine learning systems, where randomness is integral to model behaviour. Dutta et al. [61] reported algorithmic nondeterminism, incorrect API usage, and hardware effects as typical sources of randomness-related flakiness. Eck et al. [25] also identified overly restrictive assertion ranges, where expected results are too narrowly defined, as a subcategory of this issue.

Platform dependency. Tests may pass on one platform but fail on another due to differences in performance, APIs, or configurations. Eck et al. [25] and Thorve et al. [8] highlighted this issue in Android projects, while Moran et al. [70] analysed environmental variability in web applications. Infrastructure-related flakiness, such as unstable CI environments, has also been reported [60].

Hardware. Specialized hardware can introduce nondeterminism. For example, some ML applications/libraries may use specialized hardware, as discussed in Dutta et al. [61]. This is distinct from platform dependency but similarly difficult to control or through parallel floating-point computations that yield inconsistent results [7, 8].

Test Order Dependency and *Concurrency* have been studied more widely than other noted sources of flakiness. On the other hand, *Platform* or more general *Environment* flakiness gets less attention.

Environmental factors, such as operating systems, external libraries, or containerized runtime environments, can greatly influence the behaviour of tests, often leading to nondeterministic outcomes [29, 71]. Such variability is a common source of test flakiness, as

tests that pass in one environment may fail in another due to subtle differences in system behaviour or resource availability. Developers frequently introduce operating system (OS)-specific tests to handle these variations, performing operations like calling platform-dependent APIs, mocking OS-level objects, or controlling execution flow through timing and suspension. A study by Job et al. [72] revealed that OS-specific tests are often employed to work around missing external resources, unsupported standard libraries, and the presence of flaky tests themselves.

Additional evidence of the environment's influence comes from Berndt et al. [73], who examined SAP HANA's continuous integration pipeline and found that infrastructure characteristics, such as execution environment, test distribution, and runtime scheduling, strongly correlated with flaky behaviour. Although CPU and memory usage had weaker effects, execution time and environmental variability are significant predictors of flakiness.

In the context of web applications, Moran et al. [74] developed *FlakyLoc*, a tool that systematically diagnoses environment-related flakiness by re-executing tests under varied configurations (e.g., memory size, number of CPU cores, browsers, and screen resolutions). Their work illustrates the potential of environment-sensitive testing frameworks to expose nondeterministic behaviour that traditional deterministic test execution often overlooks. Environmental flakiness is discussed in detail in Chapter 6, which focuses on understanding and mitigating environment-dependent test failures in JavaScript.

2.2.2 Flaky Tests Detection Mechanisms

Existing approaches for flaky test detection generally fall into three main categories, each offering a different trade-off between accuracy, cost, and scalability. These three categories are:

- **Static approaches.** Static analysis-based methods analyse the source code of tests or the code under test without executing them [75, 76]. They aim to detect code patterns, smells, or dependencies that are associated with flakiness. For instance, certain API calls, use of asynchronous operations, or reliance on system time may serve as indicators. Static analysis is efficient and can be applied early in the development process, but it may miss environment- or runtime-dependent sources of flakiness that only appear during execution [75].

- **Dynamic approaches.** Dynamic methods detect flakiness through repeated test executions under varied conditions, such as randomized order, environment configurations, or timing constraints [75]. These approaches observe differences in test

outcomes to identify nondeterministic behaviour. While dynamic detection can accurately capture runtime and environment-related flakiness, it often requires substantial computational resources and time, particularly for large test suites [77].

- **Hybrid approaches.** Hybrid methods combine static and dynamic techniques to leverage the advantages of both [75]. Static analysis is first used to identify potentially flaky tests or risky code areas, which are then prioritized for dynamic re-execution. This combination helps improve detection efficiency and accuracy while reducing execution cost.

Table 2.2 summarizes a list of tools found in the literature[1] regarding their approaches and the targeted programming languages. Below is an overview of some of the most used flaky test detection tools.

DeFlaker [79] detects flaky tests by checking the coverage of failed tests against the recent changes. If a test fails but does not cover any changed code, the tool reports the test as flaky without requiring any reruns. Shi et al. [52] also apply code coverage techniques to detect flaky tests for Java-based programs. They propose techniques that manage flakiness throughout the mutation testing process, based on strategically re-running tests. RootFinder detects both the causes and the locations in code responsible for test flakiness. It can identify several categories of flaky behaviour, including network, time, I/O, randomness, floating-point operations, test order dependency, unordered collections, and concurrency. The tool instruments API calls during test execution to log key runtime information, such as time, context, and return values, and may inject additional behaviour, such as delays, to expose concurrency or async wait related issues. After execution, RootFinder analyses the collected logs by evaluating predicates that compare values between passing and failing runs. By identifying predicates that differ between these runs, the tool can highlight the conditions that explain the observed flakiness.

Flash [61] was developed to detect flaky tests due to assertions passing and failing in different runs on the same code. Flash is used to detect and understand non-determinism in Python library code. For this purpose, the authors examined four probabilistic programming systems and machine learning frameworks: Pyro, PyMC3, TensorFlow Probability, and PyTorch. The authors confirmed that projects which depend on probabilistic programming systems and machine learning frameworks have a larger percentage of flaky tests.

[1] Based on our review paper [78]

Table 2.2: Flaky Test Detection Tools in Prior Studies

Approach	Tool / Study	Language	Method	Focus or Technique
Dynamic	iDFlakies [55]	Java	Rerun (vary order)	Detects order-dependent tests via randomized execution
	iFixFlakies [52]	Java	Rerun + delta debugging	Automated repair of order-dependent flaky tests
	DeFlaker [79]	Java	Differential coverage	Identifies new flaky tests through coverage-based analysis
	NonDex [57]	Java	Rerun (vary API implementation)	Detects nondeterminism via controlled randomization
	RootFinder [80]	Java	Log analysis	Identifies root causes of flaky tests
	FlakeScanner [81]	Java	Rerun (vary event schedules)	Detects concurrency-related flakiness
	FlakeShovel [27]	Java	Rerun (vary event schedules)	Identifies concurrency-induced failures
	Shaker [82]	Java	Rerun (add noise to environment)	Detects async and concurrency issues
	PRADET [26]	Java	Rerun (dynamic dataflow analysis)	Practical dynamic detection through repetition
	FLASH [61]	Python	Rerun (vary random number seeds)	Detects probabilistic flakiness in ML systems
	FITTER [83]	Python	Rerun (vary test code)	Identifies flaky tests in CI pipelines
	NodeRacer [32]	JavaScript	Rerun (vary event order)	Detects concurrency and async race conditions
	FlakyLoc [70]	Web	Rerun (vary environment)	Locates environment-induced flakiness in web applications
	Fair [84]	Multi-language	Machine learning	Detects fairness-related flakiness in tests
	FlaPy [66]	Python	Rerun	Detect flaky tests
	iPFlakies [56]	Python	Rerun	Detects and repairs order-dependent flaky tests
Static	FLAST [85]	Java	Machine learning (code similarity)	Learns flaky patterns from static features
	Peeler [86]	Java	Feature extraction (static analysis)	Extracts code features for flaky test prediction
	Determinism Checker [87]	Java	Type checking	Verifies deterministic test behaviour
	PolDet (JPF) [88]	Java	Model checking	Detects policy-based nondeterminism
	Flakify [89]	Cross-domain	Machine learning	Predicts flakiness using code embeddings
	Ahmad et al. [90]	Java	Machine learning	Evaluates static code characteristics correlated with flakiness
Hybrid	FlakeFlagger [91]	Java	Machine learning (hybrid features)	Combines static and dynamic analysis for flaky test prediction

Parry et al. [83] introduced *FITTER* (Flakiness Inducing and Fixing via Test Repair), an approach that leverages Automated Program Repair (APR) techniques to proactively detect and mitigate latent flakiness in test suites. The core idea behind FITTER is to automatically expose and repair potential sources of flakiness before they manifest during regular testing. The approach targets two major root causes of flakiness, test order dependency and resource leaks, by systematically mutating test code and execution environments to reveal nondeterministic behaviours. FITTER monitors the resulting changes in test outcomes and uses these observations to identify tests that are sensitive to shared states or improper cleanup. The technique is implemented and evaluated on Python programs, demonstrating its effectiveness in uncovering and repairing hidden sources of flakiness that may not surface through conventional rerun-based or dynamic detection methods.

Gruber et al. [66] developed *FlaPy*, a tool designed to systematically detect and classify flaky tests in Python projects. FlaPy repeatedly executes the entire test suite of a given project under controlled conditions and monitors variations in test outcomes across runs. Based on the consistency and dependencies observed, it automatically categorizes tests into four groups: not flaky (tests that consistently pass or fail), non-order-dependent (flaky tests unrelated to execution order), order-dependent (tests whose outcomes vary depending on the execution order of other tests), and infrastructure (flaky tests caused by external factors such as network or file system instability).

FLAST [85] detects flaky tests by using test code similarity. It operates by analysing the source code similarity between new tests and previously identified flaky and non-flaky tests. Each test is first tokenized using a bag-of-words model, where tokens such as keywords, identifiers, and literals are transformed into numerical vectors weighted by their term frequency. FLAST then computes cosine similarity between test vectors to measure how closely they resemble each other and applies a k-nearest neighbour (k-NN) classifier to predict whether a new test is flaky.

iPFlakies [56] is a framework designed to detect and automatically repair order-dependent flaky tests in Python projects. It extends the concepts of *iDFlakies*, originally developed for Java, to the Python ecosystem, adapting its analysis and instrumentation techniques to Python's dynamic testing frameworks such as `pytest` and `unittest`. iPFlakies repeatedly executes tests in different orders to expose inconsistencies in test outcomes and identifies pairs or groups of tests whose results change based on execution sequence. Once order-dependent tests are detected, the framework analyses their execution traces and shared state interactions to infer the root causes of dependency.

Current flaky test detection tools are mainly available for Java and Python. Looking at

previous reviews on tools that are used to detect flaky tests in JavaScript programs [30,31], we identified only two tools. The first one from King et al. [33], uses machine learning techniques to detect flaky tests. They model the problem like a disease in which symptoms can identify a flaky test. The tool uses static and dynamic metrics (e.g., Test Assertion Count, Coupling Between Objects, Cyclomatic Complexity, Average Execution Time, Failure Rate) to build a predictive model. The Bayesian networks classifier is trained on historical data from the Ultimate Software company. The other tool is *NodeRacer* [32], which can detect test flakiness that is caused by event races in the Node.js platform. The NodeRacer approach is composed of three main phases: collecting information, inferring the relation between the callbacks, and rerunning the application. NodeRacer instruments the application and collects information relevant for the following phase from a run, driven by a test or by a manual interaction with the application. After that, it uses the collected information to infer a happens-before relation between the callbacks. Finally, it reruns the application a number of times in a special mode that selectively postpones callbacks while respecting the happens-before relation, to expose event race errors.

Beyond tool-based detection, several studies have proposed alternative approaches that rely on predictive modelling and data driven analysis to identify flaky tests. Machalica et al. [17] introduced a predictive test selection strategy aimed at improving efficiency in continuous integration environments. Their approach learns from a large dataset of historical test executions to predict which subset of tests is most likely to reveal faults for each submitted change. By applying machine learning techniques, the model prioritizes tests based on their historical behaviour and relevance to the modified code, thereby reducing overall testing cost while maintaining fault detection effectiveness.

Groce et al. [92] focused on flaky test detection and debugging in Python by formulating a theoretical model of non determinism and applying automated test generation to Python library code. Their work provides a practical understanding of how non-deterministic behaviour can emerge in testing environments and demonstrates how generated tests can expose subtle sources of flakiness that may not surface through standard reruns.

Pinto et al. [93] used machine learning techniques to predict flakiness directly from test source code. They constructed a dataset of over 64000 test cases, each executed 100 times, to label tests as flaky or stable. Using this dataset, they trained classifiers to learn a "vocabulary of flakiness," identifying lexical and structural code patterns that are more likely to correlate with nondeterministic test behaviour.

More recently, Parry et al. [94] extended this line of research by using machine learning to classify tests into order-dependent and non-order-dependent groups. Their study analysed

the test suites of 26 Python projects and compared the effectiveness of static features derived from source code and dynamic features derived from runtime behaviour as predictors of flakiness. Their results showed that combining both feature types yields better predictive performance than relying on either alone.

In addition, some recent work has explored combining rerunning-based detection with machine learning models to reduce the execution cost of flaky test detection. For example, Parry et al. [95] propose CANNIER, which uses machine learning predictions as a heuristic to prune the search space of rerunning-based techniques. While effective at reducing time and cost, such approaches focus on optimizing existing detectors rather than exposing new flaky behaviours or diagnosing specific flakiness causes, and are therefore orthogonal to the techniques considered in this thesis.

Taken together, these tools and approaches show that research on test flakiness has made significant progress. However, most existing solutions have been designed for specific ecosystems, particularly Java, and only recently extended to other languages such as Python. The variety of approaches, from dynamic detectors like iDFlakies and RootFinder to static and predictive models such as FLAST and vocabulary based classifiers, indicates that researchers have explored different levels of automation and analysis, including dynamic, static, and combined methods, to address the common challenge of nondeterminism in testing. Yet, despite this growing body of work, JavaScript remains relatively unexplored, even though it is one of the most widely used languages in modern software development and testing.

2.2.3 Responses to Flaky Tests

When developers detect flaky behaviour, they usually follow different strategies to respond to it. This includes scenarios where the developers attempt to eliminate the flaky behaviour altogether or reduce its possible occurrence. In general, developer responses can be grouped into three main categories [25, 96–98]:

- **Fixing flakiness:** Developers aim to completely eliminate the cause of flakiness by identifying and correcting the underlying issue in either the test code or the code under test. This may involve refactoring the test, improving synchronization, cleaning shared state, or stabilizing external dependencies such as file systems or network calls.
- **Improving flakiness:** When the root cause cannot be easily removed, developers attempt to reduce the likelihood or impact of flakiness. This includes practices such as quarantining unstable tests, rerunning failed tests to confirm failures, adding waits

or retries, or restructuring the build process to isolate problematic tests. These approaches help maintain workflow stability while allowing ongoing investigation of the flaky behaviour.

- **Ignoring flakiness:** In some cases, developers choose to ignore flaky tests, particularly when the failures are rare, non-critical, or too costly to diagnose. Such tests may remain in the suite with their failures tolerated, muted, or temporarily disabled. While this strategy reduces immediate disruption, it risks masking underlying defects and reducing confidence in test results over time.

Luo et al. [7] discussed common fixes that developers use for the top three categories of flaky tests in Java projects: *Async Wait*, *Concurrency*, and *Test Order Dependency*. They explained that some changes to the test code or code under test completely eliminate the flakiness, while others only decrease the likelihood of failures in the flaky tests. In other words, some changes do not completely eliminate or fix flakiness; they simply decrease the likelihood of flaky behaviour. Thorve et al. [8] found that changes aimed at improving the implementation in 53/77 of examined commits. In addition, they identified that 10/77 commits were targeted to remove the flaky tests altogether.

One of the most common strategies used in practice is to isolate flaky tests from the rest of the test suite by placing them in a quarantined area. Quarantining refers to the process of separating known or suspected flaky tests so that they do not interfere with regular testing or block continuous integration pipelines. Tests placed in quarantine are excluded from the main build results and are executed in a separate environment, where they can be monitored, diagnosed, and eventually fixed without affecting overall build stability [96–98]. This approach allows developers to maintain a reliable signal from healthy tests while continuing to gather information about flaky ones. New test monitoring tools offer some mechanism to quarantine flaky tests by identifying them from previous reruns (e.g., [99]).

As manual strategies can be time-consuming and error-prone, several tools have been proposed in recent years to automatically repair flaky tests (Table 2.3). Dutta et al. [100] proposed several repair strategies for flaky tests caused by randomness in machine learning projects, where nondeterminism often arises from uncontrolled random seeds, stochastic algorithms, or tolerance-based assertions. Their approach focuses on improving test reproducibility and reliability by tuning hyperparameters, fixing random seeds to ensure deterministic execution, and adjusting assertion thresholds to accommodate small but acceptable variations in numerical results.

Zhang et al. [101] introduced a repair tool that addresses flaky tests caused by implementation-dependent behaviour of standard library components, similar to the issues investigated by NonDex [57]. NonDex is a framework that detects flakiness arising from code that assumes deterministic behaviour in APIs or libraries that are, in fact, non-deterministic, such as hash-based data structures (e.g., HashMap or HashSet) or methods that return unordered collections. NonDex executes tests multiple times with controlled randomization of such API behaviours, thereby exposing tests that rely on these unintended deterministic assumptions. Building on this idea, Zhang et al.'s repair tool automatically detects and fixes these implementation dependencies by enforcing deterministic ordering or replacing nondeterministic API calls with stable, predictable alternatives.

Shi et al. [52] proposed *iFixFlakies*, an automated repair framework for order-dependent flaky tests in Java. It operates by analysing the relationships between tests to identify order dependencies, locating the source of shared state pollution, and automatically inserting cleanup or reinitialization code where needed. *iFixFlakies* uses dependency analysis and test reruns to determine which tests introduce state changes that cause subsequent tests to fail, then synthesizes patches that restore the state to its expected configuration.

Building on this concept, Wang et al. [56] developed *iPFlakies*, a Python adaptation of *iFixFlakies*. Like its Java counterpart, *iPFlakies* detects and repairs order-dependent flaky tests but is designed for Python testing frameworks such as `pytest`. The framework repeatedly reorders and executes tests to expose order-dependent failures, identifies the tests responsible for polluting shared state, and then searches for existing test or helper methods that can clean or reset the environment before rerunning the affected test. This strategy helps restore test independence without requiring manual intervention.

Li et al. [63] proposed *ODRepair*, an automated repair approach for order-dependent flaky tests that uses automatic test generation to produce cleanup code. Unlike *iFixFlakies* or *iPFlakies*, which rely on existing project code to restore state, *ODRepair* dynamically detects shared state pollution, generates new code to reinitialize or reset the affected state, and integrates the fix directly into the test suite. This approach ensures that each test executes in a clean environment, thereby eliminating the dependency between tests and improving overall test determinism.

2.3 Flaky Tests in JavaScript

Test flakiness has been widely studied in Java and Python programs, with only a few studies examining flakiness in JavaScript. We discuss studies that investigated test flakiness in

Table 2.3: Automated Tools for Repairing Flaky Tests

Tool / Study	**Language**	**Flakiness Type Addressed**	**Description**
Dutta et al. [100]	Python / ML	Randomness	Fixes flakiness by tuning hyperparameters, fixing seeds, and adjusting assertions.
NonDex [57]	Java	Implementation dependency	Detects flakiness from nondeterministic APIs using controlled randomization.
Zhang et al. [101]	Java	Implementation dependency	Repairs API-related flakiness via deterministic replacements and ordering.
iFixFlakies [52]	Java	Order dependency	Repairs order-dependent tests by inserting cleanup or reinitialisation code.
iPFlakies [56]	Python	Order dependency	Python version of iFixFlakies using reordering and state cleanup.
ODRepair [63]	Java	Order dependency	Generates cleanup code to reset shared state and restore determinism.

JavaScript as follows.

Costa et al. [22] conducted a large-scale empirical study on flaky tests across five programming languages (Java, Python, Go, C, and JavaScript) using data collected from both open-source and industrial projects. Their methodology involved mining thousands of test executions from continuous integration logs to identify flaky tests and manually analysing their root causes. They systematically classified each flaky test according to previously established taxonomies of flakiness causes, such as those by Luo et al. [7] and Thorve et al. [8]. The study revealed that a small number of causes account for the majority of flaky tests across all languages, with more than 78% of cases explained by only five root causes. Specifically for JavaScript, the most dominant causes were identified as *async wait*, *concurrency*, and *platform dependency*, highlighting the influence of asynchronous execution and environmental variation on flakiness in JavaScript testing frameworks. Yost [102] leveraged stress testing and reordering test suites to uncover flaky tests in JavaScript programs. By subjecting the application to stress conditions and modifying the order of test suite execution, the aim was to reveal the hidden issues that might not surface under normal circumstances. This study differs from ours in that it reorders test suites 10 times and runs each test suite only once, as well as attempting to introduce machine stress to mitigate non-order-dependent flakiness.

FlakyFix [103] explored the use of large language models to automatically predict categories of fixes applied to flaky tests. The study aimed to understand how different types of fixes correspond to specific flakiness causes by analysing test code before and after patches. To achieve this, the authors trained machine learning models on a large dataset of historical flaky test fixes drawn from open source repositories. Two pre-trained code models, CodeBERT and UniXCoder, were fine-tuned for this task using code written in six programming

languages, including JavaScript. By learning from language patterns, syntax structures, and contextual cues in the code, *FlakyFix* was able to identify the most likely fix category for a given flaky test, such as synchronization, environment handling, or order dependency. For JavaScript, the most frequent fix categories were related to *async wait* and *timing issues*.

To the best of our knowledge, there are only two tools for detecting flaky tests in JavaScript programs [32, 33]. *NodeRacer* [32] focuses on detecting concurrency-related issues in Node.js applications by systematically exploring and controlling thread interleaving during test execution. Although it was not explicitly designed for flakiness detection, it can expose non-deterministic behaviours that often lead to flaky tests, particularly those caused by asynchronous operations, race conditions, and event loop scheduling.

King et al. [33] use Bayesian networks to classify and predict flaky tests. They use a set of static and dynamic metrics (e.g., Test Assertion Count, Coupling Between Objects, Cyclomatic Complexity, Average Execution Time, and Failure Rate) to build a predictive model for flaky tests.

Despite these efforts, existing work on JavaScript flakiness remains limited in scope, often focusing on rerun-based detection or isolated causes, leaving systematic detection and mitigation of order-dependent and environment-dependent flakiness underexplored.

2.4 Summary

In summary, previous research has provided valuable insights into the nature, causes, and mitigation strategies of flaky tests across different programming languages and systems. Empirical studies have shown that flakiness is a widespread and persistent issue that significantly affects software quality, developer productivity, and confidence in automated testing. The literature identifies several major sources of flakiness, including concurrency and asynchronous behaviour, order dependency, resource leaks, network instability, timing issues, random values, platform differences, and environmental variations. Although these causes have been widely studied, most existing research has focused on languages such as Java and Python, where automated testing is more established and better supported.

Researchers have also proposed a wide range of techniques and tools for detecting, predicting, and addressing flaky behaviour. Detection methods are generally classified as static, dynamic, or hybrid, depending on whether they rely on code analysis, test execution, or a combination of both. Dynamic approaches that repeatedly execute tests, reorder them, or monitor coverage information remain the most common. In addition, several tools have been developed to automatically detect and repair flaky tests by targeting specific

causes such as randomness, order dependency, and implementation dependence. Developers also use practical strategies such as refactoring test code, isolating unstable tests, and using quarantine mechanisms to minimize the negative impact of flakiness in continuous integration environments.

Despite the growing body of work in this area, research on test flakiness in JavaScript remains limited. Only a small number of studies have investigated the nature and causes of flaky tests in JavaScript projects, and only two tools have been found that specifically support this language. Given that JavaScript is dynamic, event-driven, and widely used across web, mobile, and server platforms, it introduces unique sources of instability that differ from other ecosystems. In particular, test order and environmental dependencies have not been thoroughly explored in JavaScript, even though they are highly relevant given the language's reliance on asynchronous operations, diverse runtime environments, and platform-specific configurations. These gaps motivate the problem addressed in this thesis, as outlined in Section 1.2.

Chapter 3

Methodology

Software engineering is widely recognized as a cross-disciplinary field that involves technical, social, and psychological dimensions [104]. Four common research methods in software engineering are *scientific*, *engineering*, *empirical*, and *analytical*. Scientific methods build models based on observation, such as simulations. Engineering methods involve studying existing solutions, proposing improvements, and evaluating them. Empirical methods develop and assess models through case studies or experiments, while analytical methods propose formal theories and compare them against empirical observations.

This thesis uses an *empirical paradigm*, as empirical research seeks to generate knowledge by comparing theory with observations of reality [105]. Given the limited prior research on flaky tests in JavaScript programs, mining software repositories is appropriate for enhancing our understanding of flaky tests, refining relevant research questions, and evaluating hypotheses and solutions. The overall form of empirical software engineering applied in this work is shown in Figure 3.1.

In addition to its empirical foundation, this thesis also involves *experimental* and *engineering* components. The thesis goes beyond observation and analysis, incorporating the design, development, and evaluation of tools and techniques for detecting and mitigating different types of flaky tests.

As outlined in Chapter 1, this work is guided by three broad research questions. In this chapter, we detail our research philosophy and epistemic stance. We then introduce the more focused, smaller-scope research sub-questions used to address our broad research questions. We also provide an overview of the methods used to answer each sub-question. The detailed methodologies for each research question are presented in their respective chapters (4 to 6).

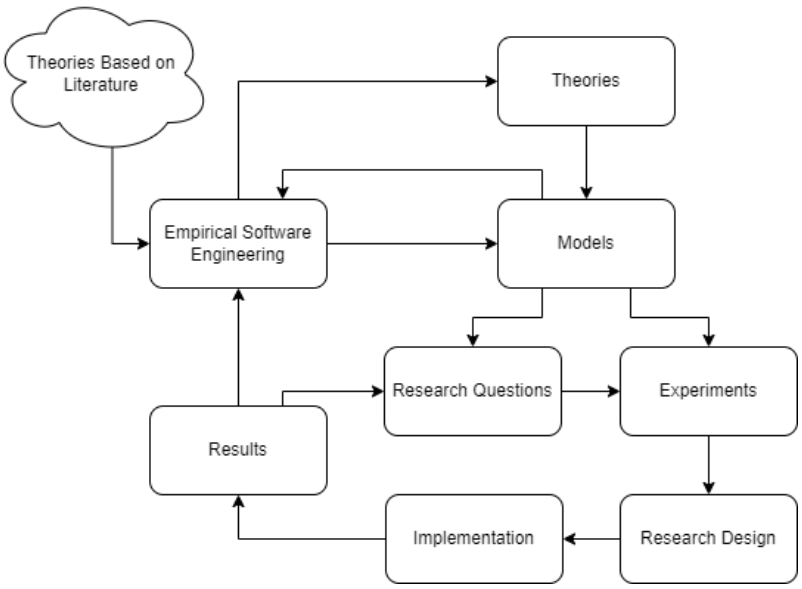

Figure 3.1: Empirical Software Engineering Model followed in this research (adopted from [106])

3.1 Research Philosophy

Empirical methods aim to compare proposed theories with real-world observations. They can be conducted using either quantitative or qualitative approaches, such as case studies, surveys, or experiments. Quantitative research collects, analyses, and interprets numerical data to address research questions requiring measurable evidence, while qualitative research focuses on textual or descriptive data to address questions about meaning, processes, or context [107]. Both approaches can investigate the same topic but typically address different types of questions [108]. Detailed discussions of these methods are provided by Bianchi et al. [109] and Williams et al. [107].

Although empirical studies have long been fundamental in many established scientific disciplines, their adoption in software engineering emerged more gradually and remained relatively limited until the early 2000s. Today, however, they form an essential part of both research and practice in the field [110]. Software engineering is highly people-oriented and, as such, inherits the variability and unpredictability of human processes. These circumstances make it difficult to establish universal guiding principles, leading some to describe software engineering as more of an *art* or *craft* than a rigid science [111]. Moreover, the diversity of software products, ranging from large-scale real-time systems to embedded and

web-based software, adds further complexity to empirical investigation.

The choice of empirical method in a study depends on various factors, including available resources, access to subjects, and the ability to control relevant variables [112]. In this research, the empirical paradigm is applied to study the presence and causes of flaky tests in JavaScript programs, as well as the strategies developers employ to address them. To this end, we conducted a large-scale empirical mining study of flaky tests in JavaScript using open-source projects hosted on GitHub. We employed both quantitative and qualitative methods to classify and measure the prevalence of flakiness, as well as to interpret developer responses and contextualize the findings. This philosophy underpins the three research questions outlined in Chapter 1: an observational empirical study of flaky tests in real-world projects (RQ1), an experimental investigation through systematic test reordering to detect order-dependent flaky tests (RQ2), and a combination of software repository mining and cross-platform evaluation across operating systems, Node.js versions, and browsers to understand and mitigate test flakiness caused by variations in test environment (RQ3).

3.2 Data Collection

Using a suitable data collection method is critical for obtaining reliable results. Several data collection methods are used in software engineering research. Based on the way that researchers interact with participants, these methods can be classified into three categories:

Direct techniques need direct contact with a participant population.

Indirect techniques require indirect contact with participants, e.g., access to their environment as they work, but without requiring either direct access to participants or for participants and researchers to interact.

Independent techniques need access to work artifacts, such as source code or documentation.

For this research, the independent techniques align well with the research's objectives. In independent techniques, researchers attempt to gain insight into how software engineers work by examining their output and by-products, such as source code, documentation, reports, work requests, and change logs. These archives can be accessed by mining open-source software repositories, which provide free, unrestricted access to software projects.

GitHub[1] is one of the well-known and widely used code hosting sites that hosts high-quality open-source software. *GitHub* is used for gathering test data (including from source code, commit messages, and pull requests) from publicly available JavaScript projects. The process of constructing and mining the dataset is explained in more detail in Chapters 4 to 6.

3.3 Sub-research Questions

This section presents the sub-research questions derived from the objectives and main research questions of this thesis (outlined in Chapter 1). We also provide a brief overview of the methodology used to address each of them. Figure 3.2 illustrates the detailed research questions.

3.3.1 Understanding Flaky Tests in JavaScript

RQ1 What are the main causes of flaky tests in real-world JavaScript projects, and how do developers respond to them?

To address RQ1, we performed a large-scale empirical study on open-source repositories. The detailed methodology used to answer RQ1 is explained in Chapter 4. The sub-research questions we are addressing here are:

RQ1.1 What are the main causes of flaky tests in JavaScript projects?

We collected 452 flaky test-related commits from popular JavaScript projects hosted on GitHub. We then analysed the cause of the flaky tests and the fix strategies followed. Our target was to analyse commits that are believed to refer to tests with flaky behaviour, rather than exposing flakiness by compiling and analysing the programs themselves. The study has been conducted in two phases. First, commits (including commit messages and source code changes) from open-source JavaScript projects are collected. In the second phase, these commits were manually analysed to identify the main causes of test flakiness.

RQ1.2 What are the strategies followed when dealing with flaky tests in JavaScript projects?

Using the dataset, we manually analysed the strategies used to fix those flaky tests. This involved examining the source code diff, commit messages, and corresponding issue discussions to identify how developers responded to and resolved flaky tests.

