Awaiting Trouble: Characterizing Concurrency Issues in Modern Promise-Based JavaScript Applications

Pedro Vinícius Guandalini Vicente
Universidade Federal de São Carlos
São Carlos, São Paulo, Brazil
pedrovicente@estudante.ufscar.br

André Takeshi Endo
Universidade Federal de São Carlos
São Carlos, São Paulo, Brazil
andreendo@ufscar.br

Abstract

JavaScript is one of the most widely used programming languages, powering both client-side and server-side applications. Its modern asynchronous features with promises and async/await syntax enable cleaner and more scalable code for concurrent computations. However, JavaScript applications remain prone to concurrency issues, ranging from internal event races due to event handlers' interleavings, to nondeterminism in interactions with external systems. Existing studies have mainly focused on callback-based event races, leaving a gap in understanding how such issues manifest in modern Promise-based code. This paper presents an empirical study that analyzes, classifies, and tries to reproduce merged modifications that contain concurrency issues in modern JavaScript applications. To do so, we assessed 22 pull requests from open source Promise-based JavaScript projects, and classified them using bug patterns proposed in the literature. Furthermore, we introduce four manifestation patterns that characterize how such issues occur in modern scenarios. Our results give evidence that these concurrency issues frequently manifest as flaky tests, but are difficult to reproduce. We anticipate that the analyzed cases can assist the development of newer tools and approaches for detecting concurrency issues in modern JavaScript applications.

Keywords

Automated Tests, Concurrency, Event Race, Race Condition, Node.js

1 Introduction

JavaScript is one of the most widely used programming languages for developing both client-side and server-side applications [22, 23]. With the introduction of Node.js, a runtime built on top of Chrome's V8 engine, JavaScript expanded its reach into high-performance, scalable, event-driven backend systems used by industry leaders such as Netflix, PayPal and Amazon [4]. Node.js promotes a non-blocking, single-threaded architecture for I/O-bound and concurrent applications. JavaScript's asynchronous programming model, centered around Promises and the `async/await` syntax, has largely replaced callback-based patterns, offering cleaner and more maintainable code [10, 14].

Despite these improvements, modern asynchronous structures do not eliminate issues that arise from code that performs concurrent computations. JavaScript's event-driven model introduces a well-documented class of concurrency issues that arise from the nondeterministic interleaving of asynchronous tasks and event handlers, including atomicity violations, order violations, and starvation scenarios [25]. These issues can manifest in several forms, including application crashes, corrupted states, flaky tests (tests that pass and fail nondeterministically [15]), and hard-to-reproduce failures [5, 26]. Some concurrency issues are related to event races [1], which are race conditions generated by non-deterministic interleavings in the execution order of events. Such issues arise in Promise-based code, where developers may inadvertently trigger multiple event flows that interfere with each other due to the complex nature of Promises [2, 16].

While the detection and characterization of concurrency issues have been studied in other programming paradigms [12, 13], research on JavaScript's event-driven model has its own unique challenges. The study by Wang et al. [25] established a structural taxonomy of concurrency issues in Node.js, categorizing them into order violations, atomicity violations, and starvation. However, the concurrency issues analyzed by Wang et al. [25] were predominantly callback-based, and their taxonomy was designed to answer what is structurally wrong in the code. It does not directly address how these issues appear in testing environments to developers and testers working with modern JavaScript applications, the common anti-pattern associated with these issues or how frequently they manifest as flaky tests. Similarly, many detection tools such as NodeRacer [9] and NRace [5] primarily focus on callback-based event races, and while newer tools like NodeRT [26] have started to address Promise-related behaviors, the literature still lacks a systematic, empirical characterization of how concurrency issues manifest in code that heavily utilizes Promises and `async/await` from the perspective of developers and testing practitioners.

This paper presents an empirical study analyzing 22 real-world cases to characterize concurrency issues in modern JavaScript applications that utilize Promises and `async/await`, complementing the established taxonomy of Wang et al. [25] with a perspective on how these issues recurrently manifest in open-source testing scenarios. Rather than replacing the established view, we systematically analyze merged code modifications that involve modern asynchronous constructs, rather than legacy callbacks.

To conduct this investigation, we mined pull requests from open-source projects using the GitHub API, where each merged fix serves as a case. For each case, we analyze its associated artifacts, including code, tests and developer discussions to build a qualitative understanding of the phenomenon, mapping each case to both Wang et al. [25] patterns and our manifestation taxonomy. We then attempt to reproduce each issue through repeated test executions reruns under normal conditions and with the aid of NACD [10], a state-of-the-art race detection tool for Node.js.

Our analysis of 22 real-world cases indicates that the structural categories established by Wang et al. [25] of order violation, atomicity violation, and starvation remain applicable to Promise-based

code, while also revealing four manifestation patterns: Stabilization Races, Lifecycle Races, Concurrent Access Races, and External Nondeterminism, that describe how these concurrency issues recurrently manifest in modern testing scenarios. While these issues often manifest as flaky tests, we discovered that only 36% of them could be reliably reproduced even with a large number of repetitions, and that only 1 case (4%) had its manifestation as a flaky test uncovered exclusively by a race detector for Node.js. Based on these findings, this paper makes the following main contributions:

(1) **A Complementary Taxonomy of Concurrency Issues:** We present an empirical classification of concurrency issues manifestation patterns in modern JavaScript applications, showing how established bug categories [25] surface in the context of Promise-based code and contemporary testing practices.
(2) **A Publicly Available Benchmark Suite:** We provide an executable suite of 22 issues related to concurrency and the reproducibility steps of how these issues can manifest as flaky tests. Each benchmark has an execution log and setup instructions to facilitate the replication and development of future analyses and detection tools.
(3) **Insights for Developers and Researchers:** By providing an analysis of real-world issues, we offer insights to help developers avoid common anti-patterns and provide a foundation for researchers and practitioners to build more effective testing tools for modern JavaScript.

