An empirical study of async wait flakiness in front-end testing

Yu Pei[1], Sarra Habchi[2], Renaud Rwemalika[1], Jeongju Sohn[1] and Mike Papadakis[1]

[1]University of Luxembourg, Luxembourg
[2]Ubisoft, Montreal, Canada

Abstract

Automated front-end regression testing is an essential part of web development, allowing fast release cycles while maintaining high-quality requirements. However, due to the asynchronous nature of web applications, front-end testing is sensitive to Async Wait flakiness which reduces the usefulness of such test suites by introducing false alarms. In this work, we conducted an empirical study to investigate the causes and the impact of Async Wait flakiness in front-end testing. To do so, we build a dataset of 62 tests exhibiting reproducible Async Wait flakiness associated with a clean fix commit, which becomes the foundation of our study. Our preliminary results suggest that tests relying on an explicit time to wait in order to synchronize the tests tend to create more flakiness (38 instances) than synchronizing on the status of DOM elements (24 instances).

Further study shows that where time-based issues are typically addressed by increasing the time to wait, DOM-based issues are resolved by actually introducing a missing synchronization point. We conclude our study with an analysis of the different implementations of synchronization mechanisms to provide tool manufacturers with concrete insights on how to improve their solutions.

Keywords

front-end testing, async wait, flaky tests, empirical study

1. Introduction

To remain competitive, companies are required to adapt to new business requirements in a timely fashion and fix potential defects or vulnerabilities as soon as they are detected to minimize any negative impact on their consumer base. As such, time to deployment has to be reduced to its minimum without compromising on the quality of the product shipped to production. To address these potentially conflicting requirements, software producers largely adopted automation testing to ensure the quality of the software they deploy. This practice has proven to be efficient at improving software quality[1] and allowing for rapidly finding vulnerabilities[19].

Regression testing is one of the software testing practices typically adopted by software-producing companies[20]. It ensures that the system under test (SUT) still functions as expected after any code changes. To achieve this goal, every time a developer makes a change to the SUT, the regression test suite is executed against the SUT. If any of the tests fail, it is an indication

yu.pei@uni.lu (Y. Pei); sarra.habchi@ubisoft.com (S. Habchi); renaud.rwemalika@uni.lu (R. Rwemalika); jeongju.sohn@uni.lu (J. Sohn); michail.papadakis@uni.lu (M. Papadakis)

CEUR Workshop Proceedings (CEUR-WS.org)

that the newly introduced change broke a requirement exercised by the test suite. However, this signal may contain noise, or false alarms, where some tests fail even though their requirements are actually met. As a result, builds fail where they should have actually passed, increasing not only the time to market but also production cost. This effect is even more pronounced in the case the failures are non-deterministic.

In the past years, researchers have increased their efforts to address non-deterministic test failures, also known as flaky test[2][3][4][5]. A flaky test script is one that might non-deterministically pass or fail on the same test code, resulting in different outcomes in different runs with no changes[6][7]. To detect flaky tests, the majority of the proposed solutions rely on test rerun where a test script is executed a certain number of times and is marked as flaky if the outcome differs in at least one execution[21].

Detecting flaky tests is an important challenge as they bring many problems:

1. Increase the debugging cost. Suppose the developer does not know that the test is flaky. The developer might try to spend plenty of time debugging only to find the observed test failure is not due to the recent changes but due to a flaky test.[7] Flaky tests often require hours or even days to debug. As Gruber et al.[22] explain, it takes approximately 170 runs of a test case to determine with certainty whether a test is flaky.
2. Undermine developers' trust in testing. The inconsistencies in the test results of flaky tests over given code changes can cause developers to lose faith in the tests themselves.
3. Hide real bugs. If a flaky test fails randomly, developers tend to ignore its failures and thus might miss real bugs[9]. It is thus important to determine whether a test has become flaky.