[1] `https://github.com/`

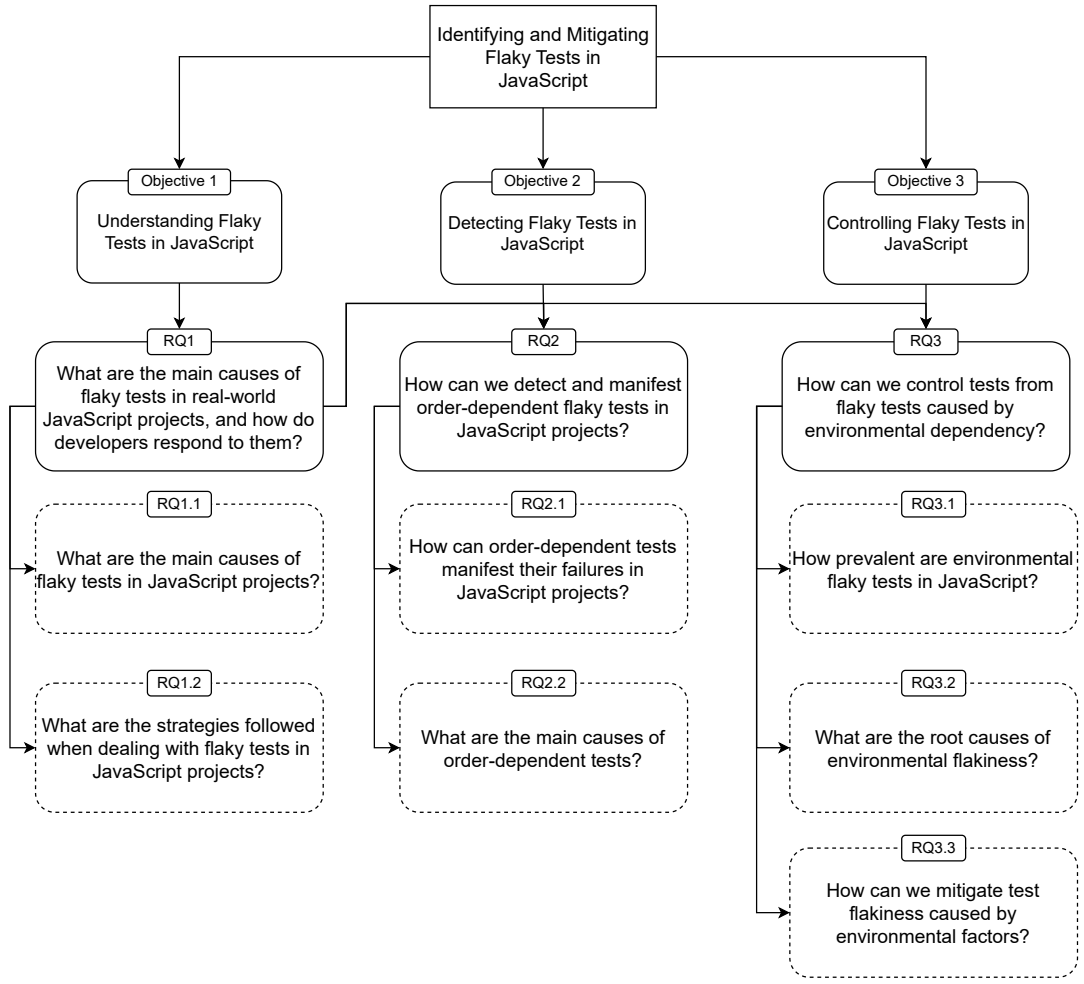

Figure 3.2: Research objectives and their corresponding sub-research questions in this thesis

3.3.2 Detecting Flaky Tests in JavaScript

RQ2 How can we detect and manifest order-dependent flaky tests in JavaScript projects?

To address RQ2, we conducted an in-depth investigation into order-dependent tests in JavaScript. We addressed two sub-research questions to understand the prevalence of order-dependent tests within popular open-source JavaScript projects after actively searching for them. We also wanted to understand the reasons for their failures, i.e., what states these tests share in common. The detailed methodology used to answer RQ2 is explained in Chapter 5.

RQ2.1 How can order-dependent tests manifest their failures in JavaScript projects?

We employed a multi-layer test reorder and rerun approach to uncover order-dependent tests in Jest. By altering the sequence in which the tests are executed within programs/test suites and rerunning them multiple times, we observed possible changes in test outcomes across different orders and runs. We evaluated a dataset of 81 projects. We reordered the tests and test suites 10 times each and ran each test 10 times.

RQ2.2 What are the main causes of order-dependent tests?

Using the dataset, we manually inspected each detected order-dependent test and classified the cause of flakiness into different categories. This process involved examining the test code, the code-under-test diffs, commit messages, and any related issue discussions to identify the main sources of order-dependent flakiness.

3.3.3 Controlling Flaky Tests in JavaScript Projects

RQ3 How can we control tests from flaky tests caused by environmental dependency?

To address RQ3, we conducted an in-depth investigation into environmental flakiness in JavaScript, focusing on the OS, browsers, and Node.js versions. We addressed three sub-research questions. The detailed methodology used to answer RQ3 is explained in Chapter 6.

RQ3.1 How prevalent are environmental flaky tests in JavaScript?

To investigate the prevalence of environmental flakiness in JavaScript, we employed a systematic rerun approach to uncover environmental flaky tests in JavaScript projects. By changing environmental factors and rerunning tests with different factors multiple times, we observe whether the test outcomes change. We evaluate a dataset of 116 projects. For each project, we rerun its test suites 10 times across three Node.js versions and three operating systems.

RQ3.2 What are the root causes of environmental flakiness?

To understand the main causes of environmental flaky tests in JavaScript. We manually inspected each detected environmental flaky test and classified the cause of flakiness into different categories.

RQ3.3 How can we mitigate test flakiness caused by environmental factors?

To mitigate such environmental flakiness causes by variation of OS, Node.js, or browsers, we developed an annotation-based approach that can sanitize and quarantine such flaky tests by skipping and reporting them (instead of failing the tests), which allows CI builds to continue/succeed without the need to rerun entire test suites or stopping the build for developers to inspect (flaky) failures.

3.4 Summary

This chapter outlined the overall research design and methodological framework adopted in this thesis. The study follows an empirical paradigm, combining both quantitative and qualitative analyses to investigate the causes, detection, and mitigation of flaky tests in JavaScript projects. We described the philosophical foundation of the research, justifying the choice of empirical, controlled, and quasi-experimental methods as appropriate for studying real-world software testing phenomena.

We then discussed data collection strategies and the use of independent techniques, particularly mining open-source repositories on GitHub, as a suitable and reproducible means of gathering evidence about flaky tests.

Finally, the chapter introduced the research questions derived from the main research objectives and provided an overview of the methodology used to address each. RQ1 investigates the causes of and responses to flaky tests through large-scale empirical analysis. RQ2 focuses on detecting and understanding order-dependent flakiness through systematic test reordering and controlled experimentation. RQ3 examines environmental flakiness and proposes a mitigation approach through automated test sanitization.

The detailed methodologies, implementation steps, and empirical evaluations for each research question are presented in the subsequent chapters.

MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
DOCTORATE WITH PUBLICATIONS/MANUSCRIPTS

We, the student and the student's main supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the student's contribution as indicated below in the Statement of Originality.	
Student name:	Negar Hashemi
Name and title of main supervisor:	Associate Professor Amjed Tahir
In which chapter is the manuscript/published work?	Chapter 4 – Understanding Flaky Tests in JavaScript
Describe the contribution that the student and members of the supervisory team have made to the manuscript/published work:[1] The student constructed the study, designed the methodology, collected and analysed the data, interpreted the results, and wrote the draft of the paper. The supervisors contributed to the design of the study, validation of the methodology, interpretation of the results, and reviewing and editing of the paper. Student (Negar Hashemi): 70% Supervisors and co-authors: 30%	
Please select one of the following three options:	
◉	**The manuscript/published work is published or in press** Please provide the full reference of the research output: Negar Hashemi, Amjed Tahir, and Shawn Rasheed. "An Empirical Study of Flaky Tests in JavaScript," 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), Limassol, Cyprus, 2022, pp. 24-34.
○	**The manuscript is currently under review for publication** Please provide the name of the journal:
○	**It is intended that the manuscript will be published, but it has not yet been submitted to a journal**
Student's signature:	Negar Hashemi Digitally signed by Negar Hashemi Date: 2026.01.26 23:04:25 +13'00'
Main supervisor's signature:	Amjed Tahir Amjed Tahir 2026.01.29 07:48:41 +03'00'
This form should be placed at the beginning of each relevant thesis chapter.	

[1] Refer to the Massey University Publishing and Authorship guidelines (OneMassey for staff, Stream for students) and/ or Contributor Roles Taxonomy (CRediT) guidelines for guidance.

Chapter 4

Understanding Flaky Tests in JavaScript

4.1 Introduction

While flaky tests have been recognized as a common challenge in software testing, there has been relatively little research on their characteristics and how they manifest in JavaScript projects. Given the widespread use of JavaScript and its diverse ecosystem, understanding the nature of flakiness in this context is essential.

The first objective of this thesis is to build a deep understanding of flaky tests in JavaScript. In this chapter, we investigate the main causes of test flakiness in JavaScript projects and how developers typically respond to them in practice. To guide this investigation, we focus on the following main research question, as introduced in Chapter 1:

RQ1: What are the main causes of flaky tests in real-world JavaScript projects, and how do developers respond to them?

As mentioned in Chapter 3, to address this question comprehensively, we divided it into two sub-questions:

RQ1.1. What are the main causes of flaky tests in JavaScript projects? This question focuses on identifying the underlying causes of flaky tests, which is essential for characterising the problem space and understanding how flakiness manifests in JavaScript projects.
RQ1.2. What are the strategies followed when dealing with flaky tests in JavaScript

projects? This question examines how developers respond to flaky tests in practice, providing insight into current behaviours, common mitigation strategies, and potential gaps that require further research or tool support.

4.2 Methodology

To answer the two research questions, 452 commits from popular JavaScript projects on GitHub were collected and analysed to determine the causes and mitigation strategies followed. These commits indicate flaky test behaviour and are identified using a keyword-based filtering process described in Section 4.2.1. Our target was to analyse commits believed to contain tests with flaky behaviour, rather than expose flakiness by running and analysing the projects themselves. To achieve this, we conducted the study in two phases. First, commits (including commit messages and source code changes) from open-source JavaScript projects are collected and manually analysed to identify the main causes of flakiness. The criteria used to select the projects and extract the relevant commits are explained in the subsequent methodological subsection (Section 4.2.1). Second, the strategies followed when those flaky tests were fixed are analysed to identify the responses to the flaky tests.

4.2.1 Dataset

We constructed a dataset of JavaScript projects obtained from GitHub. We targeted only popular projects based on their GitHub *star* rating. GitHub's stars let users mark their interests or helpful repositories. In a way, it is a metric that can serve as a proxy for the interest a project has generated. In this study, it was found that the number of stars is highly correlated with the number of forks and contributors, indicating popularity.

4.2.2 Mining Process

We used the GitHub REST API[1] to search through all publicly available GitHub repositories and filter the results of the commits for analysis. We first extracted the top 40 starred JavaScript projects on GitHub. We chose the top 40 projects because they provide a large enough sample (number of commits) that is deemed suitable for our analysis. Note that the number of commits analysed in our study is much higher than those in previous studies (i.e., [7] and [8]). Luo et al. [7] retrieved 486 commits in total, but analysed 201 commits that fix flaky tests from the Apache Software Foundation; while Thorve et al. [8] retrieved and

[1] `https://docs.github.com/en/rest/reference/search`

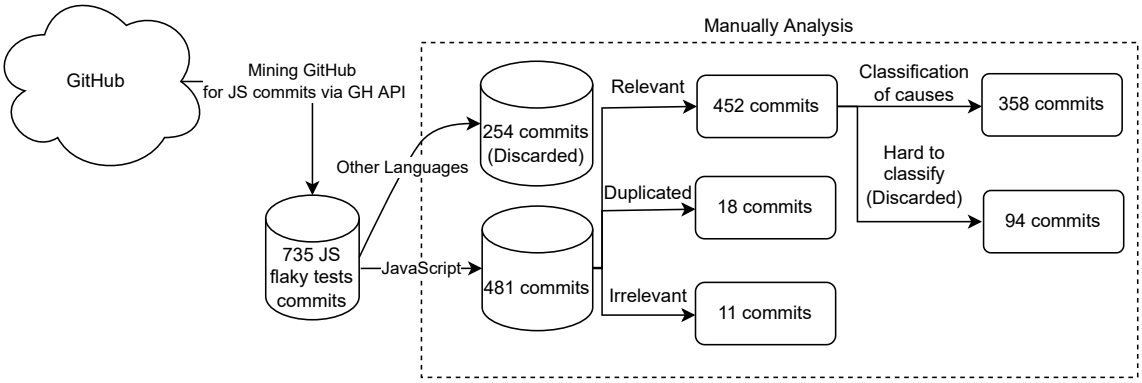

Figure 4.1: Mining Flaky Tests' Commits from GitHub's JavaScript Projects

analysed 77 commits from 29 Android projects. Romano et al. [21] analysed 235 flaky UI test samples found in 62 projects from both web and Android environments. In comparison, we extracted a total of 316,948 commits and ended up with 735 flaky tests' related commits.

Our mining process is shown in Fig. 4.1. By using GitHub REST API, we searched through 40 repositories for the following keywords: "flaky", "flakey", "flakiness", "intermit", "fragile", "brittle", "Intermittent", and "non-deterministic test". We found 18 repositories that matched at least one commit containing one of the keywords we used. This resulted in a total of 735 commits [2]. Table 4.1 shows summary statistics from the projects we analysed.

Out of those 735 commits, we found that 254 are related to languages other than JavaScript, so we excluded them, as we are interested only in flakiness related to JavaScript. Although we specifically selected JavaScript projects, many of these repositories also contain code written in other languages (e.g., build scripts, native extensions, or supporting modules implemented in TypeScript, Python, or C++). Some commits flagged by our keyword search referred to flaky tests or test fixes in these non-JavaScript components. Since these commits do not reflect flakiness in JavaScript test code or JavaScript code under test, we removed them from the dataset. This left us with 481 relevant commits for our analysis.

4.2.3 Manual Analysis of Commits

We manually analysed all 481 commits that we included. For each commit, we analysed the commit message, any code changes in the files in the repository, as well as the associated

[2]This search was conducted on May 8th, 2021.

issues (if any) in the issue tracker. Two reviewers then individually classified all 481 commits based on the cause of flakiness and compared their results. To inform our classification, we checked the associated issues (if any), pull requests, and changes in the test and code under test (CUT). When the two reviewers disagreed, or when either reviewer was unsure about the classification, a third reviewer independently analysed the case. This occurred for 139 out of 481 commits (28.9%). The reviewers then discussed these cases collectively until consensus was reached. All disagreements were resolved through discussion, resulting in full agreement on the final classification.

We categorise each of these commits into one of the following :

- **Relevant**: commits that are directly related to flaky tests.
- **Irrelevant**: commits that contain one of the keywords, but are not related to flakiness.
- **Duplicated**: identical commits.

Of those 481 commits, we found that 11 commits were irrelevant (not related to test flakiness). For example, in the following commit message from `atom/atom`, the keyword "flaky" appears, but it does not refer to flaky tests.

"Add the ability to retry flaky tests to Jasmine 1.3 specs."

In addition, 18 commits were found to be duplicates and were therefore excluded. This resulted in 452 relevant commits, which we then manually classified.

Table 4.1: Projects from The Top 40 Most Starred JavaScript Projects on GitHub with Commit Messages Containing at Least One of The Search Keywords

Repositories	**#Stars**	**#Commits**	**JavaScript File**	**#Flakiness Related Commits**	**#Contributor**	**Age**	**Description**
freeCodeCamp/freeCodeCamp	320k	28253	47.4%	4	4287	Oct. 2014	Corpus of educational JavaScript projects
vuejs/vue	188k	3200	97.6%	2	388	Dec. 2013	JavaScript framework for building UI
facebook/react	174k	14448	95.7%	22	1495	July 2013	JavaScript library for building user interfaces
twbs/bootstrap	153k	21165	41.4%	5	1231	Oct. 2011	Web development JavaScript framework
electron/electron	96.6k	25574	6.5%	48	1069	July 2013	Desktop apps framework
nodejs/node	81.6k	34487	61.2%	467	3012	May 2009	Cross-platform JavaScript runtime environment
denoland/deno	77.7k	6310	23.6%	3	635	May 2018	A JavaScript and TypeScript runtime environment
mrdoob/three.js	74.6k	38172	54.2%	1	1482	Apr. 2010	JavaScript 3D Library
mui-org/material-ui	70.9k	17626	58.2%	19	2278	Nov. 2014	A framework to build React apps
storybookjs/storybook	64.5k	35707	9%	14	1365	April 2016	UI components development tool
atom/atom	56k	38404	88.3%	68	488	Dec. 2013	Cross-platform text editor
jquery/jquery	55.3k	6536	93.6%	9	276	Jan. 2006	JavaScript Library
chartjs/Chart.js	54.7k	4043	98.3%	1	380	June 2014	Data visualization library
ElemeFE/element	50.8k	4500	26.7%	1	556	Sep. 2016	A Vue.js-based UI Toolkit for Web
lodash/lodash	50.6k	8005	100%	2	310	April 2012	A modern JavaScript utility library
moment/moment	46k	3961	99.7%	8	590	Jan. 2015	A JavaScript date library
meteor/meteor	42.6k	24211	92.3%	50	465	Jan. 2012	JavaScript web framework
yarnpkg/yarn	40.1k	2346	98.7%	11	521	June 2016	JavaScript package manager
Total		316948		735	20828		

4.3 Result

This section presents the results in relation to the research questions. We summarise the key patterns observed in the collected commits, highlight the dominant causes of flakiness, and describe how developers addressed them in practice.

4.3.1 Flaky Tests Causes (RQ1.1)

To identify possible root causes of test flakiness in JavaScript, we categorised all commits based on the relevant causes of flakiness (based on the commit message or the change in the code files associated with the commit). We used the list of causes noted in previous empirical studies (i.e., [7,8,60,61]) on test flakiness as our baseline (see Table 2.1). We then classified each commit based on the list of flakiness causes. If the cause was determined to be new or not included in Table 2.1, we then add it as a new cause and continue with our classification.

During this process, we were unable to categorise or determine the cause of flakiness in 94 commits. Hence, these were categorised as *"unsure"* or *"hard to categorise"*, and we continued analysing the remaining 358 commits.

As it turned out, most of the causes we identified fit well into one or more of the categories noted in previous studies. However, some of the causes we identified led to new categories that best reflect the nature of flakiness, thus those have been categorised separately.

Table 4.2 shows the top 10 flaky test causes we identified in the projects we analysed. We present the results of our categories below, together with examples of each cause from the projects we analysed.

1. **Concurrency:** Flakiness arises when tests interact with code that relies on parallel or interleaving operations, leading to unpredictable behaviours such as race conditions or inconsistent ordering. We found that 74 of the flaky tests are related to concurrency issues. For example, Listing 4.1 [3] from `nodejs/node`, using `setImmediate` in line 6 could lead to race conditions, which lead to non-deterministic test outcomes.

   Another example of a concurrency related flaky test is Listing 4.2 from `Meteor`[4]. This test is flaky because the template-rendered callbacks are called after flush time (line 3), but not when the template is destroyed. If the client responded to the server

[3] `https://github.com/nodejs/node/commit/e071894202c49d1dc50c80e05cb667468ce68ac2`
[4] `https://github.com/meteor/meteor/commit/e5b5858203ab726d09086e707392c4e6aff6ab86`

Table 4.2: Overall Results of Flaky Test Categories (Causes) in JavaScript

Cause of Flakiness	**# of Commits**	%
Concurrency	74	20.7%
Async Wait	70	19.6%
OS	66	18.4%
Network	45	12.6%
Platform	37	10.3%
UI	21	5.9%
Hardware	17	4.7%
Time	12	3.4%
Resource Leak	10	2.8%
Other	13	3.6%
Total	365*	

* The total number of commits is 358, as there are 7 commits related to 2 causes of flakiness.

```
1 const keepOpen = setTimeout(() => {
2 @@ -20,7 +20,7 @@ const timer = setInterval(() => {
3     timer._onTimeout = () => {
4       throw new Error('Unrefd interval fired after being cleared.');};
5     setImmediate(() => clearTimeout(keepOpen));}
6     }
7     , 1);
```

Listing 4.1: Concurrency flakiness (project: node)

rejecting the method before the client's flush cycle, the rendered callback would never fire.

2. **Async Wait** Tests may fail intermittently when asynchronous operations do not complete as expected, often due to insufficient waiting, delayed callbacks, or incorrect assumptions about execution timing. Async Wait can be classified as a subcategory of concurrency. However, we classify it separately here mainly because it represents nearly half (48%) of the concurrency-related flaky tests. In the projects we analysed, 70 commits in total fit into the async wait category, such as Listing4.3 from `Electron`[5]:

   In this test, the IPC message was sent before it was fully loaded, so the history did

[5] `https://github.com/electron/electron/commit/c2d78deeca3a9ea2034e97b688c51cdabfd86cb2`

```
testAsyncMulti('spacebars - template - defer in rendered callbacks', [function (test,↩
    expect) {
  var tmpl = Template.spacebars_template_test_defer_in_rendered;
  var coll = new Meteor.Collection("test-defer-in-rendered--client-only");
  tmpl.items = function () {
    return coll.find();
  };
  var subtmpl = Template.spacebars_template_test_defer_ in_rendered_subtemplate;
  subtmpl.rendered = expect(function () {
    Meteor.defer(function () {
    });
  });
  var div = renderToDiv(tmpl);
  Meteor._suppress_log(1);
  coll.insert({});
}]);
```

Listing 4.2: Concurrency flakiness (project: meteor)

```
it('should clear the navigation history', async () => {
   loadWebView(webview, {
   nodeintegration: 'on',
   src: `file://${fixtures}/pages/history.html`
  })
const event = await waitForEvent(webview, 'ipc-message')
```

Listing 4.3: Async wait flakiness (project: electron)

not contain everything it should have, and could not clear all the pages, leading to non-deterministic outcomes.

Listing 4.4[6] from `FreeCodeCamp`, using `cy.contains('Gotonextchallenge').click();` in line 10 and then not waiting for the result before visiting the next page can cause flakiness.

3. **Operating Systems (OS)** Differences across operating systems, such as file system behaviour, process handling, or system-level configurations, can cause tests to behave inconsistently. We classify OS as a separate category of root causes from the platform (discussed later) because it represents a large percentage of flaky tests by itself. In total, we found 66 commits in the OS category. The example in Listing 4.5 from `nodejs/node`[7] shows a test that is timing out in OS X. In OS X, events for `fs.watch()`

[6] https://github.com/freeCodeCamp/freeCodeCamp/commit/71faba15da2c5bd675e40751a17cc7c2a06681b5

[7] https://github.com/nodejs/node/commit/fd54df1e674f408c6c1bc314726e23ef58da15f6

```
describe('project submission', () => {
   it('Should be possible to submit Python projects', () => {
    const { superBlock, block, challenges } = projects;
    challenges.forEach(challenge => {
      cy.visit(`/learn/${superBlock}/${block}/${challenge}`);
      cy.get('#dynamic-front-end-form')
        .get('#solution')
        .type('https://repl.it/@camperbot/python-project#main.py');
      cy.contains("I've completed this challenge").click();
      cy.contains('Go to next challenge').click();
```

Listing 4.4: Async wait flakiness (project: freeCodeCamp)

might only start showing up after a delay. To work around that, there is a timer in the test that delays the writing of the file by 100ms, but in some cases, this might not be enough. In this case, the event never fired, and the test times out.

```
if (process.platform === 'darwin') {
  setTimeout(function() {
    fs.writeFileSync(filepathOne, 'world');
  }, 100);
 } else {
  fs.writeFileSync(filepathOne, 'world');
}
```

Listing 4.5: OS flakiness (project: nodejs)

Another example of an OS flaky test is shown in Listing 4.6 from `Angular`[8]. The example shows a test that resulted in a `ECONNREFUSED` error (i.e., refused connection) when running on Windows. It happened because of the delay between starting and stopping the Chrome process in Windows[9].

```
function launchChromeAndRunLighthouse(url, flags, config) {
  const launcher = new ChromeLauncher({autoSelectChrome: true});
  return launcher.run().
    then(() => lighthouse(url, flags, config)).
    then(results => launcher.kill().then(() => results)).
    catch(err => launcher.kill().then(() => { throw err; }, () => { throw err; }));
}
```

Listing 4.6: OS flakiness (project: angular)

[8] https://github.com/angular/angular/commit/3a604ba0bf113563b0666f06744a749a3c3b769b
[9] https://github.com/paulirish/pwmetrics/issues/63#issuecomment-282721068

4. **Network** Tests that depend on network communication can become flaky when requests experience delays, timeouts, or fluctuations in connectivity or remote service availability. There are 45 commits that fit into the network category. One example we see in this category is Listing 4.7 from `nodejs/node`[10]. The test is intended to exercise IPv6-only behaviour and binds to port '0', relying on the OS to automatically assign an ephemeral port. The test implicitly assumes that the selected port is free for both IPv6 and IPv4. However, CI results indicate that the OS may assign a port that is already in use by an IPv4 socket but remains available for IPv6, leading to inconsistent test outcomes and test failures.

```
1  net.createServer().listen({
2    host,
3    port: 0,
4    ipv6Only: true,
5  }, common.mustCall());
```

Listing 4.7: Network flakiness (project: nodejs)

Another example is Listing 4.8 from React[11] showing a test that uses absolute URLs, which can lead to flakiness. Network-related flakiness occurs because the test relies on absolute URLs to load several external scripts. When these URLs are resolved at runtime, any delay, failure, or temporary unavailability in fetching these resources can cause the test to fail intermittently. Since the test depends on loading files over the network rather than local, deterministic paths, its behaviour becomes sensitive to network conditions, leading to flaky outcomes.

5. **Platform**: Platform-specific behaviour (e.g., nodejs/node versions, browser engines, or runtime environments) can trigger inconsistent results when the underlying execution model or APIs differ. We excluded OS related flaky tests here, although they were platform-related, and were an extensive enough subcategory that we discussed separately earlier in this section. We identified 37 commits in total from the platform category. Listing 4.9 from `Electron`[12], illustrates a test whose behaviour depends on platform-specific CI characteristics. In particular, the test is sensitive to timing and execution constraints imposed by the CI environment. The test times out regularly on Travis CI, but passes when built on Jenkins.

[10] `https://github.com/nodejs/node/commit/1c5a3f0d091ef34b82f89dce4c87b431cab4c3ee`
[11] `https://github.com/facebook/react/commit/fc572832b10792ce3aca764648e6f49d29dabadb`
[12] `https://github.com/electron/electron/commit/dcbc10ac3810cb2b4b88cd583b783666aad03ccc`

```
1 var __dirname = __filename.split('/').reverse().slice(1).reverse().join('/');
2 window.ReactWebWorker_URL = __dirname + '/../src/test/worker.js' + cacheBust;
3 document.write('<script src="' + __dirname + '/../build/jasmine.js' + cacheBust + '↩
      "><\/script>');
4 document.write('<script src="' + __dirname + '/../build/react.js' + cacheBust + '↩
      "><\/script>');
5 document.write('<script src="' + __dirname + '/../build/react-test.js' + cacheBust + ↩
      '"><\/script>');
6 document.write('<script src="' + __dirname + '/../node_modules/jasmine-tapreporter/↩
      src/tapreporter.js' + cacheBust + '"><\/script>');
7 document.write('<script src="' + __dirname + '/../test/the-files-to-test.generated.js↩
      ' + cacheBust + '"><\/script>');
8 document.write('<script src="' + __dirname + '/../test/jasmine-execute.js' + ↩
      cacheBust + '"><\/script>');
```

Listing 4.8: Network related flakiness (project: react)

```
1 it('can be manually resized with setSize even when attribute is present', done => {
2       w = new BrowserWindow({show: false, width: 200, height: 200})
3       w.loadURL('file://' + fixtures + '/pages/webview-no-guest-resize.html')
```

Listing 4.9: Platform related flakiness (project: electron)

Similarly, Listing 4.10 from `nodejs/node`[13] is flaky due to reliance on platform timing through repeated calls to `platformTimeout` (lines 3, 5, 8, 9, 11, and 18). The test uses these timeouts to control how long the event loop is kept busy (`busyLoop`) and when specific callbacks (such as `mustNotCall('eventloop blocked!')`) should fire. This design assumes that the event loop and filesystem operations will complete within particular time windows. However, these assumptions do not hold consistently across different platforms and environments.

6. **UI** Tests that interact with user interface components may become flaky due to rendering delays or event-handling issues. There are 21 commits that belong to the UI category. Listing 4.11 from `Atom`[14] shows a test aimed to verify that the cursor blinks when the editor is focused, and the cursors are not moving. It expects the blinking cursor to start in the visible state (line 3) and then transition to the invisible state. However, the cursor's initial state is not important, since it toggles between visible and invisible states.

   Also, Listing 4.12 from `atom`[15] is flaky due to UI related issues. Resize event in line

[13] `https://github.com/nodejs/node/commit/6963a93d0ec7b0020f243f2d49eba889a869cbed`
[14] `https://github.com/atom/atom/commit/577dfe8deb54d47f0ff866647c238a79e5e34e62`
[15] `https://github.com/atom/atom/commit/a028eca8ba60285a85fd42854c85f8fe07437f1e`

```
const common = require('../common');
const fs = require('fs');
const platformTimeout = common.platformTimeout;
const t1 = setInterval(() => {
  common.busyLoop(platformTimeout(12));
}, platformTimeout(10));
const t2 = setInterval(() => {
  common.busyLoop(platformTimeout(15));
}, platformTimeout(10));
const t3 =
  setTimeout(common.mustNotCall('eventloop blocked!'), platformTimeout(200));
setTimeout(function() {
  fs.stat('/dev/nonexistent', () => {
    clearInterval(t1);
    clearInterval(t2);
    clearTimeout(t3);
  });
}, platformTimeout(50));
```

Listing 4.10: Platform related flakiness (project: nodejs)

3 is unreliable and may not be emitted right away. This could cause the test code to wait for an unrelated update promise (e.g., cursor blinking) that was triggered by the resize event.

7. **Hardware** Hardware differences, such as CPU speed, available memory, or system load, can influence execution timing and cause nondeterministic failures. We found a total of 17 commits related to the hardware category. In Listing 4.13 example from `nodejs/node`[16], the test runs on a Raspberry Pi, which exposes hardware-related flakiness. The issue stems from the high level of concurrency created in lines 2–4 and 6–12, where the test configures `N = 10` and `M = 10` and then spawns 100 HTTP requests within nested loops. While this workload is manageable on typical development machines, a Raspberry Pi has significantly more limited CPU power, memory, and I/O throughput. As a result, the test might intermittently miss some responses, take longer than expected, or fail to reach the expected total of `N * M` responses, even though the underlying functionality is correct.