2 Motivating Example

In this study, a case is defined as a single, reported real-world concurrency issue, documented within a pull request (PR) on GitHub that was merged and involves modern asynchronous JavaScript features. Each case represents a distinct instance of the phenomenon we aim to investigate. In modern JavaScript, a *Promise* is an object that represents the eventual completion or failure of an asynchronous operation [8, 18]. The `async/await` syntax, built on top of Promises, enables asynchronous code to be written in a sequential style. An `async` function always returns a Promise, and the `await` keyword suspends its execution until the Promise settles [17].

To illustrate how concurrency issues present in Promise-based applications, observe the case from project *cs4218-2420-ecom-project-team15*, reported in PR #21[1]. This case illustrates a concurrency issue caused by external nondeterminism that manifests as a flaky test. The problematic code is shown in Listing 1, in which the test aims to verify the selection of categories in a database.

The race occurs due to the nondeterminism between the asynchronous operations used to get the categories from the database. While the `categoryModel.create` operation (lines 2-4) inserts data in a specific sequence, the `getAllCategoryController` operation (line 8) performs a general SELECT query without an ORDER BY clause, which does not guarantee the expected order from the database's internal query execution plan, leading to nondeterminism. Consequently, the category named "Books" can be returned first, causing the assertions in lines 9 and 10 to fail intermittently, hence making the test flaky. In our experiments, this test failed four times in 1,000 executions.

[1]https://github.com/cs4218/cs4218-2420-ecom-project-team15/pull/21

```
1  it('should return all categories', async () => {
2    await categoryModel.create([
3      { name: "Electronics", slug: "electronics" },
4      { name: "Books", slug: "books" } ]);
5    const res = {
6      status: jest.fn().MockReturnThis(),
7      send: jest.fn() };
8    await getAllCategoryController(res);
9    expect(res.category[0].name).toBe("Electronics");
10   expect(res.category[1].name).toBe("Books");
11 });
```

Listing 1: Motivating example of a flaky test caused by a concurrency issue.

While this behavior is caused by an external system (i.e., MongoDB), similar issues may arise from nondeterministic scheduling of event handlers on the JavaScript side. For instance, if data for "Books" and "Electronics" were retrieved through two asynchronous promise-based operations, their responses could arrive in an unpredictable order, leading to the same type of flaky test. The proposed fix was to make the assertions indifferent to the order of the data. By checking for the presence of the expected item in the array, rather than their position, the test becomes no longer flaky.

3 Study Setup

To guide our investigation, we address the following research questions (RQs):

RQ1 (Wang et al.'s Patterns): How do concurrency issues in modern, Promise-based applications map to the established structural patterns proposed by Wang et al. [25]?
RQ2 (Manifestation Patterns): What patterns characterize how concurrency issues recurrently manifest in modern Promise-based testing scenarios?
RQ3 (Reproducibility): How often are concurrency issues in modern JavaScript applications reproduced as flaky tests?

To answer our research questions, we designed and conducted an empirical study investigating 22 real-world cases [19]. This methodology is well-suited for investigating contemporary phenomena in their real-world context, and for understanding concurrency issues as they occur in open-source JavaScript projects. This section details our protocol, including the case selection process (Subsection 3.1) and the project validation step (Subsection 3.2).

3.1 Case Selection Process

The selection of cases followed a systematic process designed to identify a relevant and representative set of concurrency issues from a large pool of candidates. We used Python scripts to query the GitHub API and collect pull requests from two distinct sets of projects. First, we collected a large-scale dataset of 86,240 pull requests from JavaScript projects on GitHub, covering PRs created between January 1, 2019 and May 1, 2025. The starting date was chosen because `async/await` was introduced in JavaScript in 2017 [7], and, according to Lucas et al. [14], the widespread adoption of modern JavaScript features typically occurs one to two years after their release. The end date corresponds to the day the GitHub API searches were conducted. Then, to ensure the inclusion of well-established projects, we collected an additional 1,745 pull requests

from 566 projects featured in curated Node.js lists [20, 21], resulting in an unfiltered dataset containing 87,985 PRs.

The initial set of 87,985 PRs was too large for manual inspection. Therefore, we developed an automated filtering script to identify PRs likely to describe relevant concurrency issues. A PR was considered a candidate if it satisfied three criteria simultaneously: *(i)* containing concurrency issues related keywords (namely, "concurrency bug", "race condition", "event race", "flaky test", "race bug"); *(ii)* showing modifications to test files containing terms "`describe`", "`it`" or "`test`", as we require the existence of a test that can be used to try to reproduce the reported issue; and *(iii)* involving modern asynchronous syntax ("`promise`", "`async`", "`await`"). This automated filtering process reduced the 87,985 PRs down to a candidate pool of 1,733 PRs; this list is available in our repository.

3.2 Project Validation and Analysis

Given the considerable effort needed to validate each potential case, we employed a random sampling strategy. We randomly drew PRs one-by-one from the candidate pool and subjected them to a validation protocol, shown in Figure 1. The protocol begins with the *Pre-Fix Setup*, where we had at most 30 minutes to configure and build the project version containing the bug (pre-fix version). If the setup failed, we moved to the next PR. Upon successful setup, we proceeded to the *Baseline Test Confirmation* by executing the relevant test ten times under normal conditions. Since concurrency issues are inherently nondeterministic, a valid case must exhibit at least 1 passing run, confirming that the test is flaky rather than consistently broken. A case in which all ten runs failed was therefore discarded. If the test passed at least once, we moved to the *Stress Testing & Reproduction*. The goal of this stage was to systematically attempt to reproduce the reported issue. To this end, we first executed the test 1,000 times under normal conditions to establish a baseline failure rate. Then, we executed it 100 times using NACD [10], a state-of-the-art race detection tool for Node.js. NACD operates by instrumenting asynchronous operations and dynamically injecting random delays, which intentionally disturbs the event handler's schedule to expose rare interleavings and race conditions. Because this dynamic delay injection introduces a runtime overhead, we limited the testing with NACD to 100 runs per case.