Due to the effects of flakiness and flakiness that can come from the test itself, researching flakiness in front-end testing can help developers by providing insight into effective detection and prevention methods. In this work, we focus our investigation on regression tests that are targeting the user-facing layer of the web applications; we refer to this type of test as front-end tests. Our preliminary investigation on front-end testing shows that flaky test failure often occurs under the following four circumstances:

- Interact with Document Object Model (DOM). DOM represents a web page. Depending on our test script, it can become flaky when there is an update to the DOM's style, structure, and content. It occurs when our test suite attempt to interact with elements in the DOM that do not render consistently.
- Render resources. Flaky tests in this category attempt to perform an action on a User Interface (UI) component or UI resources before resources are fully rendered.
- Transition and Animation. Animations cause the component to be highly time sensitive[14]. Due to the sensitivity of animation scheduling, judging the progress of an animation based on an element's state can lead to problems.
- User interface. Front-end testing is an interactive testing process that includes diverse user interactions, such as keyboard input events, mouse click events, etc. This diversity in interactions may produce unexpected flaky test failures.

The four potential circumstances for flakiness identified by our preliminary study show a strong predominance of the Async Wait category defined by Luo et al.[9]. Note that this category is not only prevalent in front-end testing. Indeed, Romano et al.[14] collected commit data to analyze, and they discovered that 45% of them fell into the Async Wait category. As such, this paper focuses on the Async Wait category of flakiness, which we refer to as Async Wait flakiness in the remainder of this paper, and aims at providing a deeper understanding of the causes of flakiness as well as providing developers with useful information to address this issue.

To do so, we carry out an empirical investigation on the Async Wait flakiness in front-end testing, to provide a better understanding of flakiness in web applications. We collected flaky test cases from GitHub written in JavaScript and conduct our analysis. We specifically searched GitHub for web apps and searched for flaky test-relevant commits using the keywords "async, wait, timeout", "flaky" and "flakiness". We discovered 62 commits in 26 web projects by manually reviewing the retrieved changes. We investigated the root cause and fixed the strategy of flakiness by analyzing each commit. By identifying the causes and fixing strategies for front-end flaky tests, our study intends to determine if our proposed characterization of flakiness varies from those researches on general software systems and if there are domain-specific flakiness patterns.

The main contributions of this study are as follows:

1. We investigated the main causes and fixing strategies behind tests exhibiting Async Wait flakiness.
2. We derive a reproducible dataset of 26 projects with 62 tests exhibiting Async Wait flakiness.
3. We compare fixing strategies for Async Wait flakiness from different testing frameworks. We investigate the relationship between the characteristics of different synchronization mechanisms, such as their respective ease of use and the likelihood of inducing Async Wait flakiness.
4. Our study provides developers and tool manufacturers with insights on the potential introduction of Async Wait flakiness.

2. Related work

Researchers have studied flaky tests and proposed several approaches to classify, identify, and fix flaky tests. Vahabzadeh et al.[13] carried out a comprehensive quantitative and qualitative study on test bugs, which are problems that might cause a test to fail or pass depending on whether or not the test code is right. Researchers identified three major causes of semantic bugs, flaky tests, and environmental bugs. Luo et al.[9] introduced ten major causes of flakiness (e.g., Async Wait, Concurrency, Test Order Dependency). Gao et al.[10] conducted a study that concluded that reproducing flaky tests can be difficult.

Although we focus on front-end tests, our experience has been similar. Among these works, we find some that try to understand the reasons for flakiness and identify the causes from developers[8]. Lam et al.[11] study the lifecycle of flaky tests in large-scale projects at Microsoft

by focusing on the timing between flakiness reappearance and the time to fix the flakiness. Terragni et al.[12] proposed a technique to run flaky tests in multiple containers with different environments simultaneously. As summarized in Romano et al.[14], flakiness problems are caused by a high diffusion of the Async Wait category, which happens when a test script makes asynchronous calls without waiting for the results.

Moreover, there are many works with different goals from ours that propose techniques and tools to diagnose flaky test scripts[15] or detect them in a test suite[16] or fix the flaky tests[17]. To prevent and mitigate the negative impact of flaky tests during the web testing workflow, we focus our attention on the async wait flaky test, since this type of problem is one of the main causes of web testing flakiness. We aim to advance the state-of-the-art in the field of research by providing valuable insights.