   Another example from this category is Listing 4.14 from `nodejs/node`[17]. When the test loops rapidly and creates many new connections (in line 5), it can cause the Raspberry Pi 2 Model to malfunction.

[16] `https://github.com/nodejs/node/commit/bbf46215489b15a79468ba26b233219bb6a3e41f`
[17] `https://github.com/nodejs/node/commit/5d7265f601f733b8a08b65f05379e39904a64593`

```
1  await component.getNextUpdatePromise()
2       const [cursor1, cursor2] = element.querySelectorAll('.cursor')
3       expect(getComputedStyle(cursor1).opacity).toBe('1')
4       expect(getComputedStyle(cursor2).opacity).toBe('1')
5       await conditionPromise(() =>
6         getComputedStyle(cursor1).opacity === '0' && getComputedStyle(cursor2).opacity↩
              === '0')
7       await conditionPromise(() =>
8         getComputedStyle(cursor1).opacity === '1' && getComputedStyle(cursor2).opacity↩
              === '1')
9       await conditionPromise(() =>
10        getComputedStyle(cursor1).opacity === '0' && getComputedStyle(cursor2).opacity↩
              === '0')
```

Listing 4.11: UI related flakiness (project: nodejs)

```
1  setEditorHeightInLines(component, 13);
2  await setEditorWidthInCharacters(component, 50);
3  expect(component.getRenderedStartRow()).toBe(0);
4  expect(component.getRenderedEndRow()).toBe(13);
```

Listing 4.12: UI related flakiness (project: atom)

8. **Time** Tests relying on system time or scheduling can cause flakiness if the test environment introduces delays, time drift, or unexpected timing boundaries. We classify 12 commits in total into the time category. Listing 4.15 from `nodejs/node`[18] shows a flaky test that is the result of relying on the system's time with the use of the `Date.now()` function in line 1.

   Listing 4.16 from `Moment`[19] can cause flakiness because the equality of times in line 2 can be different by a millisecond.

9. **Resource Leak** Flakiness occurs when resources such as memory, file handles, or open connections are not properly released, causing interference with subsequent test executions. There are 10 commits from the resource leak category. Listing 4.17 from `nodejs/node`[20] shows a flaky test due to `EMFILE`, which means that a process is trying to open too many files.

   Listing 4.18 from `nodejs/node`[21] shows a test that is not performing proper clean-up and so it would fail if run more than one time on the same machine.

[18] https://github.com/nodejs/node/node/commit/a025723e0191e3526bf1e83b066c2f1c78730134
[19] https://github.com/moment/moment/commit/a0857cd56fcec6de35d14456dc1db311e6011172
[20] https://github.com/nodejs/node/commit/bea41bc945dff623961d65e86d5a4cd0d18fffb6
[21] https://github.com/nodejs/node/commit/f030d8426aa3241ad180c385f15134548579e7ce

```
var responses = 0;
var N = 10;
var M = 10;
server.listen(common.PORT, function() {
 for (var i = 0; i < N; i++) {
   setTimeout(function() {
     for (var j = 0; j < M; j++) {
       http.get({ port: common.PORT, path: '/' }, function(res) {
         console.log('%d %d', responses, res.statusCode);
         if (++responses == N * M) {
           console.error('Received all responses, closing server');
           server.close();
```

Listing 4.13: Hardware related flakiness (project: nodejs)

```
const server = net.createServer(function listener(c) {
  connections.push(c);
}).listen(common.PORT, function makeConnections() {
  for (var i = 0; i < NUM; i++) {
    net.connect(common.PORT, function connected() {
      clientConnected(this);
    });
```

Listing 4.14: Hardware related flakiness (project: nodejs)

10. **Other Known Causes** We note that other remaining causes (13 commits) are related to a diverse number of issues. The remaining causes are (each provided with an example from the projects we analysed): IO, test order dependency, floating point operations, randomness, and implementation dependency.

    Listing 4.19 from `Yarn`[22] illustrates a case that we classified as order-dependent flakiness. In this example, several `add` tests interact with a shared Yarn cache. The flakiness arises because the correctness of each test depends on the order in which file-system operations on the cache occur across tests. When multiple tests invoke `execCommand`, their reads and writes to the shared cache interleave in different orders, occasionally leaving the cache in an inconsistent state and triggering `ENOENT` errors. As a result, the same test may pass or fail depending on whether it accesses the cache before or after another test modifies it, creating an order-dependent behaviour across executions.

    To remove this dependency, the fix assigns each `execCommand` invocation its own isolated cache directory. This prevents interference between tests and ensures that no

[22] `https://github.com/yarnpkg/yarn/commit/ccf5812b9bbcc4931b9d32966f48a09e4448a8f0`

```
1 const now = Date.now();
2 while (now + 10 >= Date.now());
```

Listing 4.15: Time related flakiness (project: nodejs)

```
1 test.expect(7);
2 test.equal(moment.utc().valueOf(), moment().valueOf(), "Calling moment.utc() should ↩
      default to the current time");
```

Listing 4.16: Time related flakiness (project: nodejs)

test outcome is influenced by the ordering of cache operations in other tests.

In our dataset, the number of order-dependent flaky tests in JavaScript projects was noticeably low compared with findings reported in studies of other languages, such as Java [7] and Python [60]. While this suggests that order-dependent flakiness is less common in the JavaScript ecosystem, it also highlights how little is currently understood about its characteristics and root causes in this context. The scarcity of observed cases motivated us to examine this phenomenon more closely. Rather than assuming it is a negligible problem, we continued our investigation to determine how order-dependent flakiness manifests in JavaScript, whether existing detection techniques are effective, and what unique challenges arise in the JavaScript execution model. This provided the foundation for the next phase of the research (Chapter 5), where we focused specifically on detecting and analysing order-dependent flaky tests.

In addition, we categorised 7 commits into multiple causes in which we believe multiple causes may have equal effects on the flaky behaviour. For example, Listing 4.20 from `nodejs/node`[23] assumed that closing the client (which also currently destroys the socket rather than shutting down the connection) would still leave enough time for the server side to receive the stream error. Both the network and the wait for the connection issues have the same impact on the test being flaky. Hence, we categorized this test under *network* and *async wait* categories.

4.3.2 Responses to Flaky Test (RQ1.2)

As we discussed in Section 2.2.3, when developers face a flaky test, they typically respond by modifying the test code, the code under test, or both. In addition, they sometimes ignore the flaky tests by skipping or quarantining them. However, the changes are intended to fix

[23] `https://github.com/nodejs/node/commit/06a43d4dcacc0c39506b8a3a1bb77411add6d9d7`

```
if (accumulated.includes('Error:') && !finished) {
    assert(
      accumulated.includes('ENOSPC: System limit for number ' +
                           'of file watchers reached'),
      accumulated);
```

Listing 4.17: Resource leak related flakiness (project: nodejs)

```
function test_up_multiple(cb) {
  console.error('test_up_multiple');
  if (skipSymlinks) {
    console.log('skipping symlink test (no privs)');
    return runNextTest();
  }
  fs.mkdirSync(tmp('a'), 0755);
  fs.mkdirSync(tmp('a/b'), 0755);
  fs.symlinkSync('..', tmp('a/d'), 'dir');
@@ -432,6 +443,7 @@ function test_up_multiple(cb) {
      if (er) throw er;
      assert.equal(abedabed_real, real);
      cb();
     });
  });
}
```

Listing 4.18: Resource leak related flakiness (project: nodejs)

flaky behaviour, they do not always completely remove the underlying cause of flakiness. In many cases, the modification only reduces the likelihood of the flaky behaviour being triggered rather than eliminating it entirely. In other words, the test may become more stable, but the conditions that led to the flakiness may still exist in the project, and the test might still fail intermittently under certain circumstances.

For each commit, we identify the main strategies developers use to address flaky tests. We categorise them into five strategies. A summary of the distribution of commits based on the response strategies followed is shown in Fig. 4.2.

1. **Fix:** We consider a fix to be a change that aims to remove the source of flaky behaviour and make the test outcome deterministic and consistent across executions. This strategy applies to flaky tests that have already been detected or reported, and whose behaviour has been confirmed through reproduction. We found that 295 commits present an immediate fix of the noted flaky test. This is not surprising, as we mainly mined commits with known flaky tests (that is, flaky tests that have already been detected by developers and therefore were actioned). Listing 4.21 from

```
// Before: tests share the same Yarn cache
exec(
  `node "${yarnBin}" ${cmd} ${args.join(' ')}`,
  { cwd: workingDir, env: { ... } }
);

// After: each test uses its own isolated cache directory
const cacheDir = path.join(workingDir, '.yarn-cache');
exec(
  `node "${yarnBin}" --cache-folder="${cacheDir}" ${cmd} ${args.join(' ')}`,
  { cwd: workingDir, env: { ... } }
);
```

Listing 4.19: Order-dependent flakiness (project: Yarn)

```
server.on('stream', common.mustCall((stream) => {
  stream.on('error', common.mustCall(() => {
  stream.on('close', common.mustCall(() => {
      server.close();
    }));
  req = client.request();
  req.resume();
  req.on('error', common.mustCall(() => {
    req.on('close', common.mustCall(() => {
      client.close();
```

Listing 4.20: Network and async wait related flakiness (project: nodejs)

`nodejs/node`[24] shows a flaky test from the *Hardware* category. As we explained in Section 4.3.1 (Listing 4.13), the test failed due to a set number of clients in the Raspberry Pi. The test was fixed by reducing the number of clients from 100 to 16.

2. **Improve:** In this strategy, the test code or the CUT is changed to decrease the chance of flakiness or to make the code more traceable to fix it later (which makes it technical debt in nature). There are 23 commits in this category. Listing 4.22 from `atom`[25] shows an added delay to the tests to make them less flaky, but they did not fix the root problem. In this case, the Node Sentinel File Watcher (NSFW) occasionally fails to trigger events for changes that occur immediately after starting to watch a folder. To work around this behaviour, the developers introduce a wait period before using the watcher, which improves the test behaviour but does not address the underlying problem.

[24] `https://github.com/nodejs/node/commit/bbf46215489b15a79468ba26b233219bb6a3e41f`
[25] `https://github.com/atom/atom/commit/fca0681c53cdc03fab24c40350f4f65cfec1dd25`

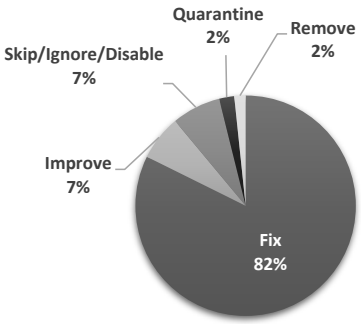

Figure 4.2: Distribution of Response Strategies

```
var N = 10;
var M = 10;
var N = 4;
var M = 4;
server.listen(common.PORT, function() {
  for (var i = 0; i < N; i++) {
    setTimeout(function() {
      for (var j = 0; j < M; j++) {
        http.get({ port: common.PORT, path: '/' }, function(res) {
```

Listing 4.21: Fixing hardware related flakiness (project: nodejs)

3. **Skip/ignore/disable:** This strategy provides developers with the option to *skip* or *ignore* a flaky test. In some cases, especially when the developer is fully aware of the implications of the flaky tests, they may decide to *disable* them and continue with the build as planned. We found 26 commits in total that belong to this category. For example, Listing 4.23 from `electron`[26] shows a test that was flaky on Windows. On this platforms, the timing of when the webview is created, when the page is rendered, and when the theme color is computed and reported is more variable. As a result, the test does not always observe the `did-change-theme-color` event in time. To avoid intermittent failures, developers added a condition to skip the test for "win32" platform or "arm64" architecture.

   The developer noted the following in a pull request[27]:

   > *"...there are a couple of tests that are consistently flaky on arm (both Linux and Windows variants). This PR disables those flakes so that we can get*

[26] `https://github.com/electron/electron/commit/4ce7ca6cfb287b2d260a9e873cfcd4fc975fa15c`
[27] `https://github.com/electron/electron/pull/26046`

```
// In Windows64, in some situations nsfw (the currently default watcher)
// does not trigger the change events if they happen just after start watching,
// so we need to wait some time. More info: https://github.com/atom/atom/issues/19442
await timeoutPromise(300);

await writeFile(subFile, 'subfile\n', { encoding: 'utf8' });
      await firstChanges;
```

Listing 4.22: Fixing hardware related flakiness (project: nodejs)

```
ifdescribe(process.platform !== 'win32' || process.arch !== 'arm64')('did-change-↩
    theme-color event', () => {
 it('emits when theme color changes', async () => {
    loadWebView(webview, {
      src: `file://${fixtures}/pages/theme-color.html`
    });
    await waitForEvent(webview, 'did-change-theme-color');
  });
});
```

Listing 4.23: Skipping flaky test (project: electron)

our ARM CI to the point where it can be relied on for PR validation instead of maintainers ignoring it."

4. **Quarantine:** In this strategy, developers isolate flaky tests from other healthy tests by keeping them in a quarantined area (e.g., a separate test suite or development branch) to be able to diagnose and fix them without disrupting regular workflows. In large repositories, the appearance of new flaky tests can be a daily occurrence, and teams are often unable to address each one immediately. To keep production workflows running smoothly, developers may quarantine tests that exceed a certain flakiness threshold. Quarantined tests still execute during builds, but their outcomes do not influence the pass or fail status of the build. This allows teams to monitor the behaviour of these tests and gather useful diagnostic information without introducing unnecessary delays. Test quarantine is a common strategy used in practice (e.g., [96–99]).

   In the category, there are eight commits that used a quarantine strategy to isolate the flaky tests from other healthy tests. An example of such a strategy is shown in Listing 4.24 from `nodejs/node`[28]. The test becomes flaky at line 10 on CentOS during the `deepEqual` assertion.

[28] `https://github.com/nodejs/node/commit/ac319c354719e02d6b9877f83227ffecabed477d`

```
    options = {
  maxBuffer: 1
};
ret = spawnSync(process.execPath, args, options);

assert.ok(ret.error, 'maxBuffer should error');
assert.strictEqual(ret.error.errno, 'ENOBUFS');
// we can have buffers larger than maxBuffer because underneath we alloc 64k
// that matches our read sizes
assert.deepEqual(ret.stdout, msgOutBuf);
```

Listing 4.24: Flaky test (project: nodejs/node)

In the example, a portion of the test was moved to a separate file. The developer then noted that:

> *"[test] has been flaky on CentOS. This allows us to ... eliminate the cause of the flakiness without affecting other unrelated portions of the test."*

5. **Remove:** The strategy suggests removing all flaky tests from the test suite. We found six commits that removed tests to eliminate the flaky behaviour completely. While removing flaky tests can immediately stabilise test suites, this strategy may reduce test coverage if no alternative validation is introduced, potentially leaving parts of the system untested.

   An example of this strategy is shown in the commit[29] from `nodejs/node`, in which a test was entirely removed from the test suite due to flakiness. The removed test was originally intended to validate an internal debugging feature by observing the creation and cleanup of internal resources during execution. However, in practice, the test asserted the exact lifecycle and ordering of internal handles such as servers, sockets, and DNS lookup requests. These lifecycles are inherently sensitive to differences in Node.js versions, platforms, and runtime scheduling, causing the test to fail intermittently or consistently across environments. As a result, the test did not reliably reflect faults in the runtime itself but rather encoded assumptions about internal behaviour that do not hold universally.

   The developer noted the following:

   > *"[the test] is supposed to test an internal debug feature, but what it effectively ends up testing, is the exact lifecycle of different kinds of internal handles*

[29] `https://github.com/nodejs/node/commit/3143d732f6efd82da76e9c53ad192ac14071bf70`

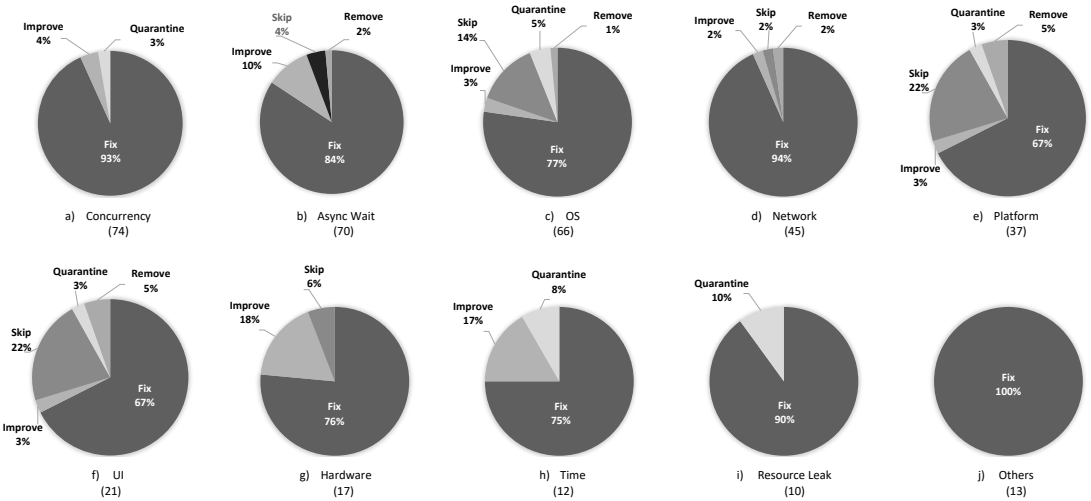

Figure 4.3: Distribution of Flakiness Causes on Strategies (the number of commits are shown in parentheses)

... making the test fail intermittently ... It's not a good test, delete it."

Fig. 4.3 demonstrates the percentages of strategies followed based on the cause of flakiness. The *Fix* strategy represents the largest portion of each category. The most commits under the *Improve* category are related to *async wait*, *hardware*, and *time* causes. Similarly, as expected, most *Skip* actions are taken to deal with *UI*, *OS*, and *Platform*-dependent tests. The reason is that the tests are conditional (run on specific OSs or platforms), so they can run without non-determinism. Also, the largest portion of *Remove* is related to *UI* and *Platform* categories. We list the most common change patterns used to fix the top four causes (concurrency, async wait, OS and network) in Table 4.3.

Regarding where the modifications happen in the project when a flaky test appears, we found that 350 commits deal with flakiness by changing the test code, three changed the CUT, and five changed both the test and the CUT. When developers are faced with a flaky test, they usually make changes in CUT[30], in the test code[31], or both of them[32].

[30] `https://github.com/yarnpkg/yarn/commit/fb40251c5b10bf3b43eed2daaaab53a1aab86f31`

[31] `https://github.com/freeCodeCamp/freeCodeCamp/commit/827bcfaaa2c71f12bc52a4c660e7213c2c426caf`

[32] `https://github.com/nodejs/node/commit/34d1b1144e1af8382dad71c28c8d956ebf709801`

Table 4.3: Common Changes for Four Top Categories

Category	Common Change	Description

Concurrency	`waitForEvent`
	`setTimeout`
	`setInterval`
	`sleep`
	add lock
	increase timeout
		Tests become flaky due to conflicting operations across threads. Developers add waits and locks to prevent race conditions and ensure proper synchronization.

Async Wait	`waitForEvent`
	`async`
	`await`
	`setTimeout`
	`setInterval`
	`timeoutPromise`
	increase timeout
	reordering code
		Developers typically introduce additional waiting time to handle async-related flakiness. Common fixes include increasing timeouts, awaiting events, or reordering code.

OS	add condition
	`async`
	`setTimeout`
		Developers often check OS type or version and run/skip tests selectively. Some flakiness arises from OS-level resource management delays.

Network	`common.port`
	`socket.setTimeout`
	`socket.destroy`
	`socket.end()`
	`client.shutdown()`
		Network-related flakiness often stems from improper socket or port handling, or unstable connections. Developers commonly adjust socket timeouts or close connections correctly.

4.4 Threats to Validity

Missing flaky tests related commits: We used a keyword-based approach to identify commits related to flaky tests. To provide an extended coverage, we used an extended list of 7 flaky test-related keywords in our search string (flaky, fragile, brittle, flakey, non-deterministic test, and intermittent). It is still possible that we could miss either 1) commits that used keywords other than the ones we identified in our research, or 2) flaky tests that have no associated commits (only reported in the issue tracker but have not been reviewed or actioned yet).

Generalisation of the findings: The study considers commits from publicly available JavaScript projects on GitHub. The observations in this study, in terms of flaky test categories, may be limited to the selected projects and may not be generalizable to other JavaScript programs. We mitigated this by selecting popular, highly starred projects managed by large communities and spanning various application domains (server, frontend/backend frameworks, web applications, graphics, etc.). Thus, our sample is somewhat representative of real-world programs. Including additional projects will surely supplement our findings, and this may lead to more generalised conclusions.

Temporal validity: The mining snapshot used to answer RQ1 was collected in 2021, and

JavaScript is a rapidly evolving ecosystem in which frameworks, libraries, runtimes, and development practices change frequently. This may affect the extent to which the exact distribution of flaky-test causes observed in this study reflects the current state of JavaScript projects. The increasing use of AI-based code generation tools may also affect the scale and characteristics of the problem, as generated test code may reproduce existing testing patterns or introduce new sources of unreliability. However, we believe that the main findings remain relevant because the dominant causes identified in RQ1, including asynchronous behaviour, concurrency-related issues, and environment dependencies, are rooted in fundamental characteristics of JavaScript systems rather than being tied to a specific framework or tool version. Moreover, the later studies in this thesis, particularly the investigations of order-dependent and environment-dependent flaky tests, revealed consistent patterns of flakiness, providing additional evidence that these issues continue to affect JavaScript projects beyond the original mining snapshot.

Manual classification of causes: A large part of the study relies on manual analysis of data (commit messages and source code changes), which could affect construct validity due to potential personal oversight and bias. To reduce potential false positives, all identified commits were independently classified by two reviewers, and when there were disagreements about a certain commit, we introduced a conflict resolution step where a third reviewer was then involved to independently classify the commits on which there was disagreement. This occurred for 139 out of 481 commits (28.9%). We followed that with a rigorous discussion of the causes between all reviewers until we reached an agreement (100% agreement level). There were still issues with several commits, and all reviewers agreed that the cause of the flakiness is unknown or cannot be identified; those have been classified as "hard to classify" and excluded from the analysis.

4.5 Summary

In this chapter, we investigated the presence of flaky tests in JavaScript projects, focusing on two aspects: (1) the main causes of flaky tests and (2) the strategies developers use to address them. Using a commit-level analysis of 452 commits from top-starred GitHub repositories, we classified causes based on prior taxonomies [7, 8, 60] and introduced new categories where needed. The distribution of causes (Table 4.2) shows that more than 70% of flaky test commits are due to one of four categories: *Concurrency* (20.7%), *Async Wait* (19.6%), *OS* (18.4%), and *Network* (12.6%). While we report *Async Wait* separately for

clarity, it is a prominent subtype of concurrency and accounts for almost half of concurrency-related cases.

With respect to developer responses, most flaky tests (>80%) are *fixed* to eliminate the nondeterministic behaviour, while a smaller fraction are *improved* (reducing but not fully removing the nondeterminism), *skipped/disabled*, *quarantined*, or *removed* (Fig. 4.2). Mapping responses to root causes (Fig. 4.3) reveals consistent patterns: concurrency and async-wait flakiness are most often addressed through event-based synchronization rather than ad hoc sleeps; OS-related flakiness is frequently handled via conditional execution or environment checks; and network flakiness can be mitigated by better socket and port management.

Our findings yielded three notable observations:

1. **Test order dependency was rare in the commit evidence we analysed.** Unlike previous studies of Java and Python that highlight test order dependency as a major cause of flakiness [7, 60], we observed only a small number of commits linked to test order effects in JavaScript. We investigate order-dependent tests in depth in Chapter 5.

2. **OS dependency was comparatively prominent in JavaScript.** While prior studies on other languages [7,8,60] rarely identified platform issues among the leading causes, our JavaScript corpus shows OS-related behaviour as one of the top three categories (Table 4.2). Many failures were specific to running tests on Windows (e.g., path handling, filesystem event timing), though browser and Node.js versions have also been a source of test nondeterminism. We investigate environmental flakiness further in more detail in Chapter 6.

3. **Developer strategies were pragmatic and locally targeted.** The dominant response was to fix the test or code under test, often by adopting deterministic waits and clarifying environment assumptions; however, skip/disable and quarantine are also used to maintain the CI pipeline when immediate fixes are impractical.

MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
DOCTORATE WITH PUBLICATIONS/MANUSCRIPTS

We, the student and the student's main supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the student's contribution as indicated below in the Statement of Originality.

Student name:	Negar Hashemi
Name and title of main supervisor:	Associate Professor Amjed Tahir
In which chapter is the manuscript/published work?	Chapter 5 – Detecting and Evaluating Order-Dependent Tests

Describe the contribution that the student and members of the supervisory team have made to the manuscript/published work:[1]

The student constructed the study, designed and built the JS-TOD tool, conducted the tool evaluation, analysed the results, and wrote the draft of the paper.
The supervisors contributed to the design of the study, the evaluation strategy, interpretation of results, and reviewing and editing of the paper.
Student (Negar Hashemi): 70%
Supervisors and co-authors: 30%

Please select one of the following three options:

- (●) **The manuscript/published work is published or in press**

  Please provide the full reference of the research output:

  Negar Hashemi, Amjed Tahir, Shawn Rasheed, August Shi, and Rachel Blagojevic, "Detecting and Evaluating Order-Dependent Flaky Tests in JavaScript," 2025 IEEE Conference on Software Testing, Verification and Validation (ICST), Napoli, Italy, 2025, pp. 13-24.

- ( ) **The manuscript is currently under review for publication**

  Please provide the name of the journal:

- ( ) **It is intended that the manuscript will be published, but it has not yet been submitted to a journal**

Student's signature:	Negar Hashemi Digitally signed by Negar Hashemi Date: 2026.01.26 23:05:11 +13'00'	Main supervisor's signature:	Amjed Tahir Amjed Tahir 2026.01.29 07:49:04 +03'00'

This form should be placed at the beginning of each relevant thesis chapter.

[1] Refer to the Massey University Publishing and Authorship guidelines (OneMassey for staff, Stream for students) and/ or Contributor Roles Taxonomy (CRediT) guidelines for guidance.

MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
DOCTORATE WITH PUBLICATIONS/MANUSCRIPTS

We, the student and the student's main supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the student's contribution as indicated below in the Statement of Originality.

Student name:	Negar Hashemi
Name and title of main supervisor:	Associate Professor Amjed Tahir
In which chapter is the manuscript/published work?	Chapter 5 – Detecting and Evaluating Order-Dependent Tests

Describe the contribution that the student and members of the supervisory team have made to the manuscript/published work:[1]

The student constructed the study, extended and refined the JS-TOD tool, conducted the experimental evaluation, analysed the results, and wrote the draft of the paper.
The supervisors contributed to the design of the study, interpretation of the results, and reviewing and editing of the paper.
Student (Negar Hashemi): 70%
Supervisors and co-authors: 30%

Please select one of the following three options:

- (●) **The manuscript/published work is published or in press**

  Please provide the full reference of the research output:

  Negar Hashemi, Amjed Tahir, Shawn Rasheed, August Shi, and Rachel Blagojevic. "JS-TOD: Detecting Order-Dependent Flaky Tests in Jest," Journal of Science of Computer Programming (2026), 103462.

- ( ) **The manuscript is currently under review for publication**

  Please provide the name of the journal:

- ( ) **It is intended that the manuscript will be published, but it has not yet been submitted to a journal**

Student's signature:	Negar Hashemi Digitally signed by Negar Hashemi Date: 2026.01.26 23:04:53 +13'00'	Main supervisor's signature:	Amjed Tahir Amjed Tahir 2026.01.29 07:48:51 +03'00'

This form should be placed at the beginning of each relevant thesis chapter.

[1] Refer to the Massey University Publishing and Authorship guidelines (OneMassey for staff, Stream for students) and/ or Contributor Roles Taxonomy (CRediT) guidelines for guidance.

Chapter 5

Detecting and Evaluating Order-dependent Tests

5.1 Introduction

In Chapter 2, we explained how test order dependency arises when the outcome of a test is influenced by the execution order of other tests, often due to hidden dependencies such as shared state between tests or improper cleanup. In practice, these issues can manifest in subtle ways, such as global variables not being cleared, side effects that persist across executions, or shared resources left in an unexpected state. These behaviours are usually not visible during normal, fixed-order execution, which makes order-dependent tests difficult for developers to notice and diagnose. As a result, they can introduce nondeterministic failures, reducing trust in the test suite and increasing the effort required for debugging and maintenance.

The findings of the empirical study presented in Chapter 4 demonstrated that order dependency appeared less frequently in JavaScript projects compared to what has been reported in other languages, notably Java [7] and Python [60], where it has been noted as the most common cause of test flakiness in those languages. This raises the question of whether order dependency is genuinely rare in JavaScript or simply harder to identify through commit analysis alone. However, these findings were based on the reported flaky tests in open-source projects (i.e., extracted from issue trackers). The projects may contain order-dependent tests that developers have yet to discover, which could lead to failing tests and mislead developers later in development.

Indeed, the most popular JavaScript testing framework, Jest [102, 113], introduced a

feature that allows developers to randomize the order in which they run Jest tests due to concerns from developers about the potential of having order-dependent tests [114]. This feature is still relatively new (made available in late 2023) and not widely adopted, meaning many projects may still contain order-dependent tests that developers are unaware of.

This chapter systematically investigates test-order dependencies in JavaScript projects. We aim to better understand how order dependency manifests in practice, what causes it, and why it remains a challenging source of flakiness for JavaScript developers.

In the chapter, we address the following main research question:

RQ2: How can we detect and manifest order-dependent flaky tests in JavaScript projects?

As mentioned in Chapter 3, to address this question comprehensively, we divided it into two sub-questions:

RQ2.1. How can order-dependent tests manifest their failures in JavaScript projects? This question aims to investigate ways that order-dependent tests' failures can manifest in Jest. Understanding how order dependency manifests is important because these failures are often subtle, inconsistent, and difficult for developers to recognise without specific tooling. By identifying the different ways in which order-dependent tests can fail, we can better characterise the nature of order-dependent behaviour in JavaScript test suites and provide more precise guidance for detection and debugging.

RQ2.2. What are the main causes of order-dependent tests? This question aims to understand the main causes of order-dependent tests in Jest. Identifying these causes is essential for building a deeper understanding of why order dependency arises in JavaScript and how it differs from other languages. Such insights help us identify which patterns in test code or code under test are most prone to creating order-dependent behaviour, and they support the design of more effective prevention and mitigation strategies.