If a test failed at least once during these 1,100 runs, we included the bug in *Inclusion: Manifestation Observed*. However, our study's inclusion criteria did not rely only on our ability to reproduce the issue. Since every case was derived from a PR where developers had already confirmed and merged, we treat the developer's report as truthful, and included it in *Inclusion: Manifestation Not Reproduced*.

Due to the labor-intensive nature of analyzing PRs, building, reproducing and performing qualitative analysis, our evaluation was limited to 61 candidate PRs, of which 39 PRs were discarded during the *Pre-Fix Setup*, leaving 22 PRs that successfully passed the setup and baseline criteria and were therefore analyzed as cases. For each case, the foundational unit of analysis consisted of all artifacts surrounding the issue, including: the PR title, body description and discussions, the relevant source code (both pre and post-fix), the implemented fix strategy, and the broader context, such as the bug's symptoms (e.g., a crash or flaky test).

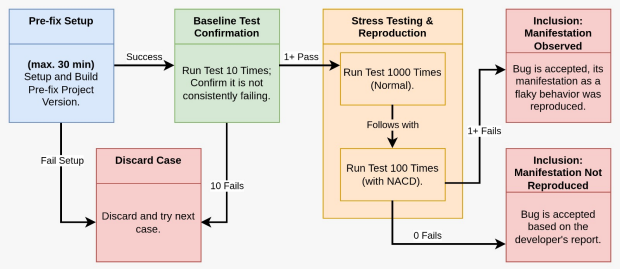

Figure 1: Protocol for analyzing and validating pull requests.

This structured information formed the basis for our analytical process. First, in an initial coding phase, we identified the cause of each issue by understanding the competing asynchronous operations and systems involved, and mapped each case to the structural categories established by Wang et al. [25] (Order Violation – OV, Atomicity Violation – AV, and Starvation – St) to verify their continued applicability in the Promise-based context. In a separate step, we analyzed each case through the lens of how the issue manifested in testing scenarios, iteratively grouping cases based on recurring similarities until our four manifestation patterns emerged from the data. Finally, to ensure the reliability of this taxonomy, the classification of all 22 cases at both levels was validated by two authors until consensus was reached. The classifications assigned by each author, together with their respective rationales, are documented in the project repository.

4 Analysis of Results

Following the protocol detailed in Section 3.1, we analyze 22 cases through three complementary lenses. To answer RQ1, in Subsections 4.1.1, 4.1.2 and 4.1.3, we map each case to the structural categories established by Wang et al. [25]. To answer RQ2, in Subsections 4.2.1, 4.2.2, 4.2.3 and 4.2.4, we present our complementary manifestation taxonomy, characterizing how these issues recurrently manifest in modern Promise-based testing scenarios. To answer RQ3, in Subsection 4.3, we analyze how often these issues manifest as flaky tests under repeated executions and with the aid of a state-of-the-art race detection tool. Finally, Subsection 4.4 addresses the threats to the validity of our findings.

Table 1 shows the 22 analyzed cases, listing each PR and project name, followed by their classification according to two patterns. The first is Wang et al.'s structural pattern (OV: Order Violation, AV: Atomicity Violation, St: Starvation). The second is our manifestation pattern (SR: Stabilization Races, LR: Lifecycle Races, CA: Concurrent Access Races, EN: External Nondeterminism). Finally, the table reports the number of failures observed in 1,000 test runs under normal conditions, and in 100 runs using NACD. A summary of the distribution of cases across both classifications is presented in Table 2.

4.1 RQ1: Wang et al.'s Patterns

Table 2 summarizes the distribution of cases across the patterns proposed by Wang et al. [25]. Next, we discuss each pattern, illustrating it with a representative case from our study.

Table 1: Concurrency issues, their classifications, and test results.

PR[a]	Project Name	Wang et al.'s Patterns	Manif. Patterns	#Failures	#Failures w. NACD[b]
PR #21	cs4218-2420-ecom-project-team15	OV	EN	4	0
PR #743	pwa-studio	OV	SR	0	0
PR #1	milo	OV	SR	252	27
PR #58629	gutenberg	OV	SR	11	1
PR #2	express-promise-middleware	OV	SR	0	0
PR #100	eq-author-app	OV	CA	0	0
PR #1120	core	OV	SR	0	*
PR #8513	pouchdb	St	LR	0	0
PR #117	holo-schedule	AV	CA	0	0
PR #2940	cozy-drive	OV	LR	2	0
PR #17	bare-http1	OV	LR	933	91
PR #396	fetch	OV	LR	0	0
PR #1186	pino	OV	LR	0	80
PR #26331	prisma	OV	EN	0	*
PR #4260	flowfuse	OV	SR	0	*
PR #1387	lion	OV	LR	0	0
PR #32	hls-video-element	OV	LR	4	*
PR #7005	rxjs	AV	LR	0	0
PR #329	tldr-node-client	OV	EN	2	*
PR #114	ember-css-transitions	OV	SR	0	*
PR #156	invest-workbench	OV	LR	0	0
PR #583	timed-frontend	OV	SR	382	*

[a] This column contains clickable links to the respective pull requests.
[b] Projects marked with symbol ∗ could not be executed using the NACD tool due to Node version compatibility or other issues.

Table 2: Analyzed cases across both classification.

Visualization	Pattern	Cases	%
Wang et al.'s Patterns	Order Violation	19/22	86%
	Atomicity Violation	2/22	9%
	Starvation	1/22	4%
Manifestation Patterns	Stabilization Race	8/22	36%
	Lifecycle Race	9/22	40%
	Concurrent Access Race	2/22	9%
	External Nondeterminism	3/22	13%

4.1.1 Order Violation (OV). Order Violation issues occur when there is a violation to the developer's order intention between two or more asynchronous operations or event handlers in executions that are not enforced by the runtime [25]. One operation assumes that another has already completed, or that the system remains in an expected state, but JavaScript engines' nondeterministic event scheduling provides no such guarantee.