3. Methodology

3.1. Research Question

Our first research question regards the origin of Async Wait flakiness. More specifically, we aim to define a finer-grain nomenclature to classify Async Wait flakiness based on its origin. Thus, we ask:

RQ1 *What are the main root cause categories behind Async Wait flakiness?*

To answer this question we collected 26 projects with 62 flaky tests where all tests are related to Async Wait flakiness. Each test makes at least one asynchronous call or asynchronous wait and passes if it completes on time, but fails if it ends too early or too late. Then, we investigate each test by analyzing its associated commits to explore the main causes of flakiness and fixing strategies employed to remove it. Different from prior work, our research tries to reveal the main causes behind Async Wait flakiness in front-end tests. To do so, we formulate the following research question:

RQ2 *How do developers fix with Async Wait flakiness?*

To understand how developers identify and fix Async Wait flakiness, we look into the changeset between the flaky and fixed versions of the test. Additionally, we study how to effectively pick synchronization mechanisms to minimize the test execution time. Finally, the last step of our endeavor regards the capabilities offered by different testing frameworks. More specifically, we ask:

RQ3 *How do developers use different test frameworks to handle Async Wait flakiness?*

With this research question, we intend to examine whether or not actions to fix Async Wait flakiness are consistent across different testing frameworks. As such, we classify the type of changes performed within the tests relying on each framework and extract utilities provided by the frameworks to help developers to avoid Async Wait flakiness.

3.2. Data Collection

To identify commits that likely fix flaky tests, we choose to search open-source projects on GitHub. During this process, we follow a procedure similar to the one used in Luo et al.[9]. We search for the keywords ”web testing”, “flaky”, and “flakiness” in commits and projects written in JavaScript. When executing this query against the Github Search API, over 600 web projects are initially returned as a result. To ensure the projects are related to Async Wait flakiness in front-end tests, a second filter[14] is performed using the keywords ”UI”, ”DOM”, ”async”, ”await”, ”delay”, ”timeout”.

Next, we proceed to a manual analysis in order to identify the commits that are modifying test code and isolate the commits related to fixing flaky tests. As a result, we identify roughly 200 projects with commits potentially fixing Async Wait flakiness. To validate this step and guarantee that the tests are flaky, we ensure that the flakiness is reproducible. Therefore, we clone the projects from GitHub and rerun the tests which were modified by the commit we isolated in the previous step. To capture the flakiness, we execute each suspicious test 20 times; we label a test to be flaky if any of its execution results in a different outcome. This manual inspection, followed by rerunning the suspicious tests, resulted in 62 tests from 26 web projects being collected in the dataset.

The collected dataset in this study differs from those of previous studies in three perspectives. First, the flaky tests in our dataset are all reproducible from the way they were collected. Second, unlike the previous flaky test datasets containing various flaky tests, ours targets a specific direction that we only focus on, namely, Async Wait flakiness in the front-end test code. Third, each flaky test is associated with a fixed commit that can be analyzed to extract the fine-grained root cause of flakiness.

3.3. Study setup

RQ1 *What are the main root cause categories behind Async Wait flakiness?*

Tests can be flaky due to an async wait issue, where an asynchronous call is made or an asynchronous await is performed, but the result is not properly waited for before using it. In the context of testing a web application, async failures may occur when the application runs an asynchronous operation that must be completed before the application state or UI is ready to be tested. Without the appropriate synchronization mechanism between the test and the SUT, during its execution, the test may perform assertion or interact with elements of the web page that are not yet available. Thus, in addition, to collecting flaky tests, we also want to provide explanations for the observed non-deterministic behavior and explore the main causes of async wait flakiness in web front-end tests.

We manually analyze each test and review the test code and developers’ fixes to assign each test to a category. In addition, we rerun each test code to get error messages, part of the cases are flakiness caused by time issues, and other parts of the cases are element-related, such as that the element does not exist, etc. Moreover, DOM and time are often the primary concerns of developers and testers when it comes to web front-end testing since they directly influence how web pages display and load.

Table 1
Summary of Cause Categories

Categories	Description of causes	Number of tests
DOM-related	DOM rendering and usage	24
Time-related	Insufficient waiting time or exceeded time limit	38

Indeed, we identify two categories based on the call performed by the test at the fixed location: (1) time-related, such as ”await page.waitForTimeout()” or ”await wait()” where the synchronization point is an explicit amount of time; and (2) DOM-related, such as ”await page.waitForSelector()” where the synchronization point depends on the rendering state of a specific DOM element. This definition allows us to determine if the fix actions are DOM-related or time-related. For example, directly extending the wait time is associated with a time-related fix and the introduction of a method to wait for elements to be rendered is associated with a DOM-related fix.