5.2 Background

This section presents important background information. We discuss order-dependent tests and Jest in more detail and outline the concepts, behaviours, and testing mechanisms that are relevant to our analysis. This background clarifies how order dependency arises in practice and how Jest executes tests, which helps to contextualize the results presented later in the chapter.

5.2.1 Test Order-Dependency

Ideally, tests should function independently of each other and produce the same results regardless of their execution order [115], but they will often have different outcomes when executed in different orders [18,115]. These tests are called *order-dependent tests*. Building on the general definition of flaky tests in Chapter 1, an order-dependent test is defined as a test whose outcome changes when the execution order of tests within the same suite is altered, even though neither the test code nor the CUT has been modified [18,79]. Previous studies have shown that these tests account for a substantial portion of flaky behavior. Luo et al. [7] reported that 12% of flakiness in their study was caused by order dependency, which often arises when tests share mutable state in memory or through external resources such as files, databases, or configuration data. Similarly, Gruber et al. [60] observed that order dependency is a much more dominant problem in Python, causing 59% of the flaky tests in their dataset.

Order-dependent tests have been classified into two categories based on their result when run in isolation: victim or brittle [52]. A *victim* passes when run in isolation but fails when run after another test(s), known as a *polluter*. Essentially, a polluter "pollutes" the state shared between the two tests, leading the victim to run in an improperly initialized state, resulting in a failure. On the other hand, a *brittle* test fails when run in isolation but passes when run after another test(s), called a *state-setter*. The state-setter sets the initial state that the brittle needs to start in to pass; otherwise, the brittle fails because it cannot run properly on its own.

Listing 5.1 shows an example of an order-dependent test from "react-testing-library"[1]. Test `first` is a state-setter (because it renders the component in line 2) for test `second`, which is a brittle test. Running the test `second` before the test `first` or in isolation will result in a test failure as the document does not have the proper body to be checked.

```
1 test('first', () => {
2   render(<div>hi</div>)})
3 test('second', () => {
4  expect(document.body.innerHTML).toEqual('<div><div>hi</div></div>')})
```

Listing 5.1: Order-dependent tests from "react-testing-library"

Another example of order-dependent tests is Listing 5.2. Test `writeTest` is a state-setter because it creates and writes to a file. Test `readTest` is a brittle test that relies

[1] https://github.com/testing-library/react-testing-library/blob/main/src/__tests__/auto-cleanup-skip.js

on the file being present and containing the expected content. Running `readTest` before `writeTest`, or running it in isolation, will result in a failure because the file has not yet created.

```
1  const fs = require('fs');
2  const path = require('path');
3  const filePath = path.join(__dirname, 'temp.txt');
4
5  test('writeTest', () => {
6  fs.writeFileSync(filePath, 'hello world');
7  });
8
9  test('readTest', () => {
10 const content = fs.readFileSync(filePath, 'utf8');
11 expect(content).toBe('hello world');
12 });
```

Listing 5.2: Order-dependent tests caused by file operations

5.2.2 Jest and Test Order-Dependency

Jest is the most popular JavaScript testing framework [113, 116] due to its ease of use and minimal configuration requirements. In Jest, a *test suite* corresponds to a JavaScript file that contains the tests to be run. A test suite may encompass multiple testing blocks. A developer may choose to use `describe` blocks to group together multiple related tests [117], introducing a level of organization for their tests. The `test` and `it` functions define individual tests within the test suites, each tasked with verifying a specific scenario. The `describe` block allows for hierarchical organization, while the `test` function outlines the specifics of each case, contributing to a formalized and systematic testing approach. Note that individual tests do not always need to be contained with a `describe` block, but if there are `describe` blocks, then tests within the same `describe` block are executed as a unit without interleaving with tests in other `describe` blocks.

Listing 5.3 demonstrates how Jest organises tests within a single test suite. The file itself represents one test suite and contains both individual tests and two separate `describe` blocks. The first test is defined outside any `describe` block, showing that standalone tests are allowed. The two `describe` blocks group related tests together: one focuses on string operations and the other on array behaviour. Each test is defined using the `test` function. This illustrates the hierarchical structure and organisational flexibility that Jest provides.

```
// A test outside any describe block
test('adds numbers', () => {
  expect(1 + 2).toBe(3);
});

describe('string utilities', () => {
  test('converts to uppercase', () => {
    expect('hello'.toUpperCase()).toBe('HELLO');
  });

  test('trims whitespace', () => {
    expect(' hi '.trim()).toBe('hi');
  });
});

describe('array behaviour', () => {
  test('checks length', () => {
    expect([1, 2, 3]).toHaveLength(3);
  });

  test('includes element', () => {
    expect([1, 2, 3]).toContain(2);
  });
});
```

Listing 5.3: Jest test file structure

Jest uses different approaches to running tests and test suites. By default, Jest runs test suites in parallel rather than sequentially. By doing so, Jest does not guarantee the running order of test suites[2]. In contrast, Mocha runs test suites sequentially by default[3]. Jest assigns a worker to run each test suite. A developer can use the option `maxWorkers` to limit the number of workers or use the `runInBand` option to have the test suites run sequentially. Jest determines, based on previous runs, if it will be faster to run test suites sequentially rather than in parallel without informing the developer[4]. It also orders the test suites to run based on past runtime, failure history, and/or file size (if no prior information is available) [102]. Jest uses a default test sequencer that orders test suites using information from previous runs. Suites that previously failed are run first, followed by longer running suites, and if no prior data is available, suites are ordered by file size as a heuristic for runtime[5]. Since these factors can change at any time, Jest may arbitrarily determine to

[2] https://github.com/jestjs/jest/issues/4032#issuecomment-424427942

[3] https://mochajs.org/#running-mocha-tests

[4] https://github.com/jestjs/jest/blob/1a487c1803124c594bdd86ee24b5949025660bc3/packages/jest-cli/src/test_scheduler.js#L88-L96

[5] https://github.com/jestjs/jest/blob/6b8b1404a1d9254e7d5d90a8934087a9c9899dab/packages/je

run test suites sequentially and in some non-deterministic order, potentially leading to order-dependent tests that fail when Jest happens to run them in a failing order.

On the other hand, Jest runs tests within test suites in the order they appear in the file (i.e., the first test in the file runs first). If there are `describe` blocks to group tests, Jest runs them in the order they are defined within the file as well. Jest runs all the `describe` blocks handlers in a test suite before executing any actual tests [118]. Listing 5.4 demonstrates how Jest order of execution works[6]. When Jest executes this file, it first runs all the `describe` callbacks during the collection phase. This is why all the console outputs inside the `describe` blocks appear first and in the order they are encountered: outer block statements, then inner `describe` blocks, and finally the remaining outer statements. Only after completing this phase does Jest execute the actual tests, leading to the outputs from `test 1`, `test 2`, and `test 3` appearing at the end, in the order the tests were registered.

Jest has multiple options for test parallelization and selection [119]. For example, the `testNamePattern` option allows running only individual tests or `describe` blocks that match a specific naming pattern. The `shard` option allows splitting a test suite into multiple "shards" and then running specific shards, which is useful for parallelizing test execution across multiple machines or environments to reduce the overall test runtime (each shard runs on different machines). The `randomize` option was released in late 2023 in response to a popular feature request from developers [114], who noted the need for test randomization common in other testing frameworks such as JUnit and NUnit. From the discussion of this feature, we observed that developers saw that randomizing test execution can help uncover cases where tests inadvertently depend on the state left by previous ones.

In Jest, test dependencies can manifest at various levels. Order dependency may arise between tests, `describe` blocks, and test suites. Fig. 5.1(a) illustrates an example of order dependency between two tests. Changing the order of `testA.1.1` and `testA.1.2` results in the failure of `testA.1.2`, as it depends on `testA.1.1`. The order dependency between `describe` blocks is demonstrated in Fig. 5.1(b), where running `describeA.2` before `describeA.1` leads to the failure of `describeA.1`. Similarly, an axample of order dependency between two test suites is shown in Fig. 5.1(c): `testA.test.js` and `testB.test.js`. The passing order for the two tests is `testA.test.js` → `testB.test.js`. By changing the running order to `testB.test.js` → `testA.test.js`, `testB.test.js` will fail.

st-core/src/TestSequencer.ts#L70

[6] https://jestjs.io/docs/setup-teardown?utm_source=chatgpt.com#order-of-execution

```
    describe('describe outer', () => {
  console.log('describe outer-a');

  describe('describe inner 1', () => {
    console.log('describe inner 1');

    test('test 1', () => console.log('test 1'));
  });

  console.log('describe outer-b');

  test('test 2', () => console.log('test 2'));

  describe('describe inner 2', () => {
    console.log('describe inner 2');

    test('test 3', () => console.log('test 3'));
  });

  console.log('describe outer-c');
});

// describe outer-a
// describe inner 1
// describe outer-b
// describe inner 2
// describe outer-c
// test 1
// test 2
// test 3
```

Listing 5.4: Example of order of execution from Jest document

5.3 Approach

We proposed a test reordering approach based on the structure of test suites, as described in Section 5.2.2, to answer our research questions. To reorder tests at all three levels (Fig. 5.1), We use a randomization test reordering approach (Algorithm 1). The function `randomizeOrder` takes as input an array, representing the set of test suite paths, tests, or `describe` blocks to be reordered, and an integer, representing the number of reorders. The function returns the set of reorders. If all possible permutations of items are smaller than the provided number of reorders, the function returns all permutations. Otherwise, it shuffles the items to create a new reorder and adds it to the set of reorders to be returned if it is unique. This process repeats until the specified number of reorders is reached. Different approaches can be used to reorder test suites, tests, and `describe` blocks, as detailed below.

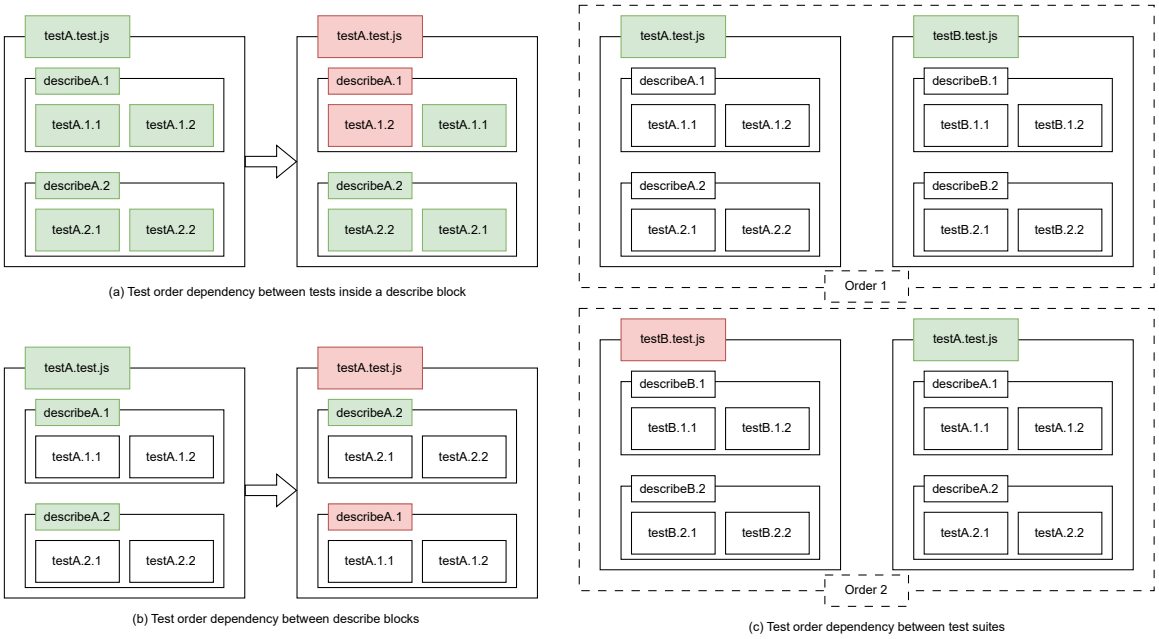

Figure 5.1: Different levels of test order dependency with passed tests in green and failed tests in red

5.3.1 Reordering Test Suites

For reordering test suites (Fig. 5.1(c)), we need to extract the paths of test suites from each project and pass the running order to Jest. We extract the test suites' paths for a given project automatically using Jest's `-listTests` option [119].

Then, we pass the test suite paths to the `randomizeOrder` function in Algorithm 1 to generate up to `reorder_number` unique reorders. In addition, Jest uses a `TestSequencer`[7] class to determine the order in which to run the test suites. We write a custom sequencer that extends this `TestSequencer` class, allowing us to run the test suites in a specific order. Listing 5.5 shows our custom sequencer, which receives the running order as an argument and returns tests in the defined order. We provide the custom sequencer and running order to Jest by passing the customSequencer' path after the `-testSequencer` option and order of test suites paths after the `-order` option, e.g., `npxjest--runInBand--testSequencer='/path/to/customSequencer.js'--order='/path/to/test1.js,/path/to/test2.js'`.

[7] `https://jestjs.io/docs/configuration#testsequencer-string`

Algorithm 1 Tests Reordering Approach

```
1: function RANDOMIZEORDER(array, reorder_number)
2:     all ← calculateAllPossibleMutation(array_lenght)
3:     if reorder_number ≥ all then
4:         result ← getPermutation(array)
5:     else
6:         while result_length ≤ reorder_number do
7:             newArray ← shuffleArray(array)
8:             if isUnique(newArray) then
9:                 result ← newArray
10:            end if
11:        end while
12:    end if
13:    return result
14: end function
```

5.3.2 Reordering Tests and `describe` blocks

Algorithm 2 describes the process for extracting and reordering both tests and `describe` blocks (Fig. 5.1(a),(b)). After extracting test suites, we use the Babel toolchain[8], a JavaScript compiler and source-code transformer to parse each test suite. Babel generates an Abstract Syntax Tree (AST) for each test suite (including nodes, parameters and types). Using the AST, we identify all `describe` blocks within each test suite of a given project. The `describe` blocks are recognised by recording AST nodes when a node's type is "*identifier*" and its name is "*describe*". These blocks are then reordered and saved in new test files in the original directory.

For identifying tests inside each `describe` block, we use the same randomizing approach as for Algorithm 2. The difference is that the identifier names of these AST nodes are either `it` or `test`.

We record the results of each run in a JSON file for further investigation. In cases where a test suite consistently fails across all reruns of the same order, we categorize it as an order-dependent test. Otherwise, if it passes and fails even as the order remains the same, the test suite is flaky, depending on other non-deterministic factors.

[8] `https://babel.dev`

Algorithm 2 Extracting and Reordering `describe` blocks/Tests in `JS-TOD`

```
1:                                                          ▷ Input: a project path
2:                ▷ Output: results of running in JSON format saved in the project path.
3: INSTALL_PROJECT(project_path)
4: testSuites ← ListTestSuites(project_path)
5: for each testSuite in testSuites do
6:     blocks ← PARSE(testSuite)
7:     if blocks_number ≥ 2 then
8:         newTests ← randomizeOrder(blocks, reordernumber)
9:         for each newTest in newTests> do
10:            newTestCode ← generate_code(newTest)
11:            generate_name_for_newTestCode
12:            save_newTestCode
13:        end for
14:    end if
15: end for
16: for each newTestSuite in newTestSuites do
17:    RUN(newTestSuie_path, rerun_number)
18: end for
19:
20: function PARSE(testSuite)
21:    ast ← babel_parser(testSuite)
22:    blocks ← TRAVERSE(ast, type)
23:                                                  ▷ type=[describe blocks, tests]
24:    return blocks
25: end function
```

```
class CustomSequencer extends TestSequencer {
  sort(tests) {
    //get the running order and put it in orderPath array
    const orderPathString = process.argv.find ((arg)=>arg.startsWith("--order")).↩
        replace("--order=","");
    const orderPath = orderPathString.split(",");
    return tests.sort((testA, testB) => {
      const indexA = orderPath.indexOf(testA.path);
      const indexB = orderPath.indexOf(testB.path);

      if (indexA === -1 && indexB === -1) return 0; // Keep both tests in their ↩
          original order if not specified in orderPath
      if (indexA === -1) return 1;
      if (indexB === -1) return -1;
      const result = indexA - indexB;
      return result;
    });}
}
```

Listing 5.5: Custom Sequencer Class

5.3.3 Alternative methods

We experimented with different methods to detect order-dependent tests, to increase the likelihood of revealing order-dependent flakiness. For instance, we attempted to run the test suites in parallel to introduce scheduling nondeterminism and to increase the chances of overlapping setup and teardown phases across tests. To do this, we created a custom sequencer that randomly selected different numbers of test suites and executed them multiple times. Another approach was to run each test in isolation to determine whether interactions with other tests caused observed failures. By executing tests independently, we aimed to distinguish order-dependent flakiness from failures intrinsic to individual tests. To achieve this, we used two isolation mechanisms provided by Jest: 1) running tests with the Jest's `test.only` option, 2) running tests by matching name patterns using the `testNamePattern` option. In addition to Jest options, we placed each test in its own individual test suite and ran all test suites in parallel. For each of these methods, we reran the tests 10 times. The results showed that our proposed randomization approach (explained in Algorithm 1) can reveal order-dependent tests with less overhead than the other approaches. The alternative approaches we experimented with did not yield any new order-dependent tests, and in most cases they failed to detect those identified by the randomization method.

5.4 Implementation

We implemented our test reordering approach in a tool called JavaScript Test Order-dependency Detector (`JS-TOD`). Similar to flaky tests detection tools such as iDFlakies [55] and FlaPy [66], `JS-TOD` detects order-dependent behaviour by reordering and rerunning test suites and recording the outcomes. However, unlike existing tools that target Java or Python and often require additional steps, such as automated classification or repair, `JS-TOD` is specifically designed for the JavaScript ecosystem and the Jest testing framework. It provides a lightweight, configurable, and framework-integrated solution, addressing a gap where no prior detection tool exists for JavaScript or Jest.

`JS-TOD` reveals order-dependent behaviour by extracting, reordering, and rerunning tests according to a user-defined number of permutations and reruns. It does not perform automated classification or repair but is intended as a diagnostic aid for developers. When failures occur in certain execution orders, `JS-TOD` provides the corresponding failing permutations and error logs, enabling developers to manually investigate and pinpoint the underlying causes.

In practice, it is most effective when integrated into regression testing or continuous integration pipelines to expose latent order dependencies that may not appear under default configurations. Users can specify the reordering level, the number of permutations to generate, and the number of reruns. `JS-TOD` also saves the newly generated test orders as separate test suites for deeper analysis. Unlike Jest's `randomize` option, which shuffles test execution order within a suite using a seed value, `JS-TOD` performs systematic, reproducible reordering across three levels: test suites, `describe` blocks, and individual test blocks.

Figure 5.2 illustrates `JS-TOD` approach. For a given project with Jest test files, `JS-TOD` first extracts the test data of a project based on the specified reordering level. It then reorders and reruns the newly added test files for the given numbers. The result of each rerun is saved in a JSON file.

5.4.1 Using JS-TOD in CI/CD Pipelines

To integrate `JS-TOD` into a CI/CD workflow, a user will need to first navigate to the directory containing the tool:

```
cd /path/to/JS-TOD
```

The user will need to ensure that the required dependencies are installed and that the

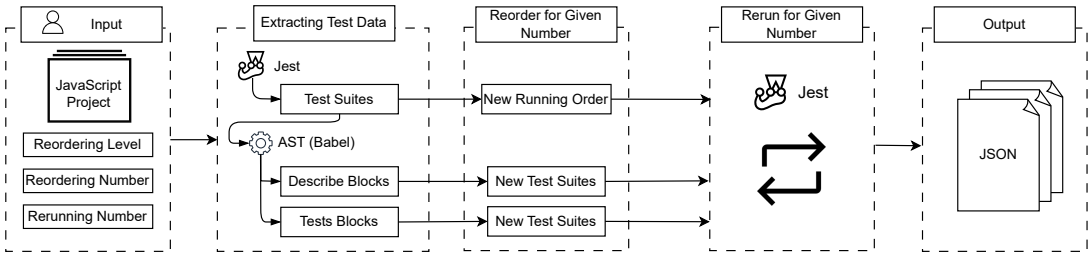

Figure 5.2: JS-TOD Approach

environment uses `Node.js (version ≥ 18.17.0)` or higher.

```
npm install
```

Then, determine the desired reordering level, `test`, `describe`, `suite`, or `all` (which executes all three levels sequentially), and provide the path to the project under analysis. JS-TOD can be invoked using its main entry point, `reorderRunner.js`, as follows:

```
node reorderRunner.js \
  --project_path="/path/to/project" \
  --level="<test|describe|suite|all>" \
  --reorder=<number_of_permutations> \
  --rerun=<number_of_reruns>
```

- `project_path` defines the root directory of the target project containing the Jest test suites.
- `level` specifies the granularity of reordering. The options are `test` (individual tests), `describe` (groups of tests), `suite` (entire test files), or `all` (runs all levels in sequence). The default is `all`.
- `reorder` sets how many distinct permutations of test order should be generated and executed. The default is 10.
- `rerun` determines how many times each permutation should be executed to confirm test stability. The default is 10.

`JS-TOD` is publicly available on GitHub[9], and the dataset can also be accessed online[10].

[9] `https://github.com/Negar-Hashemi/JS-TOD`
[10] `https://zenodo.org/records/13852085`

5.4.2 Evaluation

We evaluated `JS-TOD` in two stages: (1) whether it could correctly identify the test suites in each project, and (2) whether it could correctly generate reordered test suites. The first stage evaluates the test-path extraction step. `JS-TOD` relies on Jest's `--listTests` option to obtain the paths of test suites in a project. If this step succeeds, `JS-TOD` can identify the test files that should be used for reordering. In this experiment, `JS-TOD` returned the correct test paths for 73 out of 81 projects (90%). The remaining projects used older versions of Jest or configurations that did not support the required test-listing functionality.

The second stage evaluates the reordering step. After the test paths are identified, `JS-TOD` parses the test files, extracts tests or `describe` blocks, and generates reordered test suites. We considered this step successful when `JS-TOD` produced syntactically valid reordered test files that could be executed by Jest without introducing transformation-related errors. Among the projects for which reordering was applicable, `JS-TOD` correctly reordered tests in 59 out of 69 projects (85%). Failures in this stage were mainly due to parsing or transformation issues in real-world JavaScript test files, such as unsupported syntax or project-specific test structures.

5.4.3 Limitations

`JS-TOD` extracts test file paths using Jest's `--listTests` option, which is available starting from Jest version 20.0.0[11]. This option allows `JS-TOD` to systematically discover and process all test files in a project, regardless of their structure. However, this introduces a limitation: projects using Jest versions older than 20.0.0 do not support the `--listTests` option and, therefore, cannot use `JS-TOD` in its current form. In addition, we introduced the sequencer path to Jest using our custom sequencer class and the `--testSequencer` option, which is available from Jest version 24.7.0[12]

`JS-TOD` relies on Babel to parse test suites and construct an abstract syntax tree (AST) that is used to identify and reorder tests. In some cases, however, certain statements in the original test files are not correctly captured during AST traversal. When this occurs, these statements are omitted from the reordered test suites, which can lead to failing executions after reordering. This limitation highlights the challenges of accurately analysing and transforming real-world JavaScript test code, where variations in syntax or module-loading patterns may not always be handled uniformly by the underlying parsing infrastructure.

[11] `https://github.com/jestjs/jest/releases/tag/v20.0.0`
[12] `https://github.com/jestjs/jest/releases/tag/v24.7.0`

Additionally, `JS-TOD` depends on modern ECMAScript features and Jest's programmatic APIs (e.g., `testSequencer`), which may differ across major releases. Although this is unlikely in controlled environments, differences in Node.js or Jest versions may lead to reduced functionality or occasional execution failures. This is especially true in CI pipelines that use different base images or cached dependencies. These issues can be easily resolved by using an environment similar to the one validated in our evaluation (Node.js 18.16.1, npm 9.5.1, and Jest version 27 or higher).

Another limitation of `JS-TOD` is that reruns are executed sequentially. If a developer configures `JS-TOD` to perform many reorderings or reruns, the process may become time-consuming, particularly for large-scale projects with extensive test suites. Furthermore, the current version of `JS-TOD` randomizes a subset of tests, `describe` blocks or test suites when generating new execution orders. While this approach increases variability and helps uncover many order-dependent behaviors, it does not guarantee comprehensive coverage of all possible order interactions. More systematic ordering strategies, such as those based on Tuscan squares [65] or NDOD/NDOI sampling methods [120], can be used to improve the representativeness of selected test orders. Integrating such combinatorial designs in future versions of `JS-TOD` could enhance the detection efficiency and reduce redundant executions.

5.5 Experimental Setup

To investigate order-dependent tests and answer the research questions, we constructed a dataset of open-source JavaScript projects. We then reordered each test suite, test, and `describe` block for 10 reorders and reran each reordered test suite 10 times using `JS-TOD`. Finally, we manually analysed the recorded results. Fig. 5.3 shows an overview of our experiment, including data collection and analysis processes.

5.5.1 Dataset

We conducted our experiment using a dataset of JavaScript projects obtained from GitHub. Similar to the approach used in Chapter 4, we utilised GitHub's REST API[13] to search for projects. We used a keyword search to collect projects that contain "jest" in the repository (e.g., topics, issues, and README files). In addition, we searched for JavaScript projects without any special keywords. In both search queries, we order the projects based on GitHub's star rating.

[13] `https://docs.github.com/en/rest/reference/search`

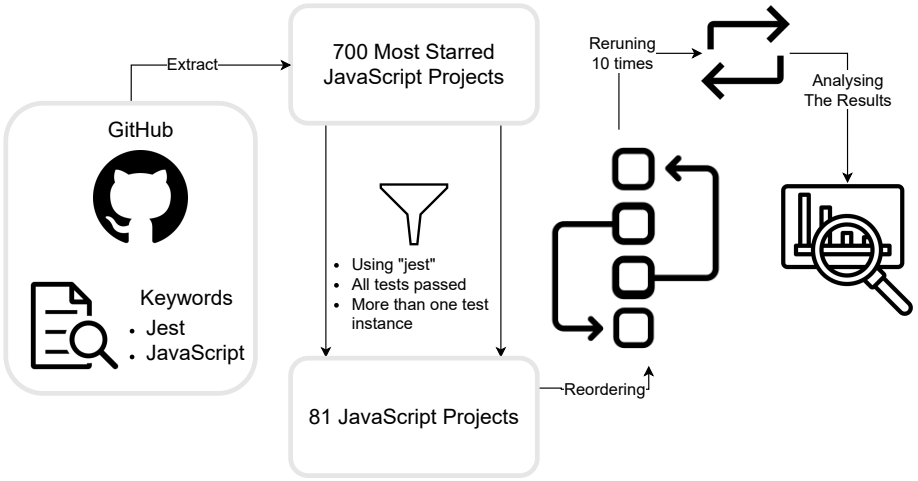

Figure 5.3: An overview of the data collection and analysis process

Based on our search results, we selected the 700 most starred projects to form a large enough initial dataset for inspection. After cloning all 700 projects, we select the projects that use Jest as their testing framework by searching with the keyword `jest` in the `package.json` file. We found 304 projects that use Jest as their testing framework.

For each project, we execute all tests in their default order (i.e., without changing the original test order as they appear in the project). After each execution, we recorded the number of test suites, `describe` blocks, and tests for further investigation. We then excluded projects that do not have enough test instances to investigate (a project must contain at least two tests so we can reorder them). We found that 82 of the 304 selected projects have more than one test instance (at least two tests, `describe` blocks, or test suites) and all passing tests when run in the default order. We excluded all other projects from our evaluation.

When we applied `JS-TOD`, we find a project ("snapshot-diff") where the reordering does not work. The project has a test with a missing semicolon, which causes Babel to fail when transforming the tests. Hence, we excluded this project from our evaluation and continue with the remaining 81 projects. Table 5.1 shows summary statistics of the selected projects.

5.5.2 Environment

We conducted our evaluation on a machine with a 2.3 GHz Intel Core i9 processor with 16 GB RAM running on macOS 14. We use `npm` version 9.5.1 and `Node.js` version 18.16.1.

Table 5.1: Project Statistics

	Production		Tests			
	LOC	#files	LOC	#tests	#describe blocks	#test suites
Mean	1791	32	690	79	18	11
Total	145092	2583	55911	6398	1420	873

The experiment took approximately 121 hours to run across all projects and reruns required (see Table 5.2 for runtime data).

5.5.3 Manual analysis

We conducted two different manual analyses on the collected results. We carried out our first manual analysis on the tests before reordering anything. The goal is to check if all tests of a project pass by default, that all test paths are available and correctly extracted by Jest's `-listtests` option (available from Jest version 20.0.0[14]). In eight projects, the *-listTests* option could not list the test paths due to using older versions of Jest. In those projects, we manually extract the test paths to pass to `JS-TOD`, allowing us to evaluate on those projects.

After reordering the tests, we conducted a second manual analysis to verify the cause of failures for failing tests. One of the coders manually inspected all results, examining the source code and error messages to identify instances of failed orders. After identifying the cause of failure, a second coder independently inspected the code to confirm if it was due to an order-dependent test. The coders discussed cases where there were disagreements, and in one case (1.8%), a third coder confirmed the cause of flakiness. Our goal is to create a categorization of common reasons for flakiness, namely the types of shared state between tests, among the detected order-dependent tests.

We make publicly available a full replication package for our evaluation, which includes all data, scripts and full results [121].

5.6 Results

To detect order-dependent tests in JavaScript projects, we applied `JS-TOD` on the tests within each of the 81 projects we evaluated, reordering tests at the three levels of tests, `describe` blocks, and test suites. Our goal is to uncover tests whose outcomes change when

[14] `https://github.com/jestjs/jest/releases/tag/v20.0.0`

the execution order is altered. To achieve this, `JS-TOD` systematically reorders tests at three levels, tests, `describe` blocks, and test suites, and reruns the modified suites to observe whether any test behaviour changes across executions. This approach allows us to capture different forms of order dependency that may arise from interactions across tests, across `describe` blocks, or across test suites. In this section, we report the results in relation to the research questions.

5.6.1 Manifesting Order-Dependent Tests (RQ2.1)

Across the evaluated projects, we observed different forms of order-dependent behaviour emerging at the test, `describe`, and suite levels. Some projects exhibit order dependency only at a single level, while others appear in multiple levels, indicating interactions that manifest differently depending on the granularity of reordering. Table 5.2 shows an overview of the results of applying `JS-TOD` on each evaluation project. The total number of projects in Table 5.2 is the summation of all projects in the three different levels, noting that some projects appear in more than one level.

In total, we find 55 order-dependent tests across 10 projects. All 55 tests fail across all 10 reruns of the same order, confirming that they indeed are order-dependent tests. The majority of these order-dependent tests were found by reordering at the level of tests (52 tests). Only three order-dependent tests were found at the level of `describe` blocks. The results for each level are discussed individually below.