In our dataset, Order Violation is the dominant pattern, accounting for 19 of the 22 analyzed cases (86%). This prevalence reflects a challenge of reasoning about sequencing in Promise-based code: while `async/await` syntax simplifies asynchronous control flow compared to nested callbacks, Order Violations still occur when developers omit an `await` or leave a Promise without proper synchronization. In these instances, the unawaited operation continues running concurrently with subsequent test logic or teardown steps, leading to unmet ordering assumptions by the developers. Order Violation cases in our sample manifest mostly as Stabilization Races (8 cases) and Lifecycle Races (7 cases), where the violated ordering assumption affects the stabilization of an asynchronous side effect before it is observed, or the sequencing of setup and teardown operations in a component's lifecycle.

A representative example of this pattern is found in *flowfuse* (PR #4260), represented in Listing 2. The test creates and starts two instances concurrently (lines 13-17), then makes an API call (simplified in line 18), and asserts the state of the instances (lines 20 and 22). The idea of the test is to assert the correct state properties of an instance that is properly awaited and is in the 'running' state (`secondInstance`), and another instance that is still in its 'starting' state (`thirdInstance`). Therefore, while the startup of `secondInstance` is correctly awaited through its `started` promise (line 16), the `started` promise returned by `thirdInstance`'s startup (line 17) is not. The test then proceeds to the API call (simplified in line 18) and assertions (lines 20 and 22), while `thirdInstance`'s startup is still asynchronously in progress. However, the issue is that in the intended order, the startup should fully resolve before the test completes. But, without proper awaiting, the pending asynchronous operation from `thirdInstance` races against the test teardown, producing nondeterministic behavior in the database and subsequent tests.

```
12  it('Get list of teams instances and stats', async () => {
13    const secondInstance = await app.createInstance();
14    const thirdInstance = await app.createInstance();
15    const startSecond = await app.start(secondInstance);
16    await startSecond.started;
17    const startThird = await app.start(thirdInstance);
18    const res = await app.inject({method:'GET',url:`..`});
19    const secondInstStat = app.find(secondInstance.id);
20    secondInstStat.should.have.property('state','running');
21    const thirdInstStat = app.find(thirdInstance.id);
22    thirdInstStat.should.have.property('state','starting');
23  });
```

Listing 2: Order Violation example from *flowfuse* (PR #4260).

The fix introduced by the PR includes the `await startThird.started` at the end of the test after the assertions. In *flowfuse*, the violation is in the test infrastructure, where

an unawaited `started` promise allows the test to complete, while an asynchronous startup operation is still in progress. This case and others in following subsections illustrate that violations in Promise-based code from test environments can arise both within application logic and within the test setup itself.

4.1.2 Atomicity Violation (AV). Atomicity Violation issues occur when a sequence of operations that the developer assumes will execute as an uninterruptible (atomic) unit is instead interleaved by other asynchronous events [25]. In JavaScript, since there is no native lock mechanism [6, 25], atomicity across multiple event handlers cannot be easily enforced, making this class of issues prevalent in event-driven architectures [6, 13].

In our dataset, Atomicity Violation accounts for 2 of the 22 analyzed cases (9%). While less frequent than Order Violation, these issues tend to be subtler, as the individual operations involved are each correctly ordered in isolation, the violation only emerges when concurrent event flows interleave across what the developer intended to be an atomic region.

A representative example is found in *holo-schedule* (PR #117), shown in Listing 3. Three asynchronous operations are launched concurrently via `Promise.all` (line 25), each performing a read-modify-write sequence on the same `localStorage` object: it reads the current storage state with `getStorage('local')` (lines 26-28), merges a new object into the result, and writes it back with `storage.set()`. The developer's intent is that each of these sequences execute atomically, each read observes the result of all preceding writes. However, since no lock or transaction mechanism is in place to enforce this assumption, all three operations may read the same initial storage state before any write has completed. Each operation then writes back only its own merge, causing the last write to overwrite the previous ones, leaving the storage in an inconsistent state and causing the assertion on line 31 to fail intermittently.

```
24  it('should update storage in order', async () => {
25    await Promise.all([
26      storage.set(await getStorage('local'), ...obj1),
27      storage.set(await getStorage('local'), ...obj2),
28      storage.set(await getStorage('local'), ...obj3) ]);
29    const finalStorage = await getStorage('local');
30    const expectedStorage={initialStorage,obj1,obj2,obj3};
31    expect(finalStorage).toEqual(expectedStorage);
32  });
```

Listing 3: Atomicity Violation example from *holo-schedule* (PR #117).

The proposed solution in the PR introduces a custom lock in a task queue to serialize the read-modify-write sequence, ensuring that each operation reads the storage state only after the previous write has completed. This enforces the atomicity that the developer originally assumed.

4.1.3 Starvation (St). Starvation issues occur when a task or event handler is indefinitely prevented from executing because other operations monopolize the runtime or deprive it of the resources it needs to proceed [25]. Wang et al. observed that starvation involves high-priority callbacks blocking lower-priority ones. We note that it may also occur when the preconditions for an event handler to execute are destroyed by a concurrent operation, causing the waiting task to hang indefinitely.