Note that this classification is already used in the literature and by practitioners. Spadini et al.[23] relying on developers' perception to create a classification of test smell severity, mentioned that using time-related locators, *i.e.* sleepy test, might introduce flakiness. This observation is corroborated by practitioners from the industry where developers believe that a good front-end testing framework should avoid using explicit thread waiting for the sake of system stability[18][24]. This is why, relying on the asynchronous instability reasons proposed by previous work, we have carried out two-class categorization, one is DOM-related, and the other is time-related. We describe those categories in Table 1. Based on the results of our analysis, we can quickly classify Async Wait flakiness and propose an appropriate repair method for each cause.

RQ2 *How do developers fix with Async Wait flakiness?*

We manually analyze all the fixes introduced by the developers by reviewing the comments and the changeset of each commit. First, we clean the changeset to remove any changes unrelated to fixing flakiness such as refactoring operations. Then, we analyze the fine-grained changes to discover what contributes to the flakiness of the test suite.

For example, in the absence of the ”waitFor” method in the test code which takes a DOM element as an attribute, to fix flakiness, a DOM-based synchronization point is introduced. DOM-related fixes are typically performed by introducing method calls such as ”waitForElements” or ”waitForBeTrue”. Thus, during the manual analysis, we extract actions consisting of insetting these methods.

Code snippet 1

```
async () => {
    - expect(connection.streams).to.have.length(0)
    + await pWaitFor(() => connection.streams.length === 0)
}
```

For instance, code snippet 1 shows the introduction of a wait for true condition method, to offer a DOM-based synchronization point for the test. Another example can be found in the Shopify-theme-inspector project (code snippet 2) where the developer directly adds the method to wait until the element renders before executing the assertion.

Code snippet 2

```
      async () => {
          await page.$eval('[data-refresh-button]', elem => elem.click());
        + await page.waitForSelector('.d3-flame-graph')
      }
```

On the other hand, time-related fixes are typically solved by increasing the time attribute (code snippet 3) or by adding an explicit wait call. We observe that most flaky behavior occurs due to a short timeout caused by elements waiting for results or callbacks being delayed.

Code snippet 3

```
      it("should see loaded profiles on the team page", function () {
          cy.visit("/team");
        - cy.wait(250);
        + cy.wait(1000);

          expect(cy.data("contributor-handle-GithubPerson")).to.exist;
      });
```

To conclude, our preliminary results suggest that developers use different strategies for fixing Async Wait flakiness depending on whether it is time-based or DOM-based. In order to solve DOM-related flakiness, they usually introduce a synchronization point on a specific DOM element, whereas for time-related flakiness, they usually add or extend the waiting time.

RQ3 *How do developers use different test frameworks to handle Async Wait flakiness?*

In order to compare the performances of the frameworks used to exercise the tests of our dataset, we extract from the repository the dependencies from the build configuration (package.json). Table 2 shows the outcome of this process and suggests that our dataset is composed of four test automation frameworks. For each testing framework, we build an exhaustive list of the Synchronization Mechanism Methods (SMM) provided by the framework; for each SMM from this catalog, we compute code metrics associated with the ease of use and the way they interact with the SUT. Analyzing the entirety of the tests from our projects, regardless of their flakiness, we assign to each test the SMMs gathered from each framework. If a SMM is present

Table 2
The use of different testing frameworks

Test framework	Number of flaky tests
Jest	27
Cypress	16
Mocha	14
Puppeteer	5

in the fix commit identified in RQ2, it is labeled as flakiness inducing and if not, it is labeled as non-inducing.

Finally, we perform a statistical analysis to define whether some types, which can be determined using the pre-computed code metrics, of SMMs are more likely to generate flakiness. Answering this question allows us to propose strategies that can be implemented by tool providers to assist developers to avoid introducing flakiness.