Order dependency between tests. From all 81 projects, we find that 12 of those projects contain only a single test inside each `describe` block, and thus, we excluded those 12 from any further analysis. We also encountered several issues when running `JS-TOD` on some remaining projects. For example, running `JS-TOD` on "jest-serializer-vue" was unusually time-consuming, taking more than six hours compared to the average runtime of less than three hours for other projects. In addition, "draggable" has a test with a missing semicolon, which causes Babel to fail when transforming the test code. Therefore, we excluded these 14 projects and continued with the remaining 67 projects for the test-level analysis.

Our analysis reveals a total of 52 order-dependent tests from nine projects. As shown in Table 5.3, most of the flaky tests appear in project "jest-webextension-mock" with 35 order-dependent tests, followed by "serverless-jest-plugin", which has five order-dependent tests and "native-testing-library" with four order-dependent tests. The rest of the six projects contain one or two order-dependent tests.

Order dependency between `describe` blocks. Here, we investigate whether order dependencies among different `describe` blocks can cause flakiness. We observed that 47

out of the 81 projects have no or just a single `describe` block inside each test suite, and therefore, it was not possible to reorder the `describe` blocks for such projects. For example, "electron-typescript-react" has only a single `describe` block in each test suite.

In addition, "draggable" has a test with a missing semicolon that causes Babel to fail in transforming the project's tests. As a result, we continue with the 33 projects for further analysis.

In some cases, the `describe` blocks can be nested (i.e., a `describe` block can contain other `describe` blocks within its structure). For the 33 remaining projects, we reorder the `describe` blocks up to two nested levels. We limited the reordering to two levels because this depth covered all but one project in our dataset and allowed us to systematically control and rewrite the test structure without introducing unintended changes to the test logic. Handling deeper nesting introduced technical issues in reliably rewriting and reconstructing the test hierarchy. Only a single project, "Jasmine-Matchers" [15] has more than two nested blocks, and we could not find a suitable way to reorder its blocks, as there were some technical challenges (related to Babel) in rewriting those tests, and therefore we exclude this project from our analysis. Ultimately, we analysed 32 suitable projects (each containing more than one `describe` block).

Among the 32 projects, we found one project, "jest-build-ins", with two order-dependent `describe` blocks, namely "native jest mock function inspectors" and "exposing mock internals".

Order dependency between test suites. We only analysed projects that contained multiple test suites to determine order dependency between test suites. We identified 62 out of 81 projects that contained more than one test suite, and thus were suitable for this analysis. In 13 of the 62 projects, we encountered an issue where the `-testSequencer` was unrecognized in the `npx` command due to using older versions of Jest. Since we rely on Jest's test sequencer class to enforce a specific running order, we cannot reorder the tests for these projects. All test suites pass for the remaining 49 projects regardless of their test-suite order. Thus, we do not find evidence of flakiness due to order dependency between test suites.

[15] `https://github.com/JamieMason/Jasmine-Matchers`

Table 5.2: Results of Reordering and rerunning Tests in Three Levels

	Default Order			Reordering Result						
Reordering Level	#projects	#tests	#test suites	#test	#test suites	#failed tests	#failed test suites	#ODFT*	#ODFTS**	runtime (s)***
Test	67	6169	782	44713	4353	108	65	**52**	**19**	273788.1
Describe block	32	5818	692	18534	558	11	6	**3**	**1**	116013.6
Test suite	49	5960	850	59600	8500	0	0	0	0	44453.3
Total	148	17947	2324	122847	13412	119	71	**55**	**20**	434255.0

* order-dependent flaky tests
** order-dependent flaky test suites
*** runtime calculated for the total of 10 reorders each executed 10 times (100 runs in total)

5.6.2 Causes of Order-Dependent Tests (RQ2.2)

We present the results of our root-cause analysis on the detected order-dependent tests at the test level and at the `describe` block level.

5.6.2.1 Order dependency between tests

At the test level, most projects remain stable under reordering, with the majority of tests passing consistently across all 10 reorders and 10 reruns. However, 11 projects exhibit at least one failing test when the execution order changes, indicating the presence of order-dependent behaviour. We manually analysed these projects to determine the underlying causes of the observed failures.

Table 5.3 shows the results of our analysis on detected order-dependent tests at the test level. We identified 52 order-dependent tests from the nine projects that have them (corresponding to unshaded rows in Table 5.3). Based on our analysis, we grouped the causes of these order-dependent tests into two categories. The first category, *sharing files*, arises when tests depend on an external/shared artefact outside the individual tests, such as files, snapshots, cache folders, generated outputs, fixtures, or module/file-backed state. The second category, *sharing mocking state*, occurs when tests reuse or modify shared mocks, stubs, or global mocking configurations, causing later tests to observe an unexpected state depending on which test executes first.

We use the category *file sharing* to describe order dependency caused by shared state that exists outside individual tests and persists across their execution. In many cases, this shared state is represented by files, directories, snapshots, cache folders, fixtures, or generated artefacts. However, the main characteristic of this category is not only explicit file I/O, but the fact that one test changes an external shared artefact and another test observes or depends on that changed state. This distinguishes file sharing from *shared mocking state*, where the shared state is internal to the test suite, such as mock call counts, mock implementations, or mocked functions that are not reset between tests.

File sharing between tests: We find that developers use different methods to share external artefacts or file-backed state between tests, which can result in test flakiness if the order is expected to be fixed. Across the projects we analysed, we identified 13 tests whose failures were caused by file sharing in seven projects. We present the results for each of these projects below:

jest-junit: We identified one order-dependent test within the “buildJsonResults” test suite,

which is related to two tests: "should report no results as error" and "should honour templates when the test has errors". The two tests shared a mock file (as shown in Listing 5.6, lines 2 and 7). The first test sets `reportTestSuiteErrors: "true"` on line 4, which alters the cached object of the mock file used by `require` (lines 2 and 7) and consequently impacts the outcome of the subsequent test. On line 11, the test checks whether the total number of errors is equal to one. In case the second test runs first, the `totals.errors` becomes not equal to one, therefore the test fails. Consequently, we noted that whenever "should honour templates when the test has errors" runs before "should report no results as error", the latter test fails. Hence, "should honour templates when the test has errors" is a polluter and "should report no results as error" is a victim.

```
1 it('should honor templates when test has errors', () => {
2     const failingTestsReport = require('../__mocks__/failing-compilation.json');
3     ...
4     reportTestSuiteErrors: "true", ...
5 it('should report no results as error', () => {
6     const failingTestsReport = require('../__mocks__/failing-compilation.json');
7     ...
8     reportTestSuiteErrors: "true" ...
9     expect(totals.errors).toEqual(1);
```

Listing 5.6: Tests that shared a file from jest-junit

jest-image-snapshot: Another example of file sharing that leads to order-dependent flakiness is in "jest-image-snapshot".

Some tests create/update image snapshot files that other tests access. The location of these files (obtained using `getSnapshotFiles()`) is shared by the tests (shown in Listing 5.7). Specifically, the test "should write the image snapshot on first run" writes files to the shared location. The tests "should not delete the snapshot when environment flag is not enabled" and "should delete the snapshot when environment flag is enabled" expect those files to exist. That is, the latter two tests depend on the state set up by the first.

```
function getSnapshotFiles() {
 return fs.readdirSync(path.join(tmpDir, '__image_snapshots__'));}
it('should write the image snapshot on first run', () => {
 const { status, stdout, stderr } = runJest(['-u']);
 expect(stderr).toContain('snapshots written');
 expect(status).toEqual(0);
 expect(stdout).toEqual('');
 expect(getSnapshotFiles()).toHaveLength(3);});
it('should not delete the snapshot when environment flag is not enabled', () => {
 const { status, stdout } = runJest(['-u', 'image.test.js']);
 expect(status).toEqual(0);
 expect(stdout).toEqual('');
 expect(getSnapshotFiles()).toHaveLength(3);});
it('should delete the snapshot when environment flag is enabled', () => {
 const { status, stdout, stderr } = runJest(['-u', 'image.test.js'], {
  JEST_IMAGE_SNAPSHOT_TRACK_OBSOLETE: '1',});
 expect(stderr).toContain('outdated snapshot');
 expect(status).toEqual(0);
 expect(stdout).toEqual('');
 expect(getSnapshotFiles()).toHaveLength(1);});
```

Listing 5.7: Tests with shared files between them from jest-image-snapshot

Table 5.3: Results of Reordering Tests in 67 Projects

	Default Order		Reordered tests							
Project	#tests	#tests suites	#test	#test suites	%failed test	%failed tests suites	#ODFT	#ODFTS	OD cause	runtime(s)
jest-image-snapshot	100	5	980	46	5.0%	35%	**2**	**1**	file sharing	12404.5
jest-junit	53	6	496	34	7.0%	18%	**1**	**1**	file sharing	1201.6
jest-when[#]	86	1	860	10	1.0%	100%	0	0	-	820.1
jest-webextension-mock	173	14	391	96	38.0%	68%	**35**	**7**	shared mocking state	7491.8
commander.js[#]	1108	102	10189	867	0.1%	1%	0	0	-	54987.3
jest-mock-now	5	1	46	6	8.7%	67%	**1**	**1**	file sharing	411.8
KaTeX	1216	8	8020	62	0.2%	16%	**2**	**1**	file sharing	8174.6
native-testing-library	159	36	949	196	2.0%	4%	**4**	**1**	shared mocking state	14322.9
react-testing-library	237	14	392	75	0.3%	1%	**1**	**1**	file sharing	2329.3
serverless-jest-plugin	30	6	288	52	10.0%	54%	**5**	**5**	file sharing	6507.7
vue-testing-with-jest-conf17	11	3	78	14	6.0%	36%	**1**	**1**	file sharing	544.6
Total	3178	196	22689	1458			52	19		109196.2

[#] shaded rows are projects with failing tests but not order-dependent flaky tests

Shared mocking state between tests: Mocking state is a type of state sharing that specifically looks at the state of a mock. Across the projects we analysed, we identified 39 tests whose failures were caused by file sharing in two projects. We present the results for each of these projects below:

jest-webextension-mock: The project has seven test suites with 35 order-dependent tests. All 35 tests fail after reordering those tests due to the issue of checking the number of times functions are called. An example from the project is shown in Listing 5.8. The "command" test suite includes two tests: "getAll" and "getAll promise". In "getAll", when the function `browser.commands.getAll()` is called (line 3), the `expect` statement anticipates the function to be called once (line 4). The "getAll promise" test calls this function (line 7). If "getAll promise" runs before "getAll", the function gets called twice, causing the `expect` statement in line 4 to return false and a test to fail. One possible fix is to use `jest.clearAllMocks()` in a `beforeEach` hook to reset the mocking state. This will ensure that the mocks are cleared and the mock state is not shared between tests.

```
1  test('getAll', (done) => {
2   ...
3    browser.commands.getAll(callback);
4    expect(browser.commands.getAll).toHaveBeenCalledTimes(1);
5    ...
6  test('getAll promise', () => {
7    return expect(browser.commands.getAll()).resolves.toBeUndefined();
```

Listing 5.8: Tests with shared mocking state from jest-webextension-mock

native-testing-library: Listing 5.9 shows another example of order-dependent tests caused by a shared mocking state. The test suite has five tests, of which four are expected to use the mock function on line 2. The remaining test, 'async act should not show an error when ReactTestUtils.act returns something', uses the same mock function with a new and different implementation specific to the test (line 12). In the default order, test 'async act should not show an error when ReactTestUtils.act returns something' is the last test in the test suite, meaning it will run last, and therefore it does not share its' inner mocking function with other tests. As a result of reordering tests, the `jest.mock('react-test-renderer')` defined in this test will impact the other four tests as the mocking state changes. This causes the other tests that come after the target test to fail.

Other failed tests: We observe two projects with failing tests after the reordering process: "jest-when" [122] and "commander.js" [123], but we do not find the failing tests to be order-dependent tests. In the "jest-when" project, we observed test logic that depends on the test

```
1     let asyncAct;
2  jest.mock('react-test-renderer', () => ({
3    act: cb => { ...}}))
4  beforeEach(() => {
5    jest.resetModules();
6    asyncAct = require('../act-compat').asyncAct;
7    jest.spyOn(console, 'error').mockImplementation(() => {});})
8  afterEach(() => {
9    console.error.mockRestore();})
10 test('async act should not show an error when ReactTestUtils.act returns something', ↩
       async () => {
11   jest.resetModules();
12   jest.mock('react-test-renderer', () => ({
13     act: () => { ...}}))
```

Listing 5.9: Tests with shared mocking state from native-testing-library

name and file structure. For example, as shown in Listing 5.10, test "fails verification check if all mocks were not called with line numbers" (line 5) depends on the name of the test suite. The tests fail when we transform them into new suites (i.e., the newly reordered test files). Therefore, we attribute the failure to the structure of the test code rather than to test-order dependencies. In addition, "commander.js" experiences failed tests when rerun in some orders. After analysis, we discovered these failures are false positives, more related to resource issues than test order dependency.

```
1  it('fails verification check if all mocks were not called with line numbers', () => {↩
       ...
2     try {
3     verifyAllWhenMocksCalled();
4   } catch (e) {
5     const errorLines = e.message.split('\n');
6     const currentFilePathPattern = /src(?:\\|\/)when\.test\.js:\d{3}(.|\s)*/;
```

Listing 5.10: Tests From jest-when

There were also nine projects with failing test suites after reordering the tests. After manually analysing the results, we found that failures occur because Babel did not parse certain parts of the original test suites, and thus `JS-TOD` did not add them in the new test suites. For example, Listing 5.11 from "jest-it-up" [124] shows part of a test suite that has a `require` statement in line 2. `JS-TOD` does not identify this statement when traversing the AST of the original test suite and does not include it in new test suites after reordering. This parsing error causes the test "returns the new contents, and changes if new thresholds" to fail. As a result, we manually fix the code of the new test suites by adding the missing

code segments and then comparing those with the original suites to ensure that they are identical. After this manual modification, we rerun all test suites, resulting in all tests passing.

In addition, there are two projects ("jest-fail-on-console" and "KaTeX") with specific Jest configurations for running tests with defined name patterns. As we add reordered tests to the new test suite, we change Jest's configuration to run them.

```
1 const fs = require('fs')
2 require('ansi-colors').enabled = false
3     ...
4 it('returns the new contents and changes if new thresholds', () => {
5     ...
```

Listing 5.11: Example of an unidentifiable code by AST from jest-it-up

5.6.2.2 Order dependency between `describe` blocks

After reordering `describe` blocks, nearly all projects continue to pass their test suites, indicating limited order-dependent behaviour at this level. Only three projects exhibit failing tests under `describe`-level reordering: "jest-when", "commander.js", and "fetch-mock-jest". As noted in Section 5.6.2.1, "jest-when" includes a test that relies on the names and structure of the test suites. Failing tests in "commander.js" are caused by resource-related issues. Hence, we do not attribute the failed tests to order-dependent flakiness. Table 5.4 shows the projects with failing tests after reordering `describe` blocks.

The project "fetch-mock-jest"[16] has two order-dependent `describe` blocks. The `describe` blocks contain three tests that expect a function to be called a set number of times (lines 5, 9 and 13 in Listing 5.12). As a result of this shared mocking state, changing the order of `describe` blocks causes tests to fail.

5.7 Proposed fixes to Order-Dependent Tests

As shown in the results of our experiment on order-dependent test (Section 5.6), the two root causes of order-dependent tests are 1) shared mocking state and 2) shared files. In this section, we examine the possible fix technique for the detected order-dependent tests.

[16] `https://github.com/wheresrhys/fetch-mock-jest/releases/tag/v1.5.1`

```
describe('exposing mock internals', () =>{
    ...
 it('exposes `calls` property', () =>{
  expect(fetch.mock.calls).toBeDefined();
  expect(fetch.mock.calls.length).toBe(2);
  ...
 it('exposes `results` property', async () =>{
  expect(fetch.mock.results).toBeDefined();
  expect(fetch.mock.results.length).toEqual(2);
  ...
describe('native jest mock function inspectors', () => {
 it('.toHaveBeenCalled()', () => {
    expect(() => expect(fetch).toHaveBeenCalled()).not.toThrow();
```

Listing 5.12: `describe` Blocks from fetch-mock-jest

Table 5.4: Result of Reordering `describe` Blocks

	Default Order		Reordered tests							
Project	#tests	#tests suites	#test	#test suites	%failed test	%failed tests suites	#ODFT	#ODFTS	OD cause	runtime(s)
jest-when	86	1	172	2	1.0%	100.0%	0	0	-	417.4
commander.js	1108	102	2590	122	0.2%	2.9%	0	0	-	14616.0
fetch-mock-jest#	216	9	2076	76	0.4%	5.3%	**3**	**1**	shared mocking state	4320.0
Total	1410	112	4838	200			3	1		19353.4

shaded rows are projects with order-dependent flaky tests

5.7.1 Fixing Shared Mocking State Flakiness

Regarding the experimental results, sharing mocking states mostly occurs when the test verify how many times a function has been invoked (35 tests out of 39 cases). In practice, developers frequently use Jest's mock-inspection functions, such as `toHaveBeenCalled`, `toHaveBeenCalledTimes`, and `toBeCalled`. These options are used to validate interactions between components, ensuring that a function is triggered exactly once or a specific number of times during execution [118]. These assertions are valuable for checking behaviour in event-driven or asynchronous code, where correct invocation patterns often signal correct system behaviour. In addition, Jest provides other options for inspecting mock states, such as `isMockFunction` (e.g., determines whether the given function is a mocked function) and `toBeUndefined` (e.g., checks whether a variable is undefined). This information is stored globally within the test suite.

To clear the shared mock state, we implement a static approach, shown in Algorithm 3. We developed an approach that accepts a JavaScript project and a rerun number as input. Then it extracts the test suite paths and searches the keywords ("toHaveBeenCalled", "toHaveBeenCalledTimes", "jest.isMockFunction", "toBeCalled", "toBeUndefined") within the test suites. If a test suite includes at least one of the keywords, the tool instruments the code and adds a `beforeEach` block (Listing 5.13) to the test suite. This forces Jest to clear the mocking state before each test and ensures that every test starts with fresh mock objects. The command `jest.clearAllMocks()` clears the `mock.calls`, `mock.instances`, `mock.contexts`, and `mock.results` properties of all mocks [118].

Algorithm 3 Clearing Shared Mocking States Approach

```
▷ Input: project path and rerun number
▷ Output: execution results in JSON format saved to the project path
INSTALL_PROJECT(project_path)
testSuites ← ListTestSuites(project_path)
for each testSuite in testSuites do
    mockingState ← SearchKeywords(testSuite)
    if mockingState then
        InsertClearMockState(testSuite)
    end if
end for
for each updatedTestSuite in updatedTestSuites do
    RUN(updatedTestSuite, rerun_number)
end for
```

```
1 beforeEach(() => {
2     jest.clearAllMocks();
3 });
```

Listing 5.13: Clearing the shared mocking states before each test

While our static approach successfully resolved most instances of flakiness caused by shared mocking states (34 out of 35), it was not effective in every case. Running the tool on "jest-webextension-mock" fixed all the order-dependent tests except for one test. The test "chrome.omnibox.onInputChanged.addListener" in Listing 5.14 checks the function "chrome.omnibox.onInputStarted.addListener" is called in line 10. While the function is called in line 3 of another test, cleaning the mocks before tests causes the test "chrome.omnibox.onInputChanged.addListener" to fail.

```
1  test('chrome.omnibox.onInputStarted.addListener', () => {
2    expect(jest.isMockFunction(chrome.omnibox.onInputStarted.addListener)).toBe(true);
3    chrome.omnibox.onInputStarted.addListener(() => {});
4    expect(chrome.omnibox.onInputStarted.addListener).toBeCalled();
5  });
6
7  test('chrome.omnibox.onInputChanged.addListener', () => {
8    expect(jest.isMockFunction(chrome.omnibox.onInputChanged.addListener)).toBe(true);
9    chrome.omnibox.onInputChanged.addListener(() => {});
10   expect(chrome.omnibox.onInputStarted.addListener).toBeCalled();
11 });
```

Listing 5.14: Order-dependent test from jest-webextension-mock

5.7.2 Fixing file sharing state

As shown in Section 5.6, file sharing is another cause of order-dependent tests in JavaScript projects. An in-depth investigation of the projects in this category shows that dependencies among tests are either intentional or do not follow a consistent pattern. For example, Listing 5.15 presents two tests that were deliberately written to be order-dependent (even their names indicate the dependency).

```
1  test('first', () => {
2    render(<div>hi</div>)})
3  test('second', () => {
4   expect(document.body.innerHTML).toEqual('<div><div>hi</div></div>')})
```

Listing 5.15: Order-dependent tests from "react-testing-library"

Listing 5.16 shows another example of order-dependent test in this category, where two tests share `Date.now()` and `NOW` (a file includes a variable name NOW). The test "Date.now, as a mock function, exposes mockRestore() in line 3, which attempts to restore mock for `Date.now()`. While calls to `jest.mock()` are hoisted to the top of the file [118], the change remains for the rest of the tests that run after it. As a result, when test "Date.now() from 'setup-jest' to match snapshot w/o options" runs after the other test will fail because line 12 checks the values of Data.now() and NOW to be equal.

Listing 5.17 shows two more order-dependent tests examples from "serverless-jest-plugin". In line 12, test "tests setEnv with testFunction1" sets the value of the environment process. As mentioned earlier, the value will be valid for all tests in the test suite. As a result, test "tests setEnv with testFunction1 (env vars)" will fail if it runs after the first test. It happens because, in line 26, the test expects a value for `process.env.TEST_VALUE_PROVIDER` to be

```
1  test('Date.now, as a mock function, exposes mockRestore()', () => {
2    const mocked = Date.now();
3    Date.now.mockRestore();
4    const real = Date.now();
5    expect(real).toBeGreaterThan(mocked);
6    console.log(NOW);
7  });
8
9  test('Date.now() from `setup-jest` to match snapshot w/o options', () => {
10   console.log('NOW:', NOW);
11   console.log('Current Date.now:', Date.now());
12   expect(Date.now()).toEqual(NOW);
13   expect(Date.now()).toMatchSnapshot('setup-jest snapshot');
14   console.log('NOW:', NOW);
15 });
```

Listing 5.16: Order-dependent test from jest-mock-now

undefined.

Jest option "jest.restoreAllMocks()" restores all mocks and replaced properties to their original value [118]. However, using this option did not fix the issue in our cases. Our observation aligns with reports from developers indicating that `jest.restoreAllMocks()` is unreliable[17] , and in some situations it fails to reset mock states as expected. These issues often arise when mocks involve getters or non-configurable properties, when mocks are applied to modules that Jest hoists or transforms, or when mock state accumulates across tests due to the use of `beforeAll` rather than `beforeEach`. In such cases, the restore operation may not fully revert the function to its original behaviour, leaving a residual state that leads to order-dependent outcomes.

5.8 Threats to Validity

There are several validity threats to the study's results, which we discuss below.

Number of runs and orders: Deciding on how many reruns are required to reveal if a test is flaky has been pointed out as a significant issue in test flakiness research [78]. We decided to use 10 reordering and 10 reruns for each new order (i.e., 100 reruns for each test suite). Based on our pilot runs and prior research on flaky tests, we observed that 10 reruns yielded results comparable to 20 or 30 reruns. This setting was selected as a practical trade-off between detection capability and execution cost. While increasing the number of reorders may reveal additional order-dependent tests, it also increases the overall

[17] `https://github.com/jestjs/jest/issues/13925`, `https://github.com/jestjs/jest/pull/13867`

```
    it('tests setEnv with testFunction1', () => {
  const serverless = {
    service: {
      provider: {
        environment: {
          TEST_VALUE_PROVIDER: 'test value provider'
        }
      },
      functions: {
        testFunction1: {
          environment: {
            TEST_VALUE_FUNCTION: 'test value function 1'}}}}
  };
  utils.setEnv(serverless, 'testFunction1');
  expect(process.env.TEST_VALUE_PROVIDER).toBe('test value provider');
  expect(process.env.TEST_VALUE_FUNCTION).toBe('test value function 1');
})
it('tests setEnv with testFunction1 (env vars)', () => {
  const serverless = {
    service: {
      provider: {},
      functions: {
        testFunction1: {}}}
  };
  utils.setEnv(serverless, 'testFunction1');
  expect(process.env.TEST_VALUE_PROVIDER).toBe(undefined);
  expect(process.env.TEST_VALUE_FUNCTION).toBe(undefined);
})
```

Listing 5.17: Order-dependent test from serverless-jest-plugin

execution time, particularly for large projects. JS-TOD is user-configurable, allowing users to increase the number of reorders and reruns based on the size of their test suites and available computational resources. In our evaluation, using 10 reorders and 10 reruns was sufficient to reveal order-dependent tests in JavaScript projects, and the observed results are consistent with our earlier empirical finding that order dependency exists in JavaScript but appears less frequently than other causes of flakiness.

Detection coverage: JS-TOD uses randomised test reordering to detect order-dependent flaky tests, which provides a practical balance between detection effectiveness, scalability, and execution cost. However, random reordering is not exhaustive and therefore cannot guarantee that all order-dependent interactions will be explored. This limitation becomes more significant as the number of tests increases, because the number of possible test orders grows factorially, making full exploration infeasible for real-world test suites. As a result, JS-TOD may miss order dependencies whose failure-inducing orders are rare within

the overall permutation space. Prior work has shown that, for some order-dependent tests, the probability of producing a failing order through random permutations can be very low; for example, Li et al. [65] reported cases with only about a 1.2% chance of generating a failure-inducing order. Therefore, JS-TOD should be interpreted as a practical and scalable detection approach rather than an exhaustive detector. Future work could improve coverage by incorporating more systematic ordering strategies, such as Tuscan-square-based combinatorial designs, which can provide stronger guarantees by ensuring that relevant test-pair interactions are exercised more deliberately, although these strategies may incur additional computational cost.

Reproducibility of results: Our results from each program may not be easily reproducible as we used a randomization approach to reorder tests (i.e., with each reorder, you may get different results). However, it is possible to reproduce the same results by running the same order we found to be flaky (see our replication package for more details [121]).

Generalizability of results: The results are based on analyzing JavaScript programs that use Jest as their testing framework. We cannot guarantee that the results will generalize to JavaScript programs that use other testing frameworks, such as Vitest and Mocha. Jest is the most popular framework and thus is considered representative. While there are similarities among testing frameworks, one must examine the approach using other frameworks, which requires updating the tool to match the framework's options and settings.

5.9 Summary

In this chapter, we systematically investigated test flakiness caused by test-order dependencies in JavaScript, with a particular focus on the Jest framework. To support this investigation, we developed a randomized test reordering approach (implemented in a tool called `JS-TOD`) that targets dependencies at three levels: individual tests, `describe` blocks, and test suites. For each level, `JS-TOD` generated ten randomized execution orders, and each reordered suite was rerun ten times to expose hidden order dependencies that may not manifest under the default execution order.

Our evaluation of 81 projects shows that order-dependent flakiness occurs in JavaScript but is considerably less prevalent than reported in prior studies of other programming languages. Out of 873 investigated test suites, we identified 55 order-dependent tests across 20 test suites in 10 projects. These failures emerged only under specific execution orders, indicating that such dependencies can remain latent in continuous integration environments

that rely on a fixed, deterministic order. In contrast to earlier empirical studies that report test order dependency as a dominant source of flakiness, our findings suggest that, in JavaScript projects, this form of flakiness is relatively rare.

All detected cases were caused by a shared state between tests, either through shared files or shared mocking state. While file-based dependencies have been reported in prior work, shared mocking state has not previously been identified as a root cause of order-dependent flakiness in other languages. This finding reflects JavaScript-specific testing practices, particularly the extensive use of global mocks and dynamic module replacement in Jest, which can introduce subtle inter-test dependencies if not properly reset.

We further analysed the root causes of these failures and evaluated the effectiveness of potential fixes. For order-dependent flakiness caused by shared mocking state, inserting explicit state-reset logic, typically via a `beforeEach` block with `jest.clearAllMocks()`, resolved nearly all cases. The remaining instances corresponded to tests that were intentionally written with dependencies, reflecting design choices rather than unintended flakiness. In contrast, order-dependent tests caused by file sharing were more challenging to address. These tests often relied on persistent file-system state in project-specific ways, were sometimes intentionally dependent on execution order, and did not follow a consistent repair pattern, thereby limiting the effectiveness of automated fixes.

Overall, our empirical investigation shows that although order-dependent flakiness is not a dominant issue in JavaScript testing, it does arise in practice and is driven by a small set of recurring patterns. Compared with findings from other ecosystems, JavaScript projects exhibit fewer order-dependent failures but introduce distinct causes associated with mocking and dynamic state management. These results refine our understanding of flaky test behaviour across languages and provide guidance for both practitioners and tool builders seeking to detect and mitigate order-dependent flakiness in JavaScript.

MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
DOCTORATE WITH PUBLICATIONS/MANUSCRIPTS

We, the student and the student's main supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the student's contribution as indicated below in the Statement of Originality.

Student name:	Negar Hashemi
Name and title of main supervisor:	Associate Professor Amjed Tahir
In which chapter is the manuscript/published work?	Chapter 6 – Understanding Environment-Dependent Tests

Describe the contribution that the student and members of the supervisory team have made to the manuscript/published work:[1]

The student constructed the study, designed the experimental infrastructure, conducted the evaluation across multiple environments, analysed the results, and wrote the draft of the paper.
The supervisors contributed to the design of the study, evaluation methodology, interpretation of results, and reviewing and editing of the paper.
Student (Negar Hashemi): 70%
Supervisors and co-authors: 30%

Please select one of the following three options:

- (●) **The manuscript/published work is published or in press**

  Please provide the full reference of the research output:

  Negar Hashemi, Amjed Tahir, Shawn Rasheed, August Shi, and Rachel Blagojevic. "A Systematic Evaluation of Environmental Flakiness in JavaScript Tests," 2026 IEEE Conference on Software Testing, Verification and Validation (ICST), Daejeon, South Korea.

- ( ) **The manuscript is currently under review for publication**

  Please provide the name of the journal:

- ( ) **It is intended that the manuscript will be published, but it has not yet been submitted to a journal**

Student's signature:	Negar Hashemi Digitally signed by Negar Hashemi Date: 2026.01.26 23:05:28 +13'00'	Main supervisor's signature:	Amjed Tahir Amjed Tahir 2026.01.29 07:49:13 +03'00'

This form should be placed at the beginning of each relevant thesis chapter.

[1] Refer to the Massey University Publishing and Authorship guidelines (OneMassey for staff, Stream for students) and/ or Contributor Roles Taxonomy (CRediT) guidelines for guidance.