In our dataset, Starvation accounts for only 1 of the 22 analyzed cases (4%), being found in *pouchdb* (PR #8513). The test runs two database operations concurrently: `func1` creates, updates, and closes a database named `test-db` (lines 34-38), while `func2` creates and destroys a database named `test-db-2` (lines 39-42). Each function would succeed in isolation, but when run in parallel, the closing of `test-db` by `func1` (line 38) may inadvertently remove the event listeners belonging to `test-db-2`. This happens because the cleanup routine in `.close` filters keys using a prefix match that is insufficiently specific: the key for `test-db-2` also satisfies this filter since it contains the same prefix. With its listeners removed, `pouch2.destroy()` (line 42) waits indefinitely for events that will never arrive, starving `func2` of the signals it needs to complete and causing the test to timeout, as shown in Listing 4.

```
33  it('Race condition in in-memory-adapter', async () => {
34    const func1 = async () => {
35      const pouch1 = new PouchDB('test-db');
36      await pouch1.bulkDocs(docs:[{id:'doc1', value:1}]);
37      await pouch1.bulkGet(docs:[{id:'doc1'}]);
38      pouch1.close(); }; // triggers cleanup:
39    const func2 = async () => {
40      const pouch2 = new PouchDB('test-db-2');
41      await pouch2.createIndex(index:{fields:['foo']});
42      pouch2.destroy(); }; // hangs:
43    await Promise.all([func1(), func2()]); // hangs
44  });
```

Listing 4: Starvation example from *pouchdb* (PR #8513).

The fix accepted in the PR narrows the prefix filter in `_close`, ensuring that only listeners belonging to the database being closed are removed. This prevents the cleanup of one instance from depriving other instances of the event signals on which they depend to proceed.

Answer to RQ1: Our findings suggest that existing patterns are applicable to concurrency issues present in modern JavaScript code. In particular, Order Violation is the dominant pattern, accounting for 19 of the 22 analyzed cases (86%). Atomicity Violation accounts for 2 cases (9%), and Starvation appears in a single case (4%).

4.2 RQ2: Manifestation Patterns

Table 2 summarizes the distribution of cases across the manifestation patterns elicited in our analyses (see Section 3). Next, we discuss each pattern, illustrating it with a representative case from our study.

While the patterns discussed previously describe the structure of each concurrency issue, they do not fully capture how these issues are exposed to developers and testers in practice. The following Subsections present our complementary manifestation taxonomy, describing four recurring patterns that emerged from our analysis of the same 22 cases. As summarized in Table 2, three of these patterns: Stabilization Races, Lifecycle Races, and Concurrent Access Races,

are forms of event races, arising from incorrect assumptions about the ordering of JavaScript's internal asynchronous events and event handlers [1, 6, 9]. The fourth pattern, External Nondeterminism, contains cases where the source of nondeterminism lies outside the JavaScript runtime. While these cases do not qualify as event races, they remain instances of concurrency issues, as the failure still originates from concurrent computations.

4.2.1 Pattern 1: Stabilization Races. The Stabilization Races pattern, observed in 8 out of 22 studied cases (36%), is a form of event race that occurs when code initiates an asynchronous action but immediately uses or asserts on the expected outcome, failing to wait for the asynchronous effects of the action to fully complete and stabilize. This pattern often manifests as flaky tests in modern applications, particularly those with graphical user interfaces (GUIs), where the test runner races against a component's asynchronous render cycle. A representative example is found in PR #58629 from *gutenberg*. As shown in Listing 5, the test navigates a tab component by simulating a key press via `press.Tab()` (line 51), and then immediately asserts that the correct tab is selected (line 52). Although the key press is awaited, the assertion may execute before React has finished processing the asynchronous effects triggered by the navigation event, causing the test to intermittently fail.

```
45  it('Handle arrow key navigation properly', async () => {
46    const { rerender } = render(
47      <ControlledTabs
48        tabs = { TABS }
49        selectedTabId = "beta"
50        selectOnMove = { selectOnMove } /> );
51    await press.Tab();
52    expect(getSelectedTab()).toHaveTextContent('Beta');
53  });
```

Listing 5: Stabilization Races example from *gutenberg* (PR #58629).

The fix merged in the code introduces a call to `waitFor`, which repeatedly evaluates the assertion until the expected condition is met or a timeout occurs. This ensures that the test only proceeds once the component has stabilized and fully completed its asynchronous rendering cycle. This GUI stabilization issue appears in other forms across our dataset. In *pwa-studio* (PR #743), tests fail by asserting on a component's state before asynchronous data fetching and rendering had finished, in *milo* (PR #1), the Web Vitals' Largest Contentful Paint measurements may be captured before the GUI has fully rendered and stabilized. This race pattern also extends beyond GUI components. In *core* (PR #1120), a test raced against the propagation of a system timezone change before running a calculation that depended on it. In all these cases, the core flaw is an implicit assumption that an asynchronous action will be fully stabilized before the next operation proceeds. The fixes proposed for these cases usually involves adding an arbitrary delay to wait for the tested component to be fully rendered before the assertion.

4.2.2 Pattern 2: Lifecycle Races. The Lifecycle Races pattern, present in 9 of our 22 studied cases (40%), is a conflict between the setup or operational logic of a process and the teardown/cleanup logic of another process or itself. This pattern is a form of event race that arises when a teardown action (e.g., a reset, cancellation, or resource cleanup) is invoked before a related setup action (e.g., initialization, data fetching, or a background task) has fully completed, leading to state corruption, resource leaks, or deadlocks. An example of this pattern is found in *lion* (PR #1387). As shown in Listing 6, the test initiates an asynchronous namespace loading operation via `manager.loadNamespace()` (lines 56-59). However, there is a possibility that before this operation completes, `manager.reset()` is called (line 60), which is intended to clear the application state. However, the initial loading operation is still ongoing. When it finally resolves, the resolution handler's logic inside the `manager`'s class executes. However, this handler's operation is now outdated, since it writes to storage that was expected to be empty after the reset, causing the test to fail.

```
54  it('empties storage after reset()', async () => {
55    manager = new LocalizeManager();
56    manager.loadNamespace({
57      new Promise(resolve => {
58        deferredResolve = { greeting: 'Hello!' };
59      }), });
60    manager.reset();
61    expect(getMembers(manager).storage).to.be.empty;
62  });
```

Listing 6: Lifecycle Races example from *lion* (PR #1387).