Our preliminary results suggest the following:

- Jest: developers solve Async Wait flakiness mainly by using separate wait, like page.wait() method or delay functions.
- Cypress: developers resolve Async Wait flakiness mainly by testing the framework's integrated cy.wait() method.
- Mocha: Async Wait flakiness is also resolved using a separate wait method.
- Puppeteer: Async Wait flakiness is resolved using the framework's integrated page method.

4. Conclusion and Future work

In this paper, we demonstrate that flaky tests are equally common in front-end testing. Among various types of flaky test failures, we specifically target Async Wait flakiness, one of the most prevalent categories of flakiness identified in prior work and throughout our preliminary study. For the benefit of the flakiness community and for our study, we generate a dataset of tests exhibiting Async Wait flakiness. With this dataset, we start to analyze the root cause of Async Wait flakiness and study how they can be fixed effectively by investigating the developers' fixes, specifically concerning the type of employed testing framework and more specifically their implementation of the different synchronization mechanisms. Our current observations suggest that this study will provide useful insights for future research on flaky tests in front-end testing.

As part of our future research, we will demonstrate that the proposed procedure is generalizable and effective using other test suites, as well as investigate the possibility of developing automated fixing strategies.

References

[1] Garousi, V., Felderer, Developing, verifying, and maintaining high-quality automated test scripts. IEEE Softw. 33, 68–75 (2016)

[2] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018, pp. 433–444.
[3] 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, pp. 101–111.
[4] Leotta, Maurizio, et al. "A family of experiments to assess the impact of page object pattern in web test suite development." 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). IEEE, 2020.
[5] Leotta, Maurizio, Andrea Stocco, Filippo Ricca, and Paolo Tonella. "Pesto: Automated migration of DOM-based Web tests towards the visual approach." Software Testing, Verification And Reliability 28, no. 4 (2018): e1665.
[6] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding Flaky tests: The developer's perspective," in Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE). ACM, 2019, pp. 830–840.
[7] Zolfaghari, Behrouz, et al. "Root causing, detecting, and fixing flaky tests: State of the art and future roadmap." Software: Practice and Experience 51.5 (2021): 851-867.
[8] Eck, Moritz, et al. "Understanding flaky tests: The developer's perspective." Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2019.
[9] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of Flaky tests," in International Symposium on Foundations of Software Engineering (FSE). ACM, 2014, pp. 643–653.
[10] Z. Gao, Y. Liang, M. Cohen, A. Memon, and Z. Wang. 2015. Making system user interactive tests repeatable: When and what should we control?. In ICSE. Florence, Italy, 55–65.
[11] W. Lam, K. Mus ̧lu, 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, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1471–1482.
[12] F. F. Valerio Terragni, Pasquale Salza, "A container-based infrastructure for fuzzy-driven root causing of flaky tests" 2020.
[13] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), Sept 2015, pp. 101–110.
[14] Romano, Alan, et al. "An empirical analysis of UI-based flaky tests." 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021.
[15] Morán Barbón, Jesús, et al. "FlakyLoc: flakiness localization for reliable test suites in web applications." Journal of Web Engineering, 2 (2020).
[16] Bell, Jonathan, et al. "DeFlaker: Automatically detecting flaky tests." 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018.
[17] Shi, August, et al. "iFixFlakies: A framework for automatically fixing order-dependent flaky tests." Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2019.

[18] Ricca, Filippo, and Andrea Stocco. "Web test automation: Insights from the grey literature." In International Conference on Current Trends in Theory and Practice of Informatics, pp. 472-485. Springer, Cham, 2021.
[19] "Flaky tests at google and how we mitigate" https://testing.googleblog.com/2016/05/□aky-tests-at-google-andhow-we.html.
[20] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ACM, 2016, pp. 426–437.
[21] M. Fowler. (2011) Eradicating non-determinism in tests. [Online]. Available: https://bit.ly/2PFHI5B
[22] Gruber, Martin, et al. "An empirical study of flaky tests in python." 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). IEEE, 2021.
[23] Spadini, Davide, et al. "Investigating severity thresholds for test smells." Proceedings of the 17th International Conference on Mining Software Repositories. 2020.
[24] Bushnev Y (2019) Top 15 ui test automation best practices. URL https://www.blazemeter.com/blog/top-15-ui-test-automation-best-practices-you-should-follow