Chapter 6

Understanding Environment-Dependent Tests

6.1 Introduction

The studies presented in the previous chapters (Chapter 4 and Chapter 5) examined test flakiness in JavaScript from two perspectives: 1) flaky tests' root causes and strategies followed in practice to address them, and 2) a systematic analysis of order-dependent tests as one of the causes of flaky tests. Together, these investigations revealed that test flakiness in JavaScript is a multifaceted problem, arising not only from internal code and test interactions but also from the conditions in which tests are executed.

Our empirical study (presented in Chapter 4) showed that, unlike in other ecosystems, environment-related issues, such as differences in OSs and browsers, emerged as one of the top three causes of flaky tests in JavaScript. These findings highlight that the execution environment itself can introduce nondeterminism, making it difficult for developers to trust their test outcomes. Such environmental flakiness is especially problematic in large-scale projects that must be tested across multiple platforms.

The third objective of this thesis is therefore to investigate, in depth, test flakiness caused by environmental variables and to explore practical approaches to mitigate them. In this chapter, we present a systematic evaluation of the impact of the running environment on test outcomes, focusing on three environmental factors: the OS, Node.js version, and the browser. Furthermore, we designed and evaluated an automated approach that enables developers to mitigate environment-dependent tests.

In the chapter, we address the following main research question:

RQ3 How can we control tests from flaky tests caused by environmental dependency?

To address this question comprehensively, we divided it into three sub-questions:

RQ3.1. How prevalent are environmental flaky tests in JavaScript?

This question aims to investigate the prevalence of environmental flakiness in JavaScript. We employed a systematic rerun approach to uncover environmental flaky tests in JavaScript projects. By varying environmental factors and rerunning tests under different conditions, we assess whether test outcomes change with the environmental setup in which the tests are executed.

RQ3.2. What are the root causes of environmental flakiness?

This question aims to understand the main causes of environmental flaky tests in JavaScript. We manually inspected each detected environmental flaky test and classified the cause of flakiness into different categories.

RQ3.3. How can we mitigate test flakiness caused by environmental factors?

A common industrial practice for dealing with flaky tests is to quarantine them, that is, to allow the CI pipeline to bypass the test while still reporting it as flaky [125]. This practice is also supported by several test monitoring tools, which identify flaky tests based on their behaviour in previous executions and automatically quarantine them [99, 126, 127]. Motivated by this widespread practice, this question aims to develop a practical approach to mitigate flakiness caused by environmental variation.

The environment in which programs are executed can significantly influence their behaviour, which, in turn, can affect test execution and lead to changes in test outcomes (i.e., flakiness) depending on the environment configuration. These tests are therefore *flaky* because their outcomes depend not on the code they test but rather on external environmental factors. Building on the general definition of flaky tests in Chapter 1, we refer to such tests as *environmental flaky tests*: tests that fail in one environment but pass in others (with other variables in control, including code). These failures may be consistently tied to a specific environment or occur intermittently. We consider compatibility issues as a subset of environmental flakiness, as they manifest as failures that occur only in certain environments (e.g., due to differences in API implementations across JVMs, while the calling code remains unchanged [57]).

Environmental flakiness refers to test failures caused not by issues in the test logic or application code itself, but by variations in the environment in which the tests are executed. These environmental factors can include variations in the operating systems, hardware configurations, available system resources, installed libraries, or browsers [78]. When a test passes in one environment but fails in another without any code changes, it is

often considered to be environmentally flaky.

This type of flakiness is particularly challenging to detect and reproduce, as it may depend on subtle, non-deterministic interactions with the system. For example, a test may rely on OS-specific commands, filesystem semantics, or path configurations, causing it to fail intermittently on macOS while passing consistently on Linux, or to behave differently when executed inside a container compared to a native environment. Such environment-specific assumptions are often implicit in the test or the code under test, making the resulting failures hard to anticipate and reproduce. These inconsistencies complicate debugging and erode developer confidence in the test suite [8]. Environmental flakiness can undermine the reliability of continuous integration (CI) pipelines, delay deployments, and introduce unnecessary engineering effort. As modern software systems increasingly rely on distributed, containerised, and cross-platform deployments, managing environmental variability becomes critical to ensuring stable and trustworthy test results. Understanding and addressing environmental flakiness is thus essential for maintaining high-quality software and efficient development workflows.

In the context of this study, we considered three environmental factors that may contribute to test flakiness in JavaScript: (1) the OS, (2) the version of the runtime environment (Node.js), and (3) the browsers.

6.1.1 Environmental Factors

Operating Systems: Our initial study demonstrated that OS are one of the main causes of flakiness in JavaScript projects (Chapter 4). In practice, the most commonly used platforms are Ubuntu, macOS, and Windows [128]. Differences between OS can lead to inconsistent test behaviour due to variations in properties such as file systems (e.g., path separators, case sensitivity), line endings, default locale and encoding, performance and timing, permission models, as well as OS-specific functions [72, 78]. For example, Linux distributions such as Ubuntu typically use case-sensitive file systems and Unix-style path separators (`/`). In contrast, Windows uses a case-insensitive file system by default and the backslash (\) as the path separator. macOS combines Unix-like behaviour with its own conventions, such as HFS+ or APFS file systems that may treat case sensitivity differently depending on configuration. Line endings also vary; Unix systems use `\n`, while Windows uses `\r\n`. In addition, native dependencies may compile or function differently across operating systems, and OS-specific APIs for networking, file watching, or process management can further impact execution.

Node.js: Node.js [1] is a cross-platform JavaScript runtime environment popular in the JavaScript community [2]. Node.js allows developers to execute JavaScript code outside a web browser. It is widely adopted in open-source and enterprise projects, with a rich ecosystem of packages supported through the Node Package Manager (npm). Node.js adheres to Semantic Versioning (SemVer), under which major releases may introduce breaking changes. Given the numerous available Node.js versions, it is important to consider how they may affect test outcomes.

The JavaScript engine and built-in APIs may change and update between Node.js versions. These changes can affect how code executes, particularly with respect to timing and asynchronous behaviour, thereby exposing hidden race conditions or timing-related issues in tests which in turn leads to test flakiness.

Browsers: Browsers introduce another major source of environmental variability for JavaScript tests. A large number of JavaScript projects involve processing and rendering browser-related events. Different browsers (e.g., Chrome, Firefox and Safari) make slightly different choices about rendering, timing, caching, and permissions. These minor differences can cause a test to pass in one browser but intermittently fail in another. In addition, the interaction between OS and browser can further contribute to test flakiness, as browsers may behave differently depending on the underlying OS. For example, a browser feature might work reliably on macOS but behave inconsistently on Windows due to differences in OS-level APIs or hardware acceleration support.

Furthermore, the interaction between operating systems and browsers can contribute to test flakiness, as browsers may behave differently depending on the underlying OS. Variations in how browsers implement web standards, manage rendering, and handle system resources can be influenced by OS-specific factors such as graphics drivers, file systems, and security settings.

6.2 Methodology

We conducted our experiment in two phases: (1) we ran tests multiple times from each project under different environment configurations to see whether tests produce different outcomes, and then (2) we manually analysed the flaky failures and categorized the causes of flakiness. Figure 6.1 shows an overview of our experiment, including the data collection

[1] `https://nodejs.org/en`

[2] `https://survey.stackoverflow.co/2024/technology#1-web-frameworks-and-technologies`

and analysis process.

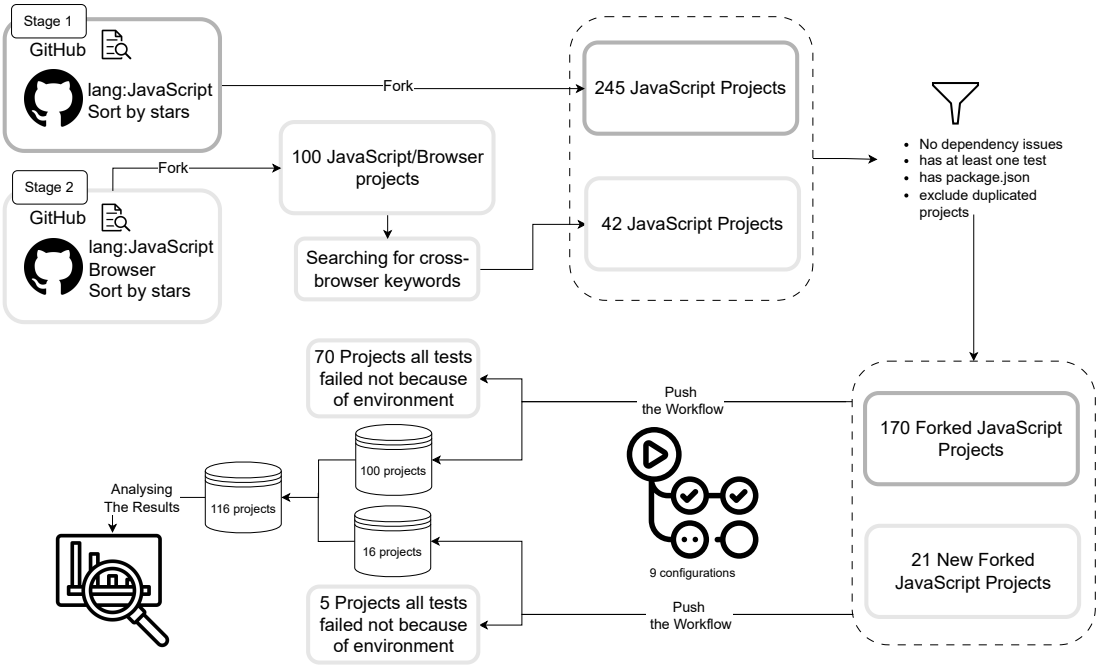

Figure 6.1: An overview of the data collection and analysis process

6.2.1 Dataset

We constructed a dataset of JavaScript projects by using GitHub's REST API [129] to collect the most popular projects ranked by GitHub stars. GitHub stars are a metric that indicates the level of interest a project has received, with prior work finding correlations between the number of stars and the number of forks and contributors [130]. We selected the 700 most-starred JavaScript projects to form an initial dataset large enough for inspection. We then filtered out projects that lacked tests or did not use the Node.js runtime environment, yielding 170 JavaScript projects. Note that these projects may use different JavaScript testing frameworks (e.g., Jest, Mocha, and Vitest).

After forking the 170 projects, we committed and pushed a GitHub Actions workflow to our forked repository to enable us to run these tests in different environments using GitHub Actions CI. We noted that some runs can fail due to dependency errors or blocked actions in the projects. We manually customized the GitHub Actions CI workflow to fix dependency errors or failed tests; further details and concrete examples are provided in Section 6.2.3. We excluded any projects for which we cannot resolve the issues, so those projects still

cannot be built correctly. After excluding all projects we could not build, we were left with 100 projects in our dataset.

Since one of the primary applications of JavaScript is to enable client-side execution across different web browsers, we aim to investigate how interactions between operating systems and browsers contribute to test flakiness. This combination can lead to different test outcomes because browsers rely on OS-level services and APIs for tasks such as file access, networking, timing, and graphics, and they may also differ in their JavaScript engines, event scheduling, and implementation details across platforms. As a result, the same test can exhibit different behaviour when executed in the same browser on different operating systems, or across different browsers on the same OS, motivating a closer examination of OS–browser interactions as a source of flakiness. However, we found that our dataset contained only 22 web-based JavaScript projects. Therefore, we conducted an additional search on GitHub to identify and locate projects specifically designed to run cross-browser tests. By filtering for JavaScript projects containing the keyword "browser", we selected the 100 most-starred repositories to expand our dataset. To determine whether a project includes cross-browser testing, we examined each project's package file for browser-specific keywords. We selected a set of terms commonly associated with tools and frameworks used for browser-based or end-to-end testing, including: "playwright", "karma-chrome-launcher", "karma-firefox-launcher", "karma-safari-launcher", "selenium-webdriver", "webdriverio", "testcafe", "cypress", "test:e2e", "gulp" and"grunt". This search identified 42 projects, 11 of which were already available in our initial dataset. We apply the same processing steps used for the initial dataset to the remaining 21 projects. After excluding overlapping projects between the two datasets, we construct an additional dataset comprising 16 new projects. Overall, our combined dataset contains 116 projects. Table 6.1 summarizes the statistics of the projects in our dataset.

Table 6.1: Project Statistics

Stats	#Stars	#Forks	#Contributors	#Test Files
Mean	35411.27	5482.19	314.78	63.72
Total	4107707	635934	36515	7392

6.2.2 Procedure

Using our dataset of JavaScript projects, we developed a procedure to run tests on those projects and assess whether variation in environmental factors affects test outcomes. For

each project, we created a GitHub Actions workflow configuration to run tests in different environments. We define various combinations of environmental factors, including operating system and Node.js version, using a matrix strategy (see Listing 6.1). This configuration enables the CI pipeline to run each project across nine distinct environment setups, with each environment rerun 10 times, resulting in a total of 90 runs per project.

```
strategy:
      fail-fast: false
      matrix:
        os: [ubuntu-latest, macos-latest, windows-latest] # Operating systems to test
        node-version: [18, 20, 22] # Node.js versions to test
        attempt: [1, 2, 3, 4, 5, 6, 7, 8, 9,10] # Repeat tests 10 times
```

Listing 6.1: Strategy used to run test across different environmental setups in GitHub Actions

Table 6.2 reports detailed information about the selected operating systems as provided by GitHub Actions runners. Specifically, we consider three widely used operating systems (Ubuntu, macOS, and Windows) and use their latest available runner images at the time of the experiment[3]. For each selected OS, the table lists the corresponding Node.js versions and the default browsers preinstalled on the runners.

We then systematically pair the selected operating systems with these Node.js versions using a matrix strategy in GitHub Actions to form a reasonable and representative set of execution configurations.

As shown in Table 6.2, the three primary OSs (Ubuntu 24.04, macOS 14, and Windows Server 2022) each include multiple pre-installed browsers, including Google Chrome, Microsoft Edge, Mozilla Firefox, and Safari (exclusive to macOS). Browser versions vary slightly across these OSs, affecting feature support and rendering behaviour. For instance, Safari is available only on macOS, whereas Microsoft Edge and Firefox are available on both Ubuntu and Windows. We therefore cannot explicitly change browser versions since they are associated with the selected OSs, and we consequently rely on the default browser versions provided by the GitHub Actions runners for each operating system.

[3] https://github.com/actions/runner-images#available-images

Table 6.2: Default OS, Node.js, and Browser in GitHub Actions runners

Label	OS	Node.js	Default Browser
ubuntu-latest	Ubuntu 24.04	18.20.8 20.19.1 22.15.0	Google Chrome 136.0.7103.92 Microsoft Edge 136.0.3240.64 Mozilla Firefox 138.0.1
macos-latest	macOS 14	18.20.8 20.19.1 22.15.0	Google Chrome 136.0.7103.49 Safari 18.4 (19621.1.15.111.1)
windows-latest	Windows Server 2022	18.20.8 20.19.1 22.15.0	Google Chrome 136.0.7103.93 Microsoft Edge 136.0.3240.64 Mozilla Firefox 138.0.1

6.2.3 GitHub Actions

We utilised GitHub Actions to run our experiment, enabling us to systematically explore the impact of environmental variations on test outcomes. GitHub Action[4] is a powerful automation tool integrated directly into GitHub that enables developers to define and execute custom workflows for CI and continuous deployment (CD)[5]. With a GitHub Actions workflow, developers can automate tasks such as building, testing, and deploying code whenever changes are pushed to a repository. GitHub Actions supports a variety of OS, including Ubuntu, macOS, and Windows, and it provides access to pre-installed environments with popular development tools and languages.

GitHub Actions supports multiple operating systems, including Ubuntu, macOS, and Windows, and provides access to pre-installed environments with popular development tools and languages. For our experiments, we set up a GitHub Actions workflow to run tests from each project across multiple OS (Ubuntu, macOS, and Windows), Node.js versions, and browsers in consistent, isolated environments.

We manually customize the GitHub Actions workflow for each project to ensure compatibility and successful execution. These adjustments help ensure that the test environment accurately reflects the project's expected setup and minimizes configuration-related failures that are unrelated to the environmental factors we vary. The process includes setting the correct branch name when the default is not `main`, as some projects use alternative names such as `master` or custom branches for active development. Additionally, we adapted the

[4] `https://docs.github.com/en/actions`

[5] `https://github.blog/news-insights/product-news/github-actions-now-supports-ci-cd/`

installation steps for each project to meet its specific requirements. For instance, some projects require running custom scripts, using different package managers (e.g., `pnpm` vs. `npm`), or installing additional dependencies, such as system packages or global tools, before tests can run.

After running each project across all environmental configurations, we saved the results of each rerun under the name of the corresponding forked project. For some failing runs due to dependency errors or blocked GitHub Actions runs, we manually customize the workflow to fix dependency errors or failed tests. For example, Bootstrap[6] requires Java Development Kit (JDK) 17, but the default JDK on Windows Server 2022 in GitHub Actions is Java 8. Therefore, we modify the workflow to install JDK 17 when running tests on Windows (Listing 6.2).

```
- name: Set up Java 17 (Windows only)
    if: runner.os == 'Windows'
    uses: actions/setup-java@v3
    with:
      distribution: 'temurin'
      java-version: '17'
```

Listing 6.2: Adding step to GitHub workflow for bootstrap

After customizing the GitHub Actions workflow for each project, we commit the changes to our forked repository. Pushing these changes triggers the GitHub Actions pipeline, which automatically starts running the test jobs.

6.2.4 Manual Analysis

We manually reviewed the results obtained from all 116 projects. For each observed test failure in GitHub Actions, two reviewers independently examined the error message and traced the relevant code to determine the root cause. If a failure was attributable to the OS, Node.js version, browser, or an interaction between them, we categorized it as environmental flakiness and assigned it to the corresponding category. When the two reviewers disagreed, a third reviewer was involved to resolve the conflict. The two reviewers agreed on the classification for 54 of these projects (81.8%). For the remaining 12 projects (18.2%), a third reviewer independently reviewed the evidence and the final classification was determined through discussion until full consensus was reached. In total, we observed test failures across 66 projects, of which 65 are environmental flaky projects and five are flaky, due to other

[6] `https://github.com/twbs/bootstrap`

factors; four projects overlap between the two categories.

6.3 Results

We present the results in relation to the first two research questions below.

6.3.1 Prevalence of Environmental Flakiness in JavaScript (RQ3.1)

By analyzing 116 projects, we categorise those projects into four groups based on the flaky tests we detected in those projects across different environmental configurations (i.e., OS type and Node.js version):

- **Non-flaky projects**: projects that exhibit no test failures when running across different environmental configurations.
- **Environment-dependent projects**: projects with test failures caused by changes in the environment. These are further divided into:
  - **Environmental flaky projects**: failures occur due to environment configurations at the project level, e.g., the project's package dependencies, before tests are even run.
  - **Environmental flaky tests**: only specific tests fail under certain environment configurations, while the rest of the project remains stable.
- **Flaky projects**: projects that contain non-environment-related flaky tests beyond the factors that we control for, e.g., failures caused by network instability or randomness.

In total, from all 116 projects, we found 65 environmental dependent projects, including 39 environmental flaky projects and 28 projects with environmental flaky tests. We detect 1355 environment-dependent tests within those 28 projects. On the other hand, 50 projects are non-environment-dependent. Most environmental flaky failures manifested on the first rerun after a change in the environment (77% of projects). We required 10 reruns in the worst-case scenario.

Table 6.3 indicates the number of projects in different categories, as well as the number of causes in each. Note that a project may belong to more than one group if it exhibits multiple types of behaviour. For example, a project may have tests that fail only in specific environments (categorized as *environmental flaky tests*) and also include tests that fail

Table 6.3: Projects with or without environmental dependencies

	#projects	#Node dependent	#OS dependent	#Browser dependent	#Combination of Node and OS dependent
Non-flaky projects	50	-	-	-	-
Environmental flaky projects	39	3	21	8	8
Environmental flaky tests	28	2	7	9	8
Flaky projects	5	-	-	-	-
Total	122	5	28	17	16

intermittently, regardless of the environment (categorized as *flaky projects*). As such, the summation of projects across all categories exceeds the total number of projects.

Environmental flaky projects exhibit different causes of flakiness. Twenty-one are OS-dependent, three are Node.js version-dependent, and eight are browser-dependent. In addition, the causes of test-level flakiness include seven OS dependent, two Node.js version-dependent, and nine browser-dependent flaky tests. At the test level, we observe two projects that exhibit a mixture of all three causes.

In addition, five projects are flaky due to non-environment-related causes beyond those that we control for. For example, Listing 6.3 shows an asynchronous setup function[7] that intermittently fails with the following error message: `error:Timeout-Asyncfunctiondidnotcompletewithin10000ms(setbyjasmine.DEFAULT\_TIMEOUT\_INTERVAL)`. This timeout failure occurs when the `await` operation (e.g., clearing the database) takes longer than Jasmine's default 10-second timeout, typically due to slow database responses.

```
1 beforeEach(async () => {
2     await new MongoStorageAdapter({ uri: databaseURI }).deleteAllClasses();
3     Config.get(Parse.applicationId).schemaCache.clear();});
```

Listing 6.3: Failure caused by async wait (project: parse-server)

6.3.2 Root Causes of Environmental Flakiness (RQ3.2)

By analysing the failures observed across the 65 environmental dependent projects, we investigate the root causes of test failures and potential causes of test flakiness. We categorized the detected flaky tests into four groups: (1) OS dependency, (2) Node.js version dependency, (3) browser dependency, and (4) combination of the above factors.

We next discuss the results concerning the four categories.

[7] `https://github.com/parse-community/parse-server/blob/e3f5ae0be78361daffee9e8bde148906e9b878f9/spec/MongoStorageAdapter.spec.js#L18`

6.3.2.1 OS dependency

Tests in this category fail when run on a specific OS due to differences in file-system behaviour (e.g., case-sensitivity in file paths), line endings, shell command compatibility, or platform-specific dependencies. In total, we identify 28 environmental dependent projects with failures caused by running on a different OS. Of these projects, 21 exhibit flakiness at the project level, while seven are environmental flaky tests. Regarding OS failures, most occur on Windows (22 of 26 projects), followed by macOS (8) and Ubuntu (2). Some projects exhibit failures on more than one OS.

```
1 "test": "npm run --silent test:config && npm run --silent test:config:base",
2 "test:config": "cd packages/eslint-config-airbnb; npm test",
3 "test:config:base": "cd packages/eslint-config-airbnb-base; npm test",
```

Listing 6.4: Package.json using Unix-style cd commands (project: airbnb/javascript)

An example of a project with OS dependency is shown in Listing 6.4. The listing shows part of the `package.json` file from the Airbnb JavaScript Style Guide project (`airbnb/javascript`)[8] that fails on Windows. The failure occurs due to the `test:config` and `test:config:base` scripts (lines 2 and 3) using Unix-style `cd` commands with semicolons (`;`) to chain commands. On Unix-based systems, the semicolon acts as a command separator, but Windows does not recognize it in the same way. As a result, the shell misinterprets the command and fails to execute the second part of the command. This difference in interpretation leads to errors such as *"The system cannot find the path specified"* and causes the test script to fail. A cross-platform fix would involve replacing semicolons with `&&`, making it work consistently across Unix-based systems and Windows.

Another example of a path-specific issue is shown in Listing 6.5 from the repository `30-seconds-of-code`, where a test fails due to platform-specific handling of file paths. The failure occurs on Windows because the OS returns file paths using backslashes (\) instead of forward slashes (`/`). This mismatch causes the test to fail when comparing the actual output to the expected snippet IDs. To avoid such issues, the test code should normalize file paths using Node.js utilities such as `path.posix` or `path.normalize()` to ensure consistent comparisons across platforms.

Another flaky behaviour in this category occurs in the `dayjs` project (Listing 6.6), where the script fails on Windows because of platform-specific handling of environment variables. The script sets the `TZ` (timezone) environment variable inline before calling `npm`

[8] `https://github.com/airbnb/javascript`

```
1  it('should output the correct snippet ids', () => {
2    expect(snippetData.map(({ id }) => id).sort()).toEqual(
3      [
4        'articles/s/web-development-tips',
5        'js/s/array-grouping',
6        'js/s/array-initialize',
7        'js/s/array-map-foreach',
8        'js/s/array-compare',
9        'css/s/content-centering',
10       'css/s/css-reset',
11     ].sort());});
```

Listing 6.5: OS-dependent test using file paths with backslashes (project: 30-seconds-of-code)

```
1  "scripts": {
2     "test": "TZ=Pacific/Auckland npm run test-tz && TZ=Europe/London npm run ↩
          ↪ test-tz && TZ=America/Whitehorse npm run test-tz && npm run ↩
          ↪ test-tz && jest",
3     ...
```

Listing 6.6: Package.json using Unix-style cd commands (project: dayjs)

`run test-tz` multiple times (line 2). This syntax (`TZ=... command`) is available on Unix-based systems, where one can set environment variables inline. However, this syntax is not natively supported on Windows, resulting in the test failing. To make the script work across multiple OS and avoid potential flakiness, tools such as `cross-env`[9] can be used to set environment variables in a way that works across different OS.

In addition to the previous examples, environmental flakiness can also arise from operating-system-specific dependency constraints encountered during test setup. The `github-profile-readme-generator` project includes the `sharp` library in its dependencies in the `package.json` file (Listing 6.7). During installation, `sharp` attempts to download a pre-built binary of `libvips`, which is unavailable for the `darwin-arm64v8` platform. As a result, the installation fails. This failure is not caused by the test logic itself but by the absence of compatible OS binaries.

6.3.2.2 Node.js Version Dependency

We identified five test failures that occur across different Node.js versions (i.e., a test passes in one Node.js version but fails in another). Those failures were typically caused by breaking

[9] `https://www.npmjs.com/package/cross-env`

```
...
    "gatsby-plugin-sharp": "2.6.14",
    "gatsby-remark-prismjs": "^3.5.10",
    "gatsby-source-filesystem": "^2.3.23",
    "gatsby-transformer-remark": "^2.8.27",
    "gatsby-transformer-sharp": "^2.5.7",
...
```

Listing 6.7: List of dependencies (project: github-profile-readme-generator)

changes in the API, deprecated APIs, stricter error handling, or differences in built-in module implementations. For example, a test may rely on undocumented behaviour that changes between Node.js versions. In total, we identified four environmental dependent projects with failures resulting from differences in Node.js versions. Two of these projects are environmental flaky projects, and two projects include environmental flaky tests, where only specific tests fail depending on the Node.js version.

Listing 6.8 shows a test from the pug repository that expects the error message to follow a specific structure. The test passes when run with Node.js version 15 but fails when run on versions 18, 20, and 22 due to changes in the way error messages are generated [131]. Listing 6.9 shows the error message when running the test with Node.js version 18.

```
it('includes detail of where the error was thrown including the filename', function()↩
     {
    var err = getFileError(
      __dirname + '/fixtures/runtime.with.mixin.error.pug',{});
    expect(err.message).toMatch(/mixin.error.pug:2/);
    expect(err.message).toMatch(/Cannot read property 'length' of null/);});
```

Listing 6.8: Node-dependent test (project: pug)

```
    Expected pattern: /Cannot read property 'length' of undefined/
    Received string: "/home/runner/work/pug/pug/packages/pug/test/fixtures/layout.with↩
        .runtime.error.pug:3
       1| html
       2| body
     > 3| = foo.length
       4| block content
       5|.
    Cannot read properties of undefined (reading 'length')"
```

Listing 6.9: Error message for a node-dependent test with error message format check (project: pug)

Listing 6.10 presents another example of a flaky test that passes on Node.js version 18 but fails on versions 20 and 22. Under Node.js 18, the ES module system exhibits more permissive behaviour in test environments that rely on on-the-fly transpilation of JSX. In this context, the test runner allows certain module resolution and syntax handling steps to succeed even when the module configuration does not explicitly handle JSX code. In contrast, Node.js 20 and 22 enforce stricter ES module semantics, particularly during module loading and syntax validation. These newer versions attempt to parse JSX as standard JavaScript, which results in syntax errors when no appropriate loader or transpilation step is configured[10]. As a result, the observed failure under newer Node.js versions is caused by stricter parsing and validation rules rather than changes in the test logic itself. To resolve this issue, the JSX code must be transpiled appropriately (e.g., using `Babel.js` transpiler) before running the tests, or the test runner configuration must be updated to handle JSX correctly in newer Node.js versions.

```
1 it('should return a valid Provider Component', () => {
2     const { Provider } = createContext();
3     const contextValue = { value: 'test' };
4     const children = [<div>child1</div>, <div>child2</div>];
5     const providerComponent = <Provider {...contextValue}>{children}</Provider>; //↩
          expect(providerComponent).to.have.property('tag', 'Provider');
6     expect(providerComponent.props.value).to.equal(contextValue.value);
7     expect(providerComponent.props.children).to.equal(children);});
```

Listing 6.10: Node-dependent test using JSX (project: preact)

Of the projects we analysed, we found that three of those projects declare a specific Node.js version or version range in their `package.json` file (via the `engines` field). They were different from the versions that we considered for the running environment and caused failures. We did not consider these failures as due to Node.js dependency issues. This choice ensures that we do not report expected incompatibility errors as instances of environmental flakiness.

6.3.2.3 Browser dependency

Some failures typically occur in browser-based or end-to-end tests, where inconsistencies between browser engines (e.g., Chromium vs. WebKit) or browser versions lead to flakiness. Errors can also be specific to certain combinations of browsers and OS (i.e., tests fail

[10] `https://nodejs.org/en/blog/release/v20.0.0/#custom-esm-loader-hooks-run-on-dedicated-thread`

on specific browser setups within a specific OS). In total, we identified 17 projects with browser-related flakiness; eight of them are flaky as projects, while the remaining nine have Environmental flaky tests. The majority of browser-related flakiness observed across eight projects occurs on macOS and affects tests run in Chrome, Firefox, or Safari. These failures are typically caused by differences in browser engine behaviour, missing binaries, or variations in how browser APIs behave across platforms.

Listing 6.11 shows an example of a configuration file from the `axios` project where part of the test suite is configured to run using Firefox. On macOS, these tests fail with the error *"No binary for FirefoxHeadless browser on your platform"*, as the required Firefox binary is not available in the test environment. The error can be fixed by installing Firefox when running tests on macOS.

```
1 "test:karma": "node bin/ssl_hotfix.js cross-env LISTEN_ADDR=:: karma start ↩
    ↪ karma.conf.cjs --single-run",
2 "test:karma:firefox": "node bin/ssl_hotfix.js cross-env LISTEN_ADDR=:: ↩
    ↪ Browsers=Firefox karma start karma.conf.cjs --single-run",
3 "test:karma:server": "node bin/ssl_hotfix.js cross-env karma start ↩
    ↪ karma.conf.cjs",
4 "test:build:version": "node ./bin/check-build-version.js",
```

Listing 6.11: package.json using Firefox browser (project: axios)

We observed a similar browser-related flakiness in the project `jQuery`, where a test fails exclusively on macOS (see Listing 6.12). The test, which checks the native abort behaviour of `jQuery.ajax()`, fails with the message "unexpected success". This message indicates that the test passed when a failure was expected, suggesting a platform-specific difference in how `XMLHttpRequest` abort events are handled in Chrome's headless mode on macOS. Such discrepancies can lead to unreliable outcomes in browser-based test suites across different environments.