The fix implemented in the PR was to modify the internal logic of the `LocalizeManager` to introduce a state validation check within its promise resolution handler. This check ensures that the handler only proceeds to write to storage if its originating asynchronous operation is still considered active. Since the `reset()` function effectively cancels all active operations by clearing the object's state, the handler from the now-outdated fetch fails this validation and aborts its execution, thus preventing the lifecycle race.

This conflict between concurrent lifecycle phases manifests in various forms across our dataset. In *rxjs* (PR #7005), a rapid sequence of subscriptions and unsubscriptions to start/stop listening for data from an observer caused an operator's internal reset logic to execute before its initialization logic had finished, corrupting its internal state. We also observed this pattern interacting with shared state and stream lifecycles. In *pouchdb* (PR #8513), the teardown logic of one database instance incorrectly removed event listeners belonging to another instance due to shared state. Finally, in *bare-http1* (PR #17), a server violated the HTTP stream lifecycle by prematurely ending its response (teardown) before the client had finished streaming its request body (setup). Cases *express-promise-middleware* (PR #2) and *hls-video-element* (PR #32) also fit this pattern, both involving asynchronous stream or connection lifecycles where teardown outpaced setup.

4.2.3 Pattern 3: Concurrent Access Races. The Concurrent Access Races pattern occurs in 2 of our 22 studied cases (9%). This form of event race occurs when multiple asynchronous operations attempt to modify the same shared resource concurrently. This often leads to a "lost update" scenario, where the last operation to complete overwrites the results of others, leaving the shared resource in an inconsistent state. A representative example is found in *eq-author-app* (PR #100). When a user blurs a field, such as an alias

or title, two `onUpdate` mutations are fired in rapid succession on the same section entity. Each mutation reads the current entity state, applies its own change, and sends the update to the backend independently. Since both mutations are dispatched concurrently, the one that completes last may overwrite the other, discarding one of the field updates and leaving the entity in an inconsistent state. As shown in Listing 7, the test captures this scenario by simulating two changes and multiple update triggers in quick succession (lines 64-67), asserting both changes should be present in the merged update entity.

```
63  it("Run update triggered in quick succession", () => {
64    wrapper.simulate("change", {name: "alias", value: "updated
        alias"});
65    wrapper.simulate("update");
66    wrapper.simulate("change", {name: "title", value: "updated
        title"});
67    wrapper.simulate("update");
68    expect(handleUpdate).toHaveBeenCalledWith({
69      title: "updated title", alias: "updated alias" });
70  });
```

Listing 7: Concurrent Access Races example from *eq-author* (PR #100).

The fix introduced in the PR debounces the `handleUpdate` function by collecting all field changes in a 20ms window into a single mutation that carries all the merged entity state. This prevents concurrent updates from racing against each other and ensures the backend receives consistent mutations. Unlike the fix in the Atomicity Violation example in 4.1.2 from *holo-schedule* (PR #117), which includes locks to prevent race conditions at the storage, this fix resolves the race by debouncing concurrent mutations before they are dispatched.

4.2.4 Pattern 4: External Nondeterminism. Unlike the three preceding patterns, External Nondeterminism encompasses cases where the source of nondeterminism lies outside the JavaScript runtime itself. Rather than arising from incorrect assumptions about the ordering of internal asynchronous events or shared resource access, these issues occur because the test incorrectly assumes that an external system's behavior is stable and repeatable. In our dataset, this pattern accounts for 3 of the 22 analyzed cases (13%).

The motivating example presented in Section 2 from *cs4218-2420-ecom-project-team15* (PR #21) is an instance of this pattern. The test flakiness is caused by nondeterminism in concurrent computations within MongoDB's internal query planner, which does not guarantee a retrieval order without an explicit `ORDER BY` clause. The test incorrectly assumes that the database will always return categories in insertion order, an assumption that the external system does not uphold. This same form of external nondeterminism appears in the remaining cases of this pattern. In *prisma* (PR #26331) and *tldr-node-client* (PR #329), the nondeterministic behavior originates from external system resource availability in cache and database issues and test environment conditions rather than from the ordering of JavaScript event handlers.

Answer to RQ2: Our analysis revealed four recurring patterns that characterize how concurrency issues manifest in modern Promise-based testing scenarios. Lifecycle Races is the most frequent pattern with 9 cases (40%), Stabilization Races appears in 8 cases (36%), External Nondeterminism accounts for 3 cases (13%), and Concurrent Access Races appears in 2 cases (9%).

4.3 RQ3: Reproducibility

To answer RQ3, we investigate the data regarding the automated test failures associated with each case, summarized in Table 1. The results in columns "#Failures" and "#Failures w. NACD" reveal two characteristics of concurrency issues in modern JavaScript applications.

First, flaky tests in JavaScript applications can be associated with concurrency issues [6, 9, 10]. In our study, 6 of the 9 cases (66%) that were reported as "flaky test" are caused by an event race (i.e., *pwa-studio* (PR #743), *milo* (PR #1), *gutenberg* (PR #58629), *core* (PR #1120), *cozy-drive* (PR #2940) and *pino* (PR #1186)), while the remaining cases involve other sources of nondeterminism, such as external system behavior and test infrastructure issues. This suggests that flakiness alone is not a conclusive indicator of event races specifically, but is a broader symptom of concurrency issues.

Second, the manifestation of flaky tests caused by concurrency issues is not easily repeatable. We were able to reproduce a flaky test in only 8 of the 22 studied cases (36%) with 1,000 ordinary reruns, and in only 1 more case with reruns assisted by NACD. It is also notable that in 4 of those cases the failure rate (number of failures divided by the number of reruns) was less than 2% (*cs4218-2420-ecom-project-team15* (PR #21), *gutenberg* (PR #58629), *cozy-drive* (PR #2940) and *hls-video-element* (PR #32)). The NACD tool, compatible with 68% of the cases, only contributed to the discovery of flaky tests that were not previously uncovered by the ordinary reruns in 1 case (namely, *pino* in PR #1186), providing some evidence about the limitations of existing race detectors when applied to modern promise-based JavaScript applications.

Answer to RQ3: Concurrency issues are difficult to reproduce using associated tests. Only 8 of the 22 cases (36%) exhibited test flakiness in 1,000 runs, and in 4 of those cases the failure rate was below 2%. The use of NACD, a state-of-the-art race detection tool, contributed to uncovering a flaky manifestation that was not observed in normal reruns in only 1 case.

4.4 Threats to Validity

Similarly to other empirical studies, our work is subjected to certain threats to validity. The primary threat to external validity is that our findings' representativeness is limited by our case selection. Although our cases were drawn from a large pool of open-source projects, the final dataset consists of 22 manually validated instances. As a result, the patterns identified in this study should not be interpreted as representative of all JavaScript applications, particularly those developed in closed-source contexts. Additionally, the keyword-based filtering strategy used to identify candidate PRs may have excluded relevant cases that do not explicitly reference

concurrency-related terminology. Given the small number of analyzed cases, percentages reported throughout this paper should be interpreted as descriptive of our sample rather than statistically representative estimates.

A threat to construct validity concerns the labeling of cases in developer-reported PRs, which may be imprecise. As observed in our study, some cases described by developers as concurrency issues were not clearly attributable to event races in the strict sense, but rather to nondeterminism originating from external systems. To mitigate this threat, we did not rely solely on developer labels to classify cases, instead, each case was independently analyzed through its associated code, tests, and discussions. This process led us to introduce the External Nondeterminism pattern as a distinct category in our taxonomy, separating cases where the source of nondeterminism lies outside the JavaScript runtime from those caused by internal event races.

The internal validity of our results is influenced by the qualitative nature of the analysis, as the identification of manifestation patterns and their classification require interpretation of code, tests and developer discussions, which may be subject to researcher bias. To mitigate this risk, both classifications were validated among the authors until consensus was reached, and the rationale for each classification is documented in the project repository. Finally, since only a subset of cases exhibited observable failures under repeated execution, we relied in part on developer affirmations from merged PRs, which may introduce false positives. This limitation is consistent with the exploratory nature of our study and motivates future work with larger datasets.

5 Related Work

Research on concurrency issues in JavaScript has evolved alongside the language. This section presents three research directions: empirical characterization of concurrency issues, automated tools for detecting event races, and testing challenges specific to asynchronous JavaScript.

Regarding empirical characterization, Lu et al. [13] provided an empirical analysis of issues in multithreaded C/C++ systems and created a taxonomy of atomicity and order violations. Wang et al. [25] applied a similar methodology to Node.js applications, indicating that this taxonomy is a valid reference for event-driven environments. However, their study was conducted on a benchmark where the dominant asynchronous structure was callback-based. Consequently, their findings may not capture how concurrency issues arise when modern Promises and `async/await` are used in testing environments. In the 57 projects studied by Wang et al. [25], Atomicity Violation was the dominant pattern (65%), followed by Order Violation (30%) and Starvation (5%). Our results show a different distribution, with Order Violation accounting for 86% of cases. Another early work by Davis et al. [6] also studied Node.js event races to assist the design of a fuzzing tool. Our work extends this line of research by building on the established structural classifications and contributing a complementary manifestation taxonomy grounded in modern, Promise-based testing scenarios.

Regarding automated detection tools, NodeRacer [9] introduced a technique based on happens-before (HB) relations to guide the selective postponement of events, thereby exploring alternative execution schedules to reveal races. NRace [5] refined the proposed HB rules by creating more precise HB graphs that account for the multi-priority event queues in Node.js, enabling predictive race detection. More recently, NodeRT [26] aimed to make detection more practical by simplifying HB rules and introducing a more efficient, tree-based partial HB graph. With a different approach, NACD [10] works by injecting random delays into core asynchronous operations in the Node.js runtime, fuzzing the schedule to expose race conditions without requiring happens-before modeling. While these tools represent advancements in the detection of event races, their focus is on identifying the presence of a race by flagging a potential issue or triggering a test failure, rather than providing a characterization of patterns through which concurrency issues manifest in modern applications. The manifestation patterns and benchmark suite provided by our study can also offer insights to guide the evolution of such tools toward more effective detection of modern Promise-based concurrency issues.

Regarding testing challenges in asynchronous JavaScript, Alimadadi et al. [2] introduce a dynamic analysis approach to find "broken promises": a set of anti-patterns that can lead to bugs. Turcotte et al. [24] developed DrAsync, a tool combining static analysis with dynamic visualization of promise lifetimes to detect eight asynchronous anti-patterns in Promise-based code focusing on code quality and performance bottlenecks. Ganji et al. [11] propose new test adequacy criteria for Promise-based code to measure whether tests execute the completion and error handling of asynchronous operations, implemented in the JScope VSCode plugin, which complements traditional coverage metrics. For traditional callback-based code, Nessie [3] is a feedback-directed test generation tool that targets APIs using asynchronous callbacks. While Nessie focuses on the callback model rather than Promises, its ability to nest API calls within callbacks addresses a common style of asynchronous code. These studies imply that modern asynchronous mechanisms in JavaScript require specialized analysis and testing techniques, but they do not provide a characterization of how concurrency issues manifest in real-world testing scenarios.

6 Concluding Remarks

This paper presents an empirical study that characterizes concurrency issues and analyze how they manifest in JavaScript applications. Based on 22 merged pull requests from open-source projects, we analyzed these issues through a structural classification based on the established taxonomy of Wang et al. [25], and a manifestation taxonomy that characterizes how these issues manifest in modern, promise-based testing contexts. We also attempt to reproduce each issue as a flaky test through repeated executions and with NACD, a state-of-the-art race detection tool for Node.js.