6.3.2.4 Combination of Node.js and operating system configurations

Flakiness in this category can be attributed to interactions among multiple environmental factors. For instance, a test may pass under most conditions but fail only on a specific OS with a particular version of Node.js or a browser. Such issues are more complex to diagnose as they involve multiple combinations of environmental factors, making them difficult to reproduce. Given the number of cases and the importance of browsers in JavaScript, we consider the combination of browser and OS to be part of the browser issue, as discussed

```
ajaxTest( "jQuery.ajax() - native abort", 2, function( assert ) {
  return {
    url: url( "mock.php?action=wait&wait=1" ),
    xhr: function() {
      var xhr = new window.XMLHttpRequest();
      setTimeout( function() {
        xhr.abort();
      }, 100 );
      return xhr;
    },
    error: function( xhr, msg ) {
      assert.strictEqual( msg, "error", "Native abort triggers error callback" );
    },
    complete: function() {
      assert.ok( true, "complete" );
    }};} );
```

Listing 6.12: Native abort behaviour (project: jQuery)

previously in Section 6.3.2.3.

In this category, we identified a total of 16 projects exhibiting flakiness attributable to the combination of the OS and Node.js version. Of these projects, eight are Environmental flaky projects, while eight are Environmental flaky tests. In addition, 13 of these projects fail on Windows, eight on macOS, and five failures are observed from combinations involving Ubuntu and Node.js versions.

An example of such flaky behaviour is shown in the `json-server` project (Listing 6.13). 80 tests fail on Windows when run with Node.js versions 18 and 20. This failure transpired because, on Windows, the default command prompt (`cmd.exe`) does not perform glob pattern expansion. As a result, commands like `node --test src/*.test.ts` pass the pattern `*.test.ts` as a literal string to Node.js, causing errors as the test runner cannot resolve the intended files. Since version 22, an improved glob handling was introduced in the built-in test runner for Windows, which can mitigate this issue[11].

In addition to the previous example, similar failures can be observed in projects that rely on glob patterns when invoking the Node.js test runner on Windows. As shown by the error message in Listing 6.14, the test command passes the pattern `test/unit/*.test.js` directly to Node.js, which then attempts to resolve it as a literal file path. On Windows systems using the default command prompt (`cmd.exe`), glob patterns are not expanded before execution. Consequently, when running with Node.js versions 18 and 20, the test runner cannot locate the intended test files and terminates with an error. This behaviour is

[11] `https://github.com/nodejs/node/issues/50658#issuecomment-1806581766`

```
"scripts": {
    "dev": "tsx watch src/bin.ts fixtures/db.json",
    "build": "rm -rf lib && tsc",
    "test": "node --import tsx/esm --test src/*.test.ts",
    "lint": "eslint src",
    "prepare": "husky",
    "prepublishOnly": "npm run build"},
```

Listing 6.13: package.json file using command that does not work on Windows with Node.js versions 18 and 20 (project: json-server)

not related to the test implementation itself, but rather to platform-specific command-line handling and the lack of built-in glob expansion in these Node.js versions. Since Node.js 22, improvements to glob handling in the built-in test runner mitigate this issue by allowing such patterns to be resolved correctly on Windows.

```
> marked@15.0.7 test:unit
> node --test --test-reporter=spec test/unit/*.test.js

Could not find 'D:\a\marked\marked\test\unit\*.test.js'
Error: Process completed with exit code 1.
```

Listing 6.14: error message happend on Windows with Node.js versions 18 and 20 (project: marked)

6.4 Mitigation Strategy (RQ3.3)

To address flaky tests caused by environmental factors, we propose a lightweight mitigation approach that enables developers to selectively skip specific tests or test files under known environmental configurations. By preventing environment-dependent flaky tests from executing in unsupported settings, the approach allows continuous integration (CI) builds to proceed without rerunning entire test suites or halting due to spurious failures, thereby reducing unnecessary developer inspection and CI disruption. As noted in Chapter 4, when flakiness is caused by OS-specific conditions, developers often resort to embedding ad hoc conditional logic in test code to detect the current operating system or runtime configuration and to execute tests only under specific environments. While effective in some cases, such practices lead to scattered, hard-to-maintain test logic and provide limited traceability.

A common industrial practice for dealing with flaky tests is to quarantine them, that is,

to allow the CI pipeline to bypass the test while still reporting it as potentially flaky [99, 126, 127]. This strategy is widely adopted to prevent known unstable tests from repeatedly failing CI builds and disrupting development workflows, while preserving visibility into their existence and behaviour. Several test monitoring and CI support tools implement this idea by identifying flaky tests based on their behaviour in previous executions and automatically quarantining them under certain conditions.

Motivated by the prevalence of this practice, this section develops a practical mitigation approach for flakiness caused by environmental variation. Rather than attempting to immediately repair environment-dependent flaky tests, which may require substantial effort, deep domain knowledge, or expensive analysis, we focus on a lightweight, developer-controlled mechanism that allows such tests to be selectively skipped under known problematic environments, while remaining explicitly documented and traceable. We emphasise that we do not claim skipping flaky tests to be superior to fixing them. Instead, quarantining acknowledges the presence of flakiness while avoiding unnecessary delays to the test workflow, particularly for otherwise stable (healthy) tests. This approach has minimal overhead and aligns with how developers manage flakiness in practice, whereas automated fixing techniques, although valuable, are typically more expensive and often rely on hybrid analyses or learning-based methods.

Importantly, the proposed mitigation approach can support the fixing process rather than replace it. Tests are skipped only when developers explicitly annotate them, and the approach generates detailed reports that document the affected tests and the environments under which failures occur. This information provides a clear starting point for diagnosing and ultimately repairing the underlying issues, while ensuring that tests continue to run in environments where they are valid and meaningful.

To this end, we introduce js-env-sanitizer, a lightweight, annotation-based mitigation approach that operates through analysis of test files, enabling environment-dependent tests to be conditionally skipped with minimal overhead and without requiring repeated test executions across multiple environments.

6.4.1 Approach

The primary goal of our lightweight approach to addressing environment-specific flaky tests is to enable developers to manage them via annotations. Developers annotate their tests with docblock[12] tags, specifying the conditions under which a test should be skipped or

[12] `https://developer.adobe.com/commerce/php/coding-standards/js-docblock/`

enabled. This approach eliminates the need for ad hoc conditional logic in test code, thereby promoting cleaner and more maintainable test suites. Tests that are intentionally skipped through our annotations are treated distinctly from tests that are skipped for other reasons, such as tests that developers have manually disabled (e.g., via `test.skip`, `it.skip`, or equivalent constructs) or tests that are marked as pending because their implementation is incomplete or temporarily deferred. All annotation-driven skipped tests are explicitly reported in a generated test report, which shows which tests are skipped and the identified possible reason for the flakiness.

The approach is designed to support multiple JavaScript testing frameworks, including Jest, Mocha, and Vitest, as these frameworks collectively cover a large portion of the JavaScript testing ecosystem and represent the dominant testing styles used in both browser-based and Node.js environments. Despite differences in their APIs, they share a common abstraction of test cases and test suites and provide explicit mechanisms for skipping or deferring test execution. By relying on static code transformation rather than framework-specific runtime hooks, the approach avoids requiring modifications to existing test logic and can be applied uniformly across these frameworks. The core components of the proposed approach are:

- **Annotation Parser:** Extracts metadata from test file docblocks, identifying *skip* or *enable* conditions.
- **Environment Detector:** Detects, at runtime, the current operating system, Node.js version, and the active browser.
- **Condition Evaluator:** Compares the extracted annotations with the detected environment to decide whether a test should be skipped.
- **Test Skipping Module:** Marks tests as skipped and logs skip reasons for review, before test execution.

The tool's high-level workflow is as follows: when a test file is loaded, it parses all docblock annotations, evaluates them against the runtime environment, and automatically skips tests that do not meet the specified conditions. This design ensures consistent, automated enforcement of environment-specific constraints across testing frameworks.

The approach provides a set of annotation tags that can be used to enable/disable tests on specific environment configurations: OS types, Node.js versions, and browser types. Table 6.4 lists all the annotation tags we defined, along with their descriptions. Tag names

Table 6.4: Annotation tags supported by `js-env-sanitizer`

Type	Tag	Effect when condition holds
OS	@enableOnOs @skipOnOs	Run only (or skip) on the specified OS.
Node.js	@enableOnNodeVersion @skipOnNodeVersion	Run only (or skip) on the specified Node.js version(s).
	@enableOnNodeRange @skipOnNodeRange	Run only (or skip) when Node.js satisfies the given version range.
Browser	@enableOnBrowser @skipOnBrowser	Run only (or skip) when the active browser matches one of the specified values.

and values are case-insensitive, and lists are comma-separated. Each test will run as the default unless a `skip` condition matches. An example of how an annotation tag can be added before a test block to ensure it is skipped under the specified conditions is shown in Listing 6.15. In this example, the annotation `@skipOnOS win32` instructs the tool to skip the test when executed on Windows, but to run it normally on other OSs.

```
/**
  * @skipOnOS win32
  */
  describe('all', () => {
    it('returns all records', () => {
      expect(User.all).toEqual([user1, user2]);
    });
  });
```

Listing 6.15: Example of using an annotiation tag with `js-env-sanitizer` to skip a test from 30-seconds-of-code

The proposed approach builds on a similar conditional test execution implemented in JUnit (for Java tests)[13]. This approach utilizes annotations to specify environmental conditions under which tests should (or should not) run. In JUnit, the annotations *@EnabledOnOs @DisabledOnOs* enable and disable tests on specific OS. This approach is common as it is widely adopted in JUnit tests. A search on GitHub Java repositories shows more than 7,000 uses of *@EnabledOnOs* [14] and more than 6,000 uses of *@DisabledOnOs* [15]. A similar approach has also been adopted in a previous tool, *saflate*, which sanitizes flaky tests due to network dependencies (i.e., tests that fail due to network availability exceptions in JUnit) using custom annotations added to test files or tests [132].

[13] `https://docs.junit.org/current/user-guide/#writing-tests-conditional-execution`
[14] `https://github.com/search?q=@EnabledOnOs+language%3AJava&type=code`
[15] `https://github.com/search?q=@DisabledOnOs+language%3AJava&type=code`

Our approach employs a similar strategy for JavaScript tests, aiming to develop an expandable, cross-framework solution that works across popular JavaScript testing frameworks. It introduces comparable functionality through static code transformation, enabling developers to skip tests based on simple annotations in their test files. In this context, it is worth noting that JavaScript testing frameworks such as Jest expose a range of test execution outcomes, including passing and failing tests, explicitly skipped tests (e.g., via `test.skip` or conditional logic), and pending tests that are declared but not yet implemented. Other JavaScript testing frameworks, such as Mocha and Jasmine, provide similar outcome categories through constructs like skipped, pending, or excluded tests, although the terminology and APIs differ. These outcomes serve different purposes: failures indicate violations of expected behaviour, while skips and pending tests are commonly used to manage known limitations, environmental constraints, or incomplete test specifications. This mirrors, at a conceptual level, richer outcome models in frameworks such as JUnit, which further distinguish errors and uncaught exceptions from assertion failures. Our approach deliberately targets controlled skipping as a first-class outcome, allowing developers to encode environment-dependent conditions explicitly, without conflating them with genuine test failures or unexpected runtime errors.

6.4.2 Implementation

The tool, `js-env-sanitizer`, is implemented as a Babel plugin[16], which extends the test runner to intercept the execution pipeline before tests run. It processes the test suite by scanning the docblock annotations associated with each test. These annotations define constraints (i.e., OS, Node.js version, or browser) under which the test should be run.

The steps taken to run or skip tests are detailed in Algorithm 4. At first, `js-env-sanitizer` detects the current system runtime environment (i.e., OS, Node.js version, and the browser). Each test's annotations are then parsed and compared against this environment. If the conditions indicate that a test should not run (e.g., `@skipOnOS win32` when the OS is Windows), the test is marked as such in a test log saved in the project directory. Otherwise, the test is preserved for execution.

This approach ensures that environment-specific tests are automatically skipped without manual intervention, improving test stability and reproducibility. `js-env-sanitizer` operates without modifying the original test code, making it compatible with existing projects and facilitating easy integration into continuous integration workflows.

[16] `https://babeljs.io/docs/plugins`

Algorithm 4 Test sanitization using annotations

Input: Test suite
Output: Modified ineligible test blocks marked as skipped

Load test suite
Detect current OS, Node.js version, and browser (if applicable)
for each testing block in test suite **do** ▷ describe, it or test blocks
    if *it has supported annotation tags before it* **then**
        shouldSkip ← ! (environment matches all enable-conditions) ∧ (environment matches any skip-condition)
    end if
    if *shouldSkip* **then**
        Change the state of the test to skip
        Log skip reason with timestamp and matched condition
    end if
end for
Return test suite for execution

The tool, `js-env-sanitizer`, currently supports the following JavaScript testing frameworks: Jest, Mocha, and Vitest. However, the tool is extensible and can be readily expanded to support other testing frameworks.

6.4.3 Evaluation

We evaluated the tool using a subset of projects from our dataset (presented in Section 6.2.1). We selected the projects from our dataset that meet the following criteria: (1) use one of the supported test frameworks (Jest, Mocha, or Vitest) and (2) contain at least one environmental flaky test. We start with 15 candidate projects. Among these projects, we found that six use multiple testing frameworks or runners. In those cases, we observed that flaky tests occur in unsupported frameworks. For example, `swagger-ui` uses both Jest and Cypress tests. Upon investigation, we found that the two detected environmental flaky tests occurred in the Cypress test suite, which `js-env-sanitizer` does not currently support. We therefore exclude those six projects from the evaluation and report results on the remaining projects. The projects we included in our evaluation are three projects that use Jest, four that use Mocha, and two that use Vitest.

By declaring `js-env-sanitizer` under a project's `devDependencies`, the tool's package auto-wires itself during installation and activates at test execution time without any manual setup. It detects the host test runner (Jest, Mocha, or Vitest) and attaches the appropriate

lightweight integration (e.g., *babel-jest* transform, a minimal *@babel/core* require-hook for Mocha, or a vite Babel plugin for Vitest).

For each project, we add `js-env-sanitizer` as a dependency to the project's package file and apply the appropriate annotation based on the experiments described in Section 6.3. We then push these changes to the forked projects and run the project's GitHub Actions CI workflow, which executes the test suite across nine environment configurations, each repeated 10 times.

In two projects, *insomnia* and *mongoose*, we were unable to evaluate this sanitization approach. The project *insomnia* features multiple subprojects and workspaces that share a common `Vitest` configuration. This made the instrumentation more challenging, as the test framework configuration file is not localized to a single package but instead inherited across different workspaces. Consequently, our setup script could not fully integrate with the heterogeneous project setup, and some tests executed before the sanitizer hooks could not be properly run. Similarly, the project *mongoose* uses the `Mocha` testing framework, and most of its tests require a live *MongoDB* instance to execute. *mongoose* already includes its own mechanism to skip specific database-dependent tests when a required environment variable (`MONGOOSE_SHARD_TEST_URI`) is not set (an interestingly similar sanitization approach to ours that targets database configurations). However, once we added `js-env-sanitizer`, this built-in skipping mechanism did not take effect as expected, and the connection attempts were still triggered. In practice, the tests that should have been skipped by *mongoose*'s own logic were executed anyway, leading to failures. Therefore, we excluded both *insomnia* and *mongoose* from our evaluation and reported the results for the remaining seven projects.

Table 6.5 summarizes our evaluation results for seven projects. Shaded cells indicate the number of test failures due to environmental dependency, followed by the number of skipped tests after applying our sanitization tool. The detailed evaluation results are available in our replication package [121].

As shown in Table 6.5, the results show that `js-env-sanitizer` can successfully sanitize and skip all tests that have failed due to environmental (OS, Node.js, or browser) dependencies. In other words, the initially failing tests have all been successfully quarantined and reported as skipped, allowing the CI build to continue without the known test failure. `js-env-sanitizer` is also shown to have a low performance overhead (see the runtime columns, runtime values are averages across the 90 runs). For most projects, we observe no significant differences in average runtime before and after applying `js-env-sanitizer`. A notable exception is `fabric.js`, where the average runtime drops significantly. In this case, skipping the tests allowed the CI to continue, which reduces the runtime.

Listings 6.16 and 6.17 demonstrate the use of annotation tags to skip environmental flaky tests, corresponding to the examples in Listings 6.10 and 6.5 in Section 6.3, respectively. Before applying the annotation in Listing 6.16, the test in `preact` project is executed across all evaluated Node.js versions and consistently fails on Node.js versions 20 and 22 due to version-specific behaviour, while passing on version 18. After adding the annotation tag "`@skipOnNodeVersion 20,22`", the test is automatically skipped when executed on Node.js versions 20 and 22 and continues to run normally on version 18. This prevents known, version-dependent failures from being reported as test failures, while preserving test coverage in supported environments.

Before annotation in Listing 6.17, the test in `30-seconds-of-code` project is executed on all OSs and fails on Windows due to its reliance on Unix-like file paths. After applying the annotation tag "`@skipOnOS win32`", the test is systematically skipped on Windows, while still being executed on Unix-based systems where it behaves correctly. This avoids OS-specific false failures without altering the original test logic.

```
1 /**
2   * @skipOnNodeVersion 20,22
3   */
4   it('should return a valid Provider Component', () => {
```

Listing 6.16: Example of an annotation tag to skip a test on Node.js versions (project: preact)

```
1 /**
2   * @skipOnOS win32
3   */
4   it('should output the correct snippet ids', () => {
```

Listing 6.17: Example of using annotation to skip a test on Windows (project: 30-seconds-of-code)

6.5 Threats to validity

The study uses open-source JavaScript projects available on GitHub. The observations regarding environmental flakiness are limited to the selected projects and may not be fully generalizable to all JavaScript projects. To mitigate the threat of generalizability, we selected popular (ranked by stars) repositories maintained by large communities and spanning diverse application domains, including servers, front-end and back-end frameworks, and web

Table 6.5: Results of sanitizing environmental flaky tests using `js-env-sanitizer`

Framework	Project	#tests	Default tests run					Sanitized Tests			
			#failure	#passed	#skipped	Avg runtime (s)	Cause	#failure	#passed	#skipped	Avg runtime (s)
Jest	iptv	8	8	7	0	31	OS & Node	0	8	8	77
	fabric.js	162	99	61	2	1020	OS & Node	0	61	101	65
	pug	650	2	648	0	59	Node range	0	648	2	85
Mocha	rollup	5213	1	5212	0	718	OS & Node	0	5212	1	721
	sails	1550	1	1536	13	237	Node	0	1536	14	238
	preact	1217	1	1203	13	68	Node	0	1203	14	147
Vitest	30-seconds-of-code	148	17	131	0	40	OS	0	131	17	63

applications. The selected projects also use a variety of testing frameworks. We believe the projects are representative.

Part of the project selection and the analysis of the cause of flakiness was performed manually, which may introduce construct validity threats due to potential oversight or bias. The manual analysis consisted of two stages: (1) installing and investigating the tests in each project to observe and classify the impact on environmental flakiness, and (2) installing `js-env-sanitizer` and manually adding annotation tags to skip the environmental dependent tests under the corresponding environments. To mitigate bias, two authors independently examined the source code and error messages to reduce false positives in classification. In cases of disagreement, a third author independently reviewed the instance, and all three authors discussed the causes until consensus was reached.

6.6 Summary

Environmental factors are among the least-studied yet most practically significant sources of test flakiness, particularly in the JavaScript ecosystem, where software is routinely executed across heterogeneous platforms and configurations. This chapter presented a systematic investigation of environment-dependent flakiness in JavaScript projects, focusing on three key environmental dimensions: operating systems, Node.js versions, and browsers. Through controlled reruns across multiple environment combinations, we demonstrated that variations in execution environment can lead to non-deterministic test outcomes even when the code under test remains unchanged.

Our empirical results showed that operating system dependency is the most prevalent form of environmental flakiness, commonly arising from differences between Windows and Unix-like systems, including path handling, filesystem behaviour, and the use of OS-specific

APIs. We also observed flakiness triggered by changes in Node.js versions, often due to compatibility issues, unsupported features, or subtle differences in runtime behaviour. In addition, browser-dependent failures were identified, reflecting inconsistencies across browser engines or missing binaries in specific environments. While many of these failures manifested immediately after changing the environment, some required multiple reruns under the same configuration to surface, highlighting that environmental flakiness may manifest as either consistent environment-specific failures or intermittent behaviour.

In total, we identified 65 projects out of 116 that exhibited environment-dependent flaky behaviour. Under our definition, such behaviour includes both intermittent passfail outcomes and failures that consistently occur only under specific environments, thereby capturing compatibility issues as a subset of environmental flakiness. This perspective aligns with prior work in other ecosystems, where platform-specific incompatibilities are recognised through the same symptom: non-deterministic outcomes across environments.

To mitigate the impact of environmental flakiness, this chapter introduced a lightweight, annotation-based approach implemented in the tool `js-env-sanitizer`. Rather than attempting to repair the underlying causes, which may require substantial manual effort or expensive analysis, the approach enables developers to explicitly document and control known environment-dependent tests by conditionally skipping them under unsupported configurations. The tool integrates with widely used JavaScript testing frameworks and operates via static code transformation, eliminating the need for repeated execution across multiple environments.

By skipping explicitly annotated tests and generating detailed reports that record affected tests and environments, the proposed approach preserves transparency while reducing test result noise. This allows continuous integration pipelines to proceed without being disrupted by known environment-specific failures, while still providing developers with clear guidance for future diagnosis and repair. Overall, this chapter's findings highlight the prevalence and diversity of environmental flakiness in JavaScript and demonstrate that lightweight, environment-aware mitigation can play a practical role in improving test reliability without compromising test effectiveness.

Chapter 7

Discussion

This chapter synthesizes the findings presented throughout this thesis to provide a unified interpretation of our investigation into various aspects of flaky tests in JavaScript projects. By integrating evidence from the empirical study of declared flaky tests in JavaScript projects (Chapter 4), the investigation of test order-dependency (Chapter 5), and the investigation of environment-dependent tests (Chapter 6), the discussion presented here provides a single, coherent body of work that discuss the finding in the larger context. The discussion draws not only on empirical observations but also on insights gained from developing and evaluating lightweight detection and mitigation approaches, which help validate and contextualize observed patterns of flakiness in practice.

7.1 A Unified View of Flakiness in JavaScript

Taken together, the results of this thesis show that flaky tests in JavaScript are primarily driven by interactions between tests and their execution context, rather than by nondeterminism in test logic alone. Although similar interaction-based causes of flakiness have been reported in other languages and testing ecosystems [7, 60], these factors play a particularly central role in JavaScript. Across all our studies, flakiness consistently emerged from factors such as asynchronous execution, operating system behaviour, and runtime configuration, while test order dependency played a more limited but still observable role. In particular, we observe a notably low number of order-dependent tests in JavaScript projects compared to other studied languages, especially Python [56, 60] and Java [7, 55], where order-dependent tests are among the most prevalent types of flaky tests.

This unified perspective suggests that flakiness in JavaScript should be understood as an

ecosystem-level phenomenon. JavaScript tests are rarely executed in isolation; instead, they are embedded within a complex execution environment that includes a runtime environment such as Node.js or React, operating system services, package managers, test frameworks, and continuous integration (CI) infrastructure. As a result, test instability often reflects implicit assumptions about the execution environment rather than defects in the code under test. Viewing the results through this lens provides a common explanatory framework that connects the empirical observations across the different chapters of this thesis.

The feasibility of systematically exposing and managing flaky behaviour using targeted tooling further supports this interpretation. The fact that both order-dependent behaviour and environment-dependent instability can be reliably detected and addressed using lightweight approaches indicates that these issues are not sporadic anomalies but recurring patterns that can be reasoned about at the ecosystem level. This reinforces the view that flakiness in JavaScript is shaped by execution context and shared infrastructure rather than isolated test defects.

Our findings also illustrate that the boundary between compatibility issues and environmental flakiness is largely conceptual rather than observable in test outcomes. In some cases, failures attributed to compatibility differences manifest through the same symptom as flakiness: tests that produce different outcomes across execution environments despite the code remaining unchanged. As a result, a subset of the cases identified in this study is fundamentally flaky in nature, in the sense that their behaviour is inherently unstable across environments rather than tied to a single, well-defined failure condition. From a testing perspective, compatibility-related failures are therefore detected, experienced, and mitigated in the same way as environmental flakiness, which motivates their inclusion within a unified analytical framework.

Table 7.1 summarises the main similarities and differences between the JavaScript findings of this thesis and flaky-test causes reported in other programming ecosystems. The comparison shows that flaky tests in JavaScript share several characteristics with those reported in Java, Android, C++, and Python, but differ in their relative distribution. Similar to prior studies, we found that concurrency, asynchronous behaviour, network interaction, timing behaviour, and shared state can lead to nondeterministic test outcomes [7, 8, 25, 60]. These causes, therefore, appear to be general sources of flakiness across programming ecosystems. However, the JavaScript results differ in the prominence of some causes. In our empirical study, the four most common causes were concurrency (20.7%), async wait (19.6%),

Table 7.1: Comparison of flaky-test causes reported across programming ecosystems

Study	**Language / Ecosystem**	**Prominent causes reported**	**Comparison with JavaScript findings**
Luo et al. [7]	Java	Async wait, concurrency, test order dependency, resource leak, network, time, I/O, randomness, floating-point operations, unordered collections	Shares several causes with JavaScript, especially async wait, concurrency, network, and time-related behaviour. However, test order dependency was more prominent in Java than in our JavaScript results.
Thorve et al. [8]	Android	Concurrency, dependency, program logic, network, and UI	Similar to JavaScript in the importance of concurrency, network, dependency, and UI-related behaviour. However, our JavaScript results show a stronger role for OS and environment-related causes.
Dutta et al. [61]	Probabilistic and machine-learning systems	Algorithmic non-determinism, floating-point computations, incorrect or flaky API usage, unsynchronised seeds, concurrency, and hardware	Some causes, such as concurrency and hardware-related variability, overlap with JavaScript. However, the dominant causes in this study are more specific to probabilistic and machine-learning systems.
Gruber et al. [60]	Python	Infrastructure, test order dependency, network, randomness, I/O, time, async wait, concurrency, resource leak, test case timeout, and unordered collections	Shares causes such as network, time, async wait, concurrency, and infrastructure-related failures. However, test order dependency was more prominent in Python than in our JavaScript study.
This thesis	JavaScript	Concurrency, async wait, OS dependency, network, platform/environment dependency, UI, hardware, time, and resource leak	JavaScript flakiness is strongly shaped by asynchronous execution and heterogeneous execution environments, including operating systems, Node.js versions, browsers, package managers, shell environments, and CI configurations.

OS dependency (18.4%), and network dependency (12.6%). Together, these categories accounted for more than 70% of the flaky-test commits analysed in Chapter 4. This distribution contrasts with studies of Java and Python, where order dependency has been reported as a more prominent cause of flakiness [7,66]. In our JavaScript study, order dependency appeared only rarely in the commit-based evidence, and the systematic reordering experiment in Chapter 5 detected 55 order-dependent tests across 10 of the 81 evaluated projects.

The comparison also highlights the stronger role of environmental factors in JavaScript. Although platform and infrastructure-related flakiness has been reported in other ecosystems, it was not usually among the dominant categories in prior studies. In contrast, OS dependency was one of the top three causes in our empirical study, and Chapter 6 further showed that environment-dependent failures are widespread across JavaScript projects. This difference is likely related to the JavaScript ecosystem itself, where projects commonly execute across different operating systems, Node.js versions, browsers, package managers, shell environments, and CI configurations.

Overall, this comparison suggests that flakiness is both a general testing problem and an ecosystem-specific phenomenon. While many broad causes recur across languages, their prevalence and manifestation depend on language features, runtime models, testing frameworks, and development practices. For JavaScript, asynchronous execution and heterogeneous execution environments make concurrency-related and environmental flakiness particularly important, while order dependency appears less prevalent than in Java and Python.

7.2 Environmental Flakiness as a Dominant Characteristic

One of the key findings of this thesis is the prominence of environmental flakiness in JavaScript projects, particularly flakiness caused by platform dependencies. While we identified OS-related issues as a dominant cause of flaky tests (Chapter 4), the results presented in Chapter 6 further reveal that such dependencies are widespread and persistent across real-world projects. Considered together, these findings demonstrate that OS-dependent flakiness is not an incidental artifact of individual projects, but a structural characteristic of contemporary JavaScript testing.

A key reason for this prominence lies in JavaScripts close interaction with its execution environment. JavaScript tests frequently rely on file system behaviour, process invocations, shell commands, and external tools. Differences in path resolution, shell semantics, file permissions, and default system utilities across operating systems can therefore directly affect test outcomes. Unlike ecosystems that provide stronger abstraction barriers around

such interactions, JavaScript exposes many of these platform-specific details to developers, increasing the likelihood that tests implicitly depend on environmental behaviour.

Development practices in the JavaScript ecosystem further amplify this effect. Tests are often written and validated on a single development platform, with cross-platform execution deferred to CI. As a result, environment-specific assumptions may remain latent until tests are executed under different configurations, at which point failures manifest as flaky behaviour.

Although environmental flakiness may appear, in principle, to be language independent, the findings of this thesis indicate that its practical impact is strongly shaped by ecosystem conventions. Differences in tooling, runtime models, and testing practices influence how environmental variability is exposed. In the JavaScript ecosystem, the combination of heterogeneous tooling, flexible execution models, and cross-platform deployment goals appears to increase susceptibility to environment-dependent flakiness.

The effectiveness of the environment-aware mitigation approach evaluated in Chapter 6 further emphasizes the structural nature of these issues. The ability to manage environment-dependent tests without modifying test logic or relying on repeated executions suggests that such flakiness stems from stable, identifiable environmental constraints rather than transient or random behaviour. This observation strengthens the argument that environmental flakiness in JavaScript is both systematic and predictable once the execution context is made explicit, regardless of the testing framework used.

7.3 Order-Dependent Tests in JavaScript

In contrast to environmental causes, order-dependent flakiness was observed less frequently in the JavaScript projects. As shown in Chapter 5, only 55 order-dependent tests were detected across 10 projects out of 81 evaluated projects, and all identified cases were caused by shared state between tests. These included shared files and shared mocking state, with the latter representing a previously unreported cause of order dependency in JavaScript testing.