Our findings suggest that the patterns established by Wang et al. [25] remain applicable to concurrency issues in applications using Promises and `async/await`, with Order Violation accounting for the majority of the analyzed cases. At the same time, our analysis indicates that these issues can manifest in more specific and recurring forms in modern JavaScript testing scenarios, such as Stabilization Races, Lifecycle Races, Concurrent Access Races, and External Nondeterminism. Regarding reproducibility, only 8 of the 22 cases (36%) could be triggered as flaky tests through repeated executions, and

NACD contributed to uncovering a previously unobserved failure in only 1 case. The limited size of our sample prevents strong generalizations, but the recurrence of the identified patterns in open-source projects is encouraging for future, larger-scale studies.

In the future, we plan to expand our benchmark suite with more cases, refine the classification, and investigate fix strategies in greater depth. Another direction is to use the insights from this study to assist the design of a dynamic analysis tool capable of detecting and debugging concurrency issues associated with the *Promise* and *async/await* structures, helping developers build more reliable JavaScript applications.

Artifact Availability

To support the reproducibility of our study and facilitate future research, all the artifacts developed during this process, including the data collection scripts, the benchmark suite with their respective execution logs and the setup instructions are presented, under the MIT license, in our Zenodo repository: https://doi.org/10.5281/zenodo.21682992.

Acknowledgements

P.V.G. Vicente is financially supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and São Paulo Research Foundation - FAPESP (grant #2025/00910-4). This work is supported by FAPESP (grant #2023/00577-8) and by the National Council for Scientific and Technological Development – CNPq (grants nr. 444311/2024-6 and 406941/2025-4).

References

[1] Christoffer Quist Adamsen, Anders Møller, and Frank Tip. 2017. Practical initialization race detection for JavaScript web applications. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 66:1–66:22. doi:10.1145/3133890

[2] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. 2018. Finding broken promises in asynchronous JavaScript programs. In *Proceedings of the ACM on Programming Languages*, Vol. 2. ACM, Boston, MA, USA, 1–26. doi:10.1145/3276532

[3] Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. 2022. Nessie: Automatically testing JavaScript apis with asynchronous callbacks. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh, PA, USA, 1494–1505. doi:10.1145/3510003.3510106

[4] Branko Krstic. 2023. Node JS Stats that Prove Its Awesomeness in 2023. Available at: https://webtribunal.net/blog/node-js-stats/.

[5] Xiaoning Chang, Wensheng Dou, Jun Wei, Tao Huang, Jinhui Xie, Yuetang Deng, Jianbo Yang, and Jiaheng Yang. 2021. Race detection for event-driven node. js applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Melbourne, Australia, 480–491. doi:10.1109/ASE51524.2021.9678814

[6] James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node. fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, Belgrade, Serbia, 145–160. doi:10.1145/3064176.3064188

[7] Ecma International. 2017. ECMAScript 2017 Language Specification. https://262.ecma-international.org/8.0/index.html#sec-async-function-definitions.

[8] Ecma International. 2025. ECMAScript 2025 Language Specification. https://262.ecma-international.org/#sec-promise-objects.

[9] André Takeshi Endo and Anders Møller. 2020. Noderacer: Event race detection for node. js applications. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Porto, Portugal, 120–130. doi:10.1109/ICST46399.2020.00022

[10] Andre Takeshi Endo and Anders Møller. 2025. Event Race Detection for Node.js Using Delay Injections. In *39th European Conference on Object-Oriented Programming (ECOOP 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:28. doi:10.4230/LIPIcs.ECOOP.2025.9

[11] Mohammad Ganji, Saba Alimadadi, and Frank Tip. 2023. Code coverage criteria for asynchronous programs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, San Francisco, CA, USA, 1307–1319. doi:10.1145/3611643.3616292

[12] Thananon Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys 2016)*. ACM, Bordeaux, France, 1–16. doi:10.1145/2872362.2872374

[13] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)*. ACM, Seattle, WA, USA, 329–339. doi:10.1145/1346281.1346323

[14] Walter Lucas, Rafael Nunes, Rodrigo Bonifácio, Fausto Carvalho, Ricardo Lima, Michael Silva, Adriano Torres, Paola Accioly, Eduardo Monteiro, and João Saraiva. 2025. Understanding the adoption of modern Javascript features: An empirical study on open-source systems. *Empirical Software Engineering* 30, 4 (2025), 107. doi:10.1007/s10664-025-10663-9

[15] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 643–653. doi:10.1145/2635868.2635920

[16] Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A model for reasoning about JavaScript promises. In *Proceedings of the ACM on Programming Languages*, Vol. 1. ACM, Vancouver, Canada, 1–24. doi:10.1145/3133910

[17] Mozilla Developer Network. 2025. Async Function. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function.

[18] Mozilla Developer Network. 2025. Promise. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

[19] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (2009), 131–164. doi:10.1007/s10664-008-9102-8

[20] Sindresorhus. 2019. Curating the best Node.js modules and resources. https://github.com/sindresorhus/awesome-nodejs?tab=readme-ov-file#weird.

[21] Sqreen. 2019. Curated list of awesome open-source applications made with Node.js. https://github.com/sqreen/awesome-nodejs-projects.

[22] Stack Exchange, Inc. 2025. 2025 Developer Survey. Available at: https://survey.stackoverflow.co/2025/technology.

[23] TIOBE Software BV. 2026. TIOBE Index for December 2026. Available at: https://www.tiobe.com/tiobe-index/.

[24] Alexi Turcotte, Michael D. Shah, Mark W. Aldrich, and Frank Tip. 2022. DrAsync: Identifying and Visualizing Anti-Patterns in Asynchronous JavaScript. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 774–785. doi:10.1145/3510003.3510097

[25] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A comprehensive study on real world concurrency bugs in Node. js. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Urbana Champaign, IL, USA, 520–531. doi:10.1109/ASE.2017.8115663

[26] Jingyao Zhou, Lei Xu, Gongzheng Lu, Weifeng Zhang, and Xiangyu Zhang. 2023. NodeRT: Detecting races in node. js applications practically. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Seattle, WA, USA, 1332–1344. doi:10.1145/3597926.3598139