The nature of these dependencies is noteworthy. Rather than arising from explicit coupling between test cases, order-dependent behaviour often resulted from incomplete cleanup or implicit assumptions about test isolation. In particular, shared mocking state reflects the extensive use of mocking frameworks in JavaScript test suites and highlights how test infrastructure itself can introduce hidden dependencies.

The targeted detection approach used to identify order-dependent tests also provides insight into the scope and limits of this problem. While the approach was effective at exposing dependencies caused by shared state, the relatively small number of detected cases suggests that order dependencies are not pervasive across JavaScript test suites. Instead, it appears to arise under specific conditions, particularly when test infrastructure introduces implicit shared state. This supports the conclusion that order-dependent flakiness, while important, is less common compared to broader environment-related instability. We attribute this low number of order-dependent tests to two main factors: 1) the way JavaScript developers organize their tests when using Jest, where related tests are typically grouped within the same blocks and test files, and 2) the way testing frameworks such as Jest handle test execution order, running test suites in parallel by default while executing tests or `describe` blocks in the order they appear within each file.

7.4 Implications for JavaScript Developers

For JavaScript developers, the findings of this thesis underscore the importance of treating the execution environment as a first-class concern in test design. Tests that interact with the filesystem, operating system, or external tools should make their environmental assumptions explicit wherever possible. When such assumptions cannot be eliminated, conditional execution or selective skipping under unsupported configurations offers a pragmatic alternative to repeated reruns or ad hoc fixes.

The results also caution against interpreting test stability solely on the basis of local execution. Tests that consistently pass on a single platform may still exhibit latent environment-dependent behaviour that manifests only under different configurations. Incorporating environment-aware testing strategies earlier in the development lifecycle can help surface these issues before they lead to persistent flakiness in CI.

With respect to order-dependent behaviour, developers should pay particular attention to shared mocking state and test setup practices. Ensuring proper cleanup between tests and consistently using framework-provided isolation mechanisms can substantially reduce the likelihood of hidden dependencies.

7.5 Implications for Tool Support and CI Practice

From a tooling perspective, the findings of this thesis indicate that effective support for flaky tests in JavaScript should prioritize explicit reasoning about execution context. The success

of the lightweight detection and mitigation approaches evaluated in this work suggests that practical tools do not need to rely on deep analysis or extensive reruns. Instead, declarative mechanisms that expose environmental assumptions and shared state can improve test reliability while remaining compatible with existing development workflows.

For CI systems, these findings suggest that repeated execution alone is an inadequate response to flakiness. While rerunning tests may temporarily mask instability, it does not address underlying causes and can increase build time and developer frustration. CI pipelines can benefit from explicit support for environment-aware execution, clearer reporting of environment-dependent failures, and mechanisms that distinguish between genuine regressions and known environmental limitations.

7.6 Implications for Future Research

The findings of this thesis open several avenues for future research. First, further empirical studies are needed to examine how environmental flakiness evolves as JavaScript tooling, runtimes, and execution models continue to change. Second, incorporating environmental awareness into automated test analysis, classification, and repair techniques is a promising approach to improving test reliability. Finally, extending this line of work to emerging JavaScript runtimes and execution environments may provide additional insight into how ecosystem-level factors influence test stability.

The tools developed in this thesis also serve as a foundation for future research, illustrating how execution-context information can be surfaced and leveraged to support more advanced analysis, classification, and repair of flaky tests.

7.7 Summary

In summary, this thesis demonstrates that flaky tests in JavaScript are best understood as the result of interactions between tests and their execution context, rather than as isolated defects in test logic. Environmental variability, particularly operating-system dependencies, plays a central role in shaping test outcomes, whereas order-dependent behaviour is a more limited but subtle source of instability. By synthesizing empirical evidence, detection techniques, and mitigation strategies, this discussion frames JavaScript flakiness as an ecosystem-level challenge and provides a foundation for more robust testing practices and tool support.

Chapter 8

Conclusion

8.1 Concluding Remarks

This thesis provides an in-depth investigation of flaky tests in JavaScript projects, a crucial issue in software testing and quality assurance. The goal of the work is to improve the reliability and trustworthiness of automated testing in this increasingly dominant software ecosystem. As JavaScript applications continue to grow in scale and complexity, their heavy reliance on asynchronous execution, diverse runtime environments, and platform-specific configurations introduces new challenges for test determinism. Flaky tests, which fail intermittently without corresponding code changes, undermine developer confidence in test results, slow down continuous integration pipelines, and increase maintenance costs.

This thesis adopted an empirical paradigm and combined large-scale repository mining and systematic experimentation, supported by tool development, to study flaky behaviour from three complementary perspectives. First, we examined the main causes of flaky tests and how developers respond to them in practice. Second, we systematically investigated order-dependent flakiness, a well-studied issue in other ecosystems but largely unexplored in JavaScript. Third, we investigated environmental flakiness and proposed a lightweight mitigation strategy to control tests that are sensitive to execution environments.

Together, the findings of this thesis provide an in-depth, evidence-based understanding of flaky tests in JavaScript, highlight how this ecosystem differs from others, and demonstrate practical techniques for detecting and mitigating two important classes of flakiness.

This thesis was guided by three main research questions, each corresponding to a distinct but interrelated objective. To address RQ1, this thesis conducted a large-scale empirical study of flaky-test-related commits in popular JavaScript projects hosted on GitHub. By

manually analysing commit messages, code changes, and associated issue discussions, the study identified the dominant causes of flakiness and documented how developers deal with them in practice.

The results show that flakiness in JavaScript is primarily driven by concurrency, async-wait, operating system dependencies, and network-related issues. Together, these causes account for more than 70% of the observed flaky-test commits. While concurrency and async-wait issues have also been reported as major causes in other languages [7, 25, 60], this thesis highlights the particularly strong role of platform-related issues in JavaScript, especially operating system variations. This finding reflects the cross-platform nature of JavaScript development and the reliance on underlying system behaviour, such as file systems, path handling, and timing.

Regarding developer responses, the study found that most flaky tests are fixed rather than ignored. The majority of flaky tests are fixed by modifying either the test code or the code under test, while a smaller proportion are skipped, quarantined, improved, or removed. These findings suggest that developers generally perceive flaky tests as a problem, but also rely on pragmatic strategies when immediate fixes are costly or unclear. Overall, the results of RQ1 establish an empirical foundation for understanding flaky tests in JavaScript and motivate the more targeted investigations conducted in the subsequent chapters.

Building on the empirical insights from RQ1, RQ2 focused on order-dependent flakiness, a phenomenon in which test outcomes depend on the order of test execution. While order-dependent tests have been extensively studied in Java [52,55,62,63,65] and Python [56,66], little was known about their prevalence and characteristics in JavaScript.

To address this gap, this thesis uses a systematic test reordering approach, implemented in *JS-TOD*, a tool specifically designed for JavaScript and the Jest testing framework. *JS-TOD* operates by extracting tests and `describe` blocks using abstract syntax tree analysis, generating multiple randomised execution orders at different levels, and rerunning the reordered test suites to observe changes in test outcomes.

The evaluation showed that order-dependent flakiness does occur in JavaScript projects, though it is less prevalent than in some other ecosystems. The detected cases were primarily caused by shared states between tests, namely shared files and shared mocking state. Notably, flakiness related to mocking behaviour emerged as an important and previously underreported cause of order dependency in JavaScript. The study also found that alternative detection strategies, such as parallel execution or isolated test runs, were less effective at revealing order-dependent behaviour than systematic reordering.

These results demonstrate that order-dependent flakiness in JavaScript exhibits distinct

characteristics and requires dedicated detection techniques. Our proposed approach addresses an important gap by providing a lightweight, configurable, and framework-integrated method for exposing hidden order dependencies that may not surface under default execution settings.

RQ3 focuses on environmental flakiness, which arises when tests fail due to variations in execution environments rather than changes in code or test logic. Motivated by the prominence of platform-related issues observed in RQ1, this thesis conducted a systematic evaluation of JavaScript projects across different operating systems, Node.js versions, and browsers.

The results show that environmental flakiness is both common in JavaScript projects and of a diverse nature. Tests may fail only on specific operating systems, with particular Node.js versions, in certain browsers, or in some cases due to combinations of these factors. Many of the identified issues were related to OS path handling, unsupported features, fragile assertions, or differences in browser behaviour.

To mitigate these issues, this thesis proposed a lightweight, annotation-based approach to sanitize environment-dependent tests, implemented in a tool called *js-env-sanitizer*. The approach follows a quarantine-based strategy, a practice commonly used in industry, in which tests that are known to be unstable under specific conditions are temporarily isolated from regular execution. This mitigation approach allows developers to explicitly specify the environments in which a test is expected to run correctly. Tests that do not satisfy these conditions are skipped, while detailed reports document the reasons for skipping and the targeted environments.

Importantly, this approach is not intended to replace fixing flaky tests, nor does it suggest that skipping is preferable to repair. Instead, it acknowledges the practical realities of continuous integration workflows, where known environment-dependent failures can significantly delay feedback and obscure genuine regressions. By quarantining only those tests that developers have explicitly identified as environment-sensitive, the approach reduces noise in test results while preserving the integrity of the remaining test suite. The overhead of this strategy is minimal, making it suitable for routine use in CI pipelines.

Moreover, the mitigation approach can act as a first step in the fixing process rather than an endpoint. By generating detailed reports that identify affected tests and the environments where failures occur, the approach provides developers with concrete information about the conditions that trigger flaky behaviour. This explicit characterisation of environmental constraints provides a clear starting point for diagnosing and, when time and resources permit, ultimately repairing the underlying issues.

8.2 Summary of Contributions

The main contributions of this thesis can be summarised as follows:

- A large-scale empirical analysis of flaky tests in JavaScript projects, providing the first detailed, commit-level characterisation of flakiness causes and developer responses in this ecosystem, and highlighting the prominent role of environmental variability.
- A systematic investigation of order-dependent flakiness in JavaScript, including the design, implementation, and evaluation of `JS-TOD`, the first dedicated tool for detecting order-dependent flaky tests in Jest-based projects.
- A comprehensive investigation of environmental flakiness in JavaScript and the design of a lightweight, annotation-based mitigation strategy that enables developers to explicitly manage environment-dependent tests in continuous integration environments.
- Actionable insights for developers and tool builders on improving test reliability in JavaScript projects, informed by empirical evidence and tool-based evaluation, with particular relevance to modern CI workflows.

8.3 Limitations and Reflections

As with any research study, this work has several limitations. The empirical analyses relied on open-source JavaScript projects hosted on GitHub and may not fully represent proprietary or industrial codebases. Future work could address this limitation by studying industrial systems through industry collaborations or by analysing internal test repositories where access is available.

In addition, the order-dependent flakiness study focused on projects using the Jest framework. Although Jest is the most widely used JavaScript testing framework, other frameworks may differ in how they structure test execution, manage shared state, or handle concurrency. As a result, the findings may not generalise directly without adaptation. This limitation could be mitigated by extending the approach to additional frameworks, such as Mocha or Vitest, and by evaluating whether similar patterns of order-dependent flakiness arise under different testing models.

In addition, both the order-dependent and environmental studies used a finite number of reruns and configurations. Although these settings were informed by prior work [52, 55, 56] and pilot experiments, increasing the number of reruns or environments could reveal

additional flaky behaviour. However, such increases would substantially raise computational cost and execution time, particularly given the scale of the studied projects and the number of environment combinations considered. Finally, the mitigation approach for environmental flakiness focuses on controlling rather than repairing flaky tests, and thus does not eliminate the underlying causes.

Despite these limitations, the consistent patterns observed across multiple studies and methods provide confidence in the validity of the findings and their relevance to real-world JavaScript development.

8.4 Future Work

This thesis opens several directions for future research. One promising direction is integrating large language models to assist with flaky test analysis. The datasets and tools developed in this work could support models that predict flaky behaviour, classify the causes of flakiness, or recommend fixes based on historical patterns.

Another direction is extending the proposed techniques to additional JavaScript testing frameworks and environments, such as Vitest or emerging runtime platforms (e.g., Deno, Bun, and edge-based runtimes), where differences in execution models and available APIs may give rise to new forms of test flakiness. Further research could also explore automated repair techniques for order-dependent and environment-dependent tests in JavaScript, building on approaches proposed in other ecosystems [55, 56].

Future work could also explore the proposed techniques in a broader range of JavaScript applications beyond Node.js-based projects. Studying how environment-dependent flakiness manifests in these settings would help assess the generality of the findings and identify ecosystem-specific differences.

More broadly, future work could investigate how environment-aware and order-aware testing strategies can be incorporated into continuous integration pipelines in a principled and automated manner, balancing test coverage, execution cost, and reliability.

8.5 Final Remarks

Flaky tests remain a significant challenge in modern software development, particularly in dynamic, heterogeneous ecosystems such as JavaScript. By combining empirical analysis with targeted tool support, this thesis demonstrates that flaky behaviour can be better

understood, detected, and controlled. The results underscore the importance of ecosystem-specific research and highlight practical paths towards more reliable, trustworthy automated testing.

References

[1] D. Martin, "11 of the most costly software errors in history," 2022, https://raygun.com/blog/costly-software-errors-history/.

[2] G. Le Lann, "The ariane 5 flight 501 failure-a case study in system engineering for computing systems," Ph.D. dissertation, INRIA, 1996.

[3] H. Dolfing, "Project failure case study: Knight capital," June 2019, accessed: 7 October 2025. [Online]. Available: https://www.henricodolfing.com/2019/06/project-failure-case-study-knight-capital.html

[4] ABC News. (2025, Sep.) Optus network outage slams WA government communications, ministers demand answers. https://www.abc.net.au/news/2025-09-20/optus-outage-wa-government-ministers-slam-telco/105797804. Accessed: 7 October 2025.

[5] Idealink Tech. (2024) Understanding software testing costs: Development breakdown. Accessed: 7 October 2025. [Online]. Available: https://idealink.tech/blog/understanding-software-testing-costs-development-breakdown

[6] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 583–594.

[7] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the ACM International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[8] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018.

[9] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018.

[10] G. Pirocanac. Test flakiness – one of the main challenges of automated testing. [Online]. Available: https://testing.googleblog.com/2020/12/test-flakiness-one-of-main-challenges.html

[11] M. Machalica, W. Chmiel, S. Swierc, and R. Sakevych, "Probabilistic flakiness: How do you test your tests? - engineering at meta," https://engineering.fb.com/2020/12/10/developer-tools/probabilistic-flakiness/, 2020, (Accessed on 12/03/2024).

[12] O. Parry, G. Kapfhammer, M. Hilton, and P. McMinn, "Systemic flakiness: An empirical analysis of co-occurring flaky test failures," *arXiv preprint arXiv:2504.16777*, 2025.

[13] F. Leinen, D. Elsner, A. Pretschner, A. Stahlbauer, M. Sailer, and E. Jürgens, "Cost of flaky tests in continuous integration: An industrial case study," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2024, pp. 329–340.

[14] A. Sandhu, "How to fix flaky tests," 2015. [Online]. Available: https://tech.justeattakeaway.com/2015/03/30/how-to-fix-flaky-tests/

[15] J. Palmer, "Test flakiness methods for identifying and dealing with flaky tests," 2019. [Online]. Available: https://engineering.atspotify.com/2019/11/18/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/

[16] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.

[17] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019.

[18] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, "Dependent-test-aware regression testing techniques," in *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2020.

[19] B. Vancsics, T. Gergely, and Á. Beszédes, "Simulating the effect of test flakiness on fault localization effectiveness," in *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, 2020, pp. 28–35.

[20] Y. Qin, S. Wang, K. Liu, X. Mao, and T. F. Bissyand, "On the impact of flaky tests in automated program repair," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 295–306.

[21] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An empirical analysis of UI-based flaky tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1585–1597.

[22] K. Costa, R. Ferreira, G. Pinto, M. d'Amorim, and B. Miranda, "Test flakiness across programming languages," *IEEE Transactions on Software Engineering*, 2022.

[23] Simply Technologies, "Why is javascript so popular?" 2018. [Online]. Available: https://web.archive.org/web/20191208013733/https://www.simplytechnologies.net/blog/2018/4/11/why-is-javascript-so-popular

[24] I. Lima, J. Silva, B. Miranda, G. Pinto, and M. dAmorim, "Exposing bugs in javascript engines through test transplantation and differential testing," *Software Quality Journal*, vol. 29, no. 1, pp. 129–158, 2021.

[25] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[26] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018.

[27] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Concurrency-related flaky test detection in android apps," 2020.

[28] A. M. Memon and M. B. Cohen, "Automated testing of gui applications: models, tools, and controlling flakiness," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1479–1480.

[29] V. Terragni, P. Salza, and F. Ferrucci, "A container-based infrastructure for fuzzy-driven root causing of flaky tests," in *ICSE-NIER '20: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, 2020, pp. 69–72.

[30] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–74, 2021.

[31] B. Zolfaghari, R. M. Parizi, G. Srivastava, and Y. Hailemariam, "Root causing, detecting, and fixing flaky tests: State of the art and future roadmap," *Software: Practice and Experience*, vol. 51, no. 5, pp. 851–867, 2021.

[32] A. T. Endo and A. Møller, "Noderacer: Event race detection for node. js applications," in *Proceedings of the IEEE International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[33] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, "Towards a bayesian network model for predicting flaky automated tests," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2018.

[34] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Hoboken, NJ: John Wiley & Sons, 2011.

[35] R. S. Pressman and B. R. Maxim, *Software Engineering: A Practitioners Approach*, 9th ed. New York, NY: McGraw-Hill Education, 2020.

[36] I. Sommerville, *Software Engineering*, 10th ed. Harlow, UK: Pearson Education Limited, 2015.

[37] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981.

[38] *IEEE Standard for System, Software, and Hardware Verification and Validation*, IEEE Computer Society Std. IEEE Std 1012-2016, 2016.

[39] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge, UK: Cambridge University Press, 2016.

[40] B. Kitchenham, "Software quality: The elusive target," *IEEE Software*, vol. 13, no. 1, pp. 12–21, 1996.

[41] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Boston, MA: Addison-Wesley, 2007.

[42] M. Fowler and M. Foemmel, "Continuous integration," https://martinfowler.com/articles/continuousIntegration.html, 2006, accessed: 2025-10-10.

[43] V. Garousi and M. V. Mntyl, "When and what to automate in software testing? a multivocal literature review," *Information and Software Technology*, vol. 76, pp. 92–117, 2016.

[44] V. Garousi, M. Felderer, and M. V. Mntyl, "The need for multivocal literature reviews in software testing," *Information and Software Technology*, vol. 123, p. 106275, 2020.

[45] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–29, 2020.

[46] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of java projects using continuous integration," in *Proceedings of the ACM Conference on Foundations of Software Engineering (FSE)*, 2017.

[47] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 399–410.

[48] A. Lassila *et al.*, "Opportunities and challenges in adopting continuous end-to-end testing: A case study," 2019.

[49] T. Hirsch, C. Schindler, M. Müller, T. Schranz, and W. Slany, "An approach to test classification in big android applications," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2019, pp. 300–308.

[50] Z. Fan, "A systematic evaluation of problematic tests generated by evosuite," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 165–167.

[51] D. R. MacIver and A. F. Donaldson, "Test-case reduction via test-case generation: Insights from the hypothesis reducer (tool insights paper)," in *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 13–1.

[52] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "ifixflakies: A framework for automatically fixing order-dependent flaky tests," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[53] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu, "A conceptual replication of continuous integration pain points in the context of travis ci," in *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 647–658.

[54] U. Zdun, E. Wittern, and P. Leitner, "Emerging trends, challenges, and experiences in devops and microservice apis," *IEEE Software*, vol. 37, no. 1, pp. 87–91, 2019.

[55] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in *Proceedings of the IEEE Conference on Ssoftware Testing, Validation and Verification (ICST)*. IEEE, 2019.

[56] R. Wang, Y. Chen, and W. Lam, "Ipflakies: A framework for detecting and fixing python order-dependent flaky tests," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022.

[57] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov, "Nondex: A tool for detecting and debugging wrong assumptions on java api specifications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 993–997.

[58] J. Chen, W. Shang, and E. Shihab, "Perfjit: Test-level just-in-time prediction for performance regression introducing commits," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1529–1544, 2020.

[59] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[60] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of flaky tests in python," in *Proceedings of the IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021.

[61] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," in *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2020.

[62] C. Li and A. Shi, "Evolution-aware detection of order-dependent flaky tests," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

[63] C. Li, C. Zhu, W. Wang, and A. Shi, "Repairing order-dependent flaky tests via test generation," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2022.

[64] A. Wei, P. Yi, T. Xie, D. Marinov, and W. Lam, "Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests," in *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part I 27*. Springer, 2021, pp. 270–287.

[65] C. Li, M. M. Khosravi, W. Lam, and A. Shi, "Systematically producing test orders to detect order-dependent flaky tests," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 627–638.

[66] M. Gruber and G. Fraser, "Flapy: mining flaky python tests at scale," in *Proceedings of the IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2023.

[67] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1471–1482.

[68] A. Berglund and O. Vateman, "Mitigation and handling of non-deterministic tests in automatic regression testing," *LU-CS-EX*, 2020.

[69] S. Rahman, A. Massey, W. Lam, A. Shi, and J. Bell, "Automatically reproducing timing-dependent flaky-test failures," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2024, pp. 269–280.

[70] J. Morán Barbón, C. Augusto Alonso, A. Bertolino, C. A. Riva Álvarez, P. J. Tuya González *et al.*, "Flakyloc: flakiness localization for reliable test suites in web applications," *Journal of Web Engineering, 2*, 2020.

[71] J. Micco, "The state of continuous integration testing at google," https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html, 2017.

[72] R. Job and A. Hora, "How and why developers implement os-specific tests," *Empirical Software Engineering*, vol. 30, no. 1, pp. 1–33, 2025.

[73] A. Berndt, T. Bach, and S. Baltes, "Do test and environmental complexity increase flakiness? an empirical study of sap hana," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2024, pp. 572–581.

[74] J. Morán Barbón, C. Augusto Alonso, A. Bertolino, C. A. Riva Álvarez, J. F. García Tuya *et al.*, "Debugging flaky tests on web applications," in *Proceedings of the 15th International Conference on Web Information Systems and Technologies-Volume 1: APMDWE*, 2019.

[75] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27.

[76] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: A manifesto," *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[77] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[78] A. Tahir, S. Rasheed, J. Dietrich, N. Hashemi, and L. Zhang, "Test flakiness causes, detection, impact and responses: A multivocal review," *Journal of Systems and Software*, vol. 206, p. 111837, 2023.

[79] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2018.

[80] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[81] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Flaky test detection in android via event order exploration," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 367–378.

[82] D. Silva, L. Teixeira, and M. d'Amorim, "Shake it! detecting flaky tests caused by concurrency with shaker," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 301–311.

[83] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Flake it'till you make it: Using automated repair to induce and fix latent test flakiness," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 11–12.

[84] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. L. Traon, "Discerning legitimate failures from false alerts: A study of chromium's continuous integration," *arXiv preprint arXiv:2111.03382*, 2021.

[85] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino, "Know you neighbor: Fast static prediction of test flakiness," *IEEE Access*, vol. 9, pp. 76 119–76 134, 2021.

[86] Y. Qin, S. Wang, K. Liu, B. Lin, H. Wu, L. Li, X. Mao, and T. F. Bissyandé, "Peeler: Learning to effectively predict flakiness without running tests," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022.

[87] R. Mudduluru, J. Waataja, S. Millstein, and M. Ernst, "Verifying determinism in sequential programs," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 37–49.

[88] P. Yi, A. Wei, W. Lam, T. Xie, and D. Marinov, "Finding polluter tests using java pathfinder," *ACM SIGSOFT Software Engineering Notes*, vol. 46, no. 3, pp. 37–41, 2021.

[89] S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify: A black-box, language model-based predictor for flaky tests," 2021. [Online]. Available: https://arxiv.org/abs/2112.12331

[90] A. Ahmad, O. Leifler, and K. Sandahl, "An evaluation of machine learning methods for predicting flaky tests," in *QuASoQ@APSEC*, 2020.

[91] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "Flakeflagger: Predicting flakiness without rerunning tests," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2021.

[92] A. Groce and J. Holmes, "Practical automatic lightweight nondeterminism and flaky test detection and debugging for python," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2020, pp. 188–195.

[93] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 492–502.

[94] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Evaluating features for machine learning detection of order-and non-order-dependent flaky tests," in *Proceedings of the IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022.

[95] ——, "Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models," *Empirical Software Engineering*, vol. 28, no. 3, p. 72, 2023.

[96] J. Micco, "Google testing blog: Flaky tests at google and how we mitigate them," https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html, (Accessed on 25/03/2024).

[97] E. Zhu, "Solving flaky tests in rspec," 2019. [Online]. Available: https://flexport.engineering/solving-flaky-tests-in-rspec-9ceadedeaf0e

[98] R. Agarwal, "Handling Flaky Unit Tests in Java," Jun. 2021. [Online]. Available: https://eng.uber.com/handling-flaky-tests-java/

[99] B. Chen and B. Lee, "Flaky tests: their hidden costs and how to address flaky behavior," Online; Datadog Engineering blog, 2024, accessed: 2025-12-16. [Online].

Available: https://www.datadoghq.com/blog/datadog-flaky-tests/#quarantine-tests-that-frequently-flake

[100] S. Dutta, A. Arunachalam, and S. Misailovic, "To seed or not to seed? an empirical analysis of usage of seeds for testing in machine learning projects," in *15th IEEE International Conference on Software Testing, Verification and Validation*, 2022.

[101] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi, "Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 50–61.

[102] G. A. Yost, "Finding flaky tests in javascript applications using stress and test suite reordering," Master's thesis, The University of Texas at Austin, 2023.

[103] S. Fatima, H. Hemmati, and L. Briand, "Flakyfix: Using large language models for predicting flaky test fix categories and test code repair," *arXiv preprint arXiv:2307.00012*, 2024.

[104] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[105] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: a roadmap," in *Proceedings of the conference on The future of Software engineering*, 2000, pp. 345–355.

[106] C. Wohlin, M. Höst, and K. Henningsson, "Empirical research methods in software engineering," in *Empirical methods and studies in software engineering*. Springer, 2003, pp. 7–23.

[107] C. Williams *et al.*, "Research methods," *Journal of Business & Economics Research (JBER)*, vol. 5, no. 3, 2007.

[108] C. Wohlin, M. Höst, and K. Henningsson, "Empirical research methods in web and software engineering," in *Web engineering*. Springer, 2006, pp. 409–430.

[109] A. Bianchi, D. Caivano, R. Conradi, L. Jaccheri, M. Torchiano, and G. Visaggio, "Cots products characterization: Proposal and empirical assessment," in *Empirical Methods and Studies in Software Engineering*. Springer, 2003, pp. 233–255.

[110] L. Zhang, J.-H. Tian, J. Jiang, Y.-J. Liu, M.-Y. Pu, and T. Yue, "Empirical research in software engineeringa literature survey," *Journal of Computer Science and Technology*, vol. 33, no. 5, pp. 876–899, 2018.

[111] R. Dawson, P. Bones, B. J. Oates, P. Brereton, M. Azuma, and M. L. Jackson, "Empirical methodologies in software engineering," in *Eleventh Annual International Workshop on Software Technology and Engineering Practice.* IEEE, 2003, pp. 52–58.

[112] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to advanced empirical software engineering.* Springer, 2008, pp. 285–311.

[113] M. Taleb, "Javascript unit testing frameworks in 2024: A comparison raygun blog," https://raygun.com/blog/javascript-unit-testing-frameworks/, 2023, (Accessed on 09/02/2024).

[114] J. Community, "Issue #4386: Flakiness in tests with jest," https://github.com/jestjs/jest/issues/4386, 2023, accessed: 2024-09-11.

[115] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 385–396.

[116] "State of javascript 2023: Testing," https://2023.stateofjs.com/en-US/libraries/testing/, (Accessed on 09/02/2024).

[117] "Globals jest," https://jestjs.io/docs/api#describename-fn, (Accessed on 10/02/2024).

[118] "Setup and teardown jest," https://jestjs.io/docs/setup-teardown#order-of-execution, (Accessed on 10/02/2024).

[119] "Jest cli options jest," https://jestjs.io/docs/cli, (Accessed on 10/02/2024).

[120] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in java projects," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE).* IEEE, 2020, pp. 403–413.

[121] Anonymous, "Detecting and evaluating order-dependent flaky tests in javascript - replication package," 2024. [Online]. Available: https://doi.org/10.5281/zenodo.13852085

[122] "timkindberg/jest-when: Jest support for mock argument-matched return values." https://github.com/timkindberg/jest-when, (Accessed on 10/01/2024).

[123] "tj/commander.js: node.js command-line interfaces made easy," https://github.com/tj/commander.js, (Accessed on 10/01/2024).

[124] "rbardini/jest-it-up: Automatically bump up global jest thresholds whenever coverage goes above them," https://github.com/rbardini/jest-it-up, (Accessed on 10/01/2024).

[125] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. L. Traon, "A qualitative study on the sources, impacts, and mitigation strategies of flaky tests," *arXiv preprint arXiv:2112.04919*, 2021.

[126] N. Malik, "Taming test flakiness: How we built a scalable tool to detect and manage flaky tests," Online; Atlassian, 2025, accessed: 2025-12-16. [Online]. Available: https://www.atlassian.com/blog/atlassian-engineering/taming-test-flakiness-how-we-built-a-scalable-tool-to-detect-and-manage-flaky-tests

[127] "Trunk," Online; Trunk, 2025, accessed: 2025-12-16. [Online]. Available: https://docs.trunk.io/flaky-tests/quarantining

[128] A. Adekotujo, A. Odumabo, A. Adedokun, and O. Aiyeniko, "A comparative study of operating systems: Case of windows, unix, linux, mac, android and ios," *International Journal of Computer Applications*, vol. 176, no. 39, pp. 16–23, 2020.

[129] "Rest api endpoints for search - github docs," https://docs.github.com/en/rest/search/search, (Accessed on 10/01/2024).

[130] S. Brisson, E. Noei, and K. Lyons, "We are family: analyzing communication in github software repositories and their forks," in *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 59–69.

[131] M. Dawson. (2017) Node.js errors changes you need to know about. (Accessed 9/6/2025). [Online]. Available: https://medium.com/the-node-js-collection/node-js-errors-changes-you-need-to-know-about-dc8c82417f65

[132] J. Dietrich, S. Rasheed, and A. Tahir, "Flaky test sanitisation via on-the-fly assumption inference for tests with network dependencies," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2022, pp. 264–275.