Addressing the Synchronization Challenge in Cypress End-to-End Tests

Thiago Santos de Moura
Federal University of Campina Grande
Campina Grande, Brazil
thiago.moura@copin.ufcg.edu.br

Everton L. G. Alves
Federal University of Campina Grande
Campina Grande, Brazil
everton@computacao.ufcg.edu.br

Regina Letícia Santos Felipe
Federal University of Campina Grande
Campina Grande, Brazil
regina.felipe@ccc.ufcg.edu.br

Cláudio de Souza Baptista
Federal University of Campina Grande
Campina Grande, Brazil
baptista@computacao.ufcg.edu.br

Ismael Raimundo da Silva Neto
Federal University of Campina Grande
Campina Grande, Brazil
ismael.silva.neto@ccc.ufcg.edu.br

Hugo Feitosa de Figueirêdo
Federal Institute of Paraíba
Esperança, Brazil
hugo.figueiredo@ifpb.edu.br

ABSTRACT

Automated end-to-end testing plays a crucial role in modern web software projects, helping testers identify faults within complex applications and shorten development cycles. Frameworks such as Cypress are essential to provide a comprehensive testing environment with features that facilitate better access and validation of page elements. However, time-related challenges (synchronization issues) remain a significant concern in such suites. Testers need to be aware of these challenges and employ appropriate waiting mechanisms to ensure test reliability. This paper presents a catalog of waiting mechanisms for Cypress tests and a set of empirical studies that investigate the potential impact of synchronization issues and waiting mechanisms on test suites. Our studies examine the suites of an open-source and industrial project. Our findings reveal that up to 32% of a suite can break due to synchronization issues, exposing flaky tests. Subsequently, we revised the suites by applying four waiting mechanisms (*Static Wait*, *Stable DOM Wait*, *Network Wait*, and *Explicit Wait*). *Network Wait* and *Explicit Wait* emerged as the most promising strategies leading to no breakages.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

end-to-end testing, flaky tests, synchronization, cypress, empirical studies

1 INTRODUCTION

End-to-end (E2E) testing plays an important role in the industry, especially in preventing functionality regression [1, 24]. E2E web test scripts provide a sequence of instructions for a testing framework, directing its interactions with the Application Under Test (AUT) [18]. These instructions encompass tasks such as element selection, interaction, and checks for the presence of elements within the application interface. These actions not only guide test execution but also serve as oracle assertions, establishing criteria to determine the expected AUT behavior [6, 28].

Selenium[1] and Cypress[2] are two well-known frameworks for automated E2E web testing [21]. While Selenium has traditionally been the default choice for E2E testing, providing a framework for validating web application functionality [5], Cypress has recently emerged as a valuable alternative, gaining substantial attention within the test automation community [23, 29].

Data from official NPM trends[3] reveal that Cypress downloads more than tripled between 2021 and June 2024 (exceeding 5.7 million), while for the Selenium WebDriver, the number of downloads remained nearly the same during the same period (around 2 million). This suggests that testers prefer Cypress for E2E testing with JavaScript. Possible reasons for this shift include the fact that Cypress includes new features that have attracted the attention of testers. For instance, Cypress provides a simple setup process, where a single download includes all essential browser engines, frameworks, and assertion libraries, eliminating the need for separate installations [7, 10]. This infrastructure challenge is a well-known problem reported by Selenium users [12, 19]. Additionally, Cypress also has a vibrant and active community that helps evolve the framework with NPM packages and add-ons. Although very popular among developers, we found little research on E2E testing using Cypress. We believe that this shift in focus to Cypress presents an important opportunity for new research on how this framework addresses underlying E2E testing challenges.

To ensure cost-effectiveness in E2E testing, it is imperative for a test suite to demonstrate both efficiency and reliability [8, 16]. Therefore, flaky tests are an issue to be addressed. Flaky tests are characterized by nondeterministic behavior that produces random and inconsistent results, which can compromise the reliability of an E2E web test suite [30]. In the continuous integration environment, flaky tests present even greater risks, as a false failure may obstruct new builds and/or releases [37]. Moreover, flaky tests often diminish the tester's confidence in the suite, as a significant amount of time might be spent debugging non-existent faults [13].

Mitigating flaky tests in the context of E2E web testing is a challenging task. For that, it is essential to comprehend and address their underlying causes. Several studies have identified possible root causes of flakiness. Asynchronicity and concurrency issues are often listed as key factors for flaky tests in E2E tests [19, 22, 25, 36, 37]. Consequently, effective techniques for addressing flaky tests should prioritize handling these factors to yield significant improvements.

[1] https://selenium.dev
[2] https://www.cypress.io
[3] https://npmtrends.com/cypress-vs-selenium-webdriver

Asynchronicity and concurrency issues are so prevalent in E2E web testing that they are frequently referred to as the *synchronization challenge*[*] [19, 37]. This challenge refers to scenarios where the tester does not properly consider response time when creating test scripts, which may result in flaky tests and intermittent failures during the execution of the test script. Such issues often arise when test scripts attempt to perform actions on a web page not yet ready to handle them, leading to breakage in the test execution. For instance, when a test script logs into a web application and tries to interact with elements on the home page, if the server response is not received quickly enough, it may try to execute instructions before the application's home page is fully loaded. This premature execution can lead to test breakages that mistakenly indicate fault detection [14, 15, 38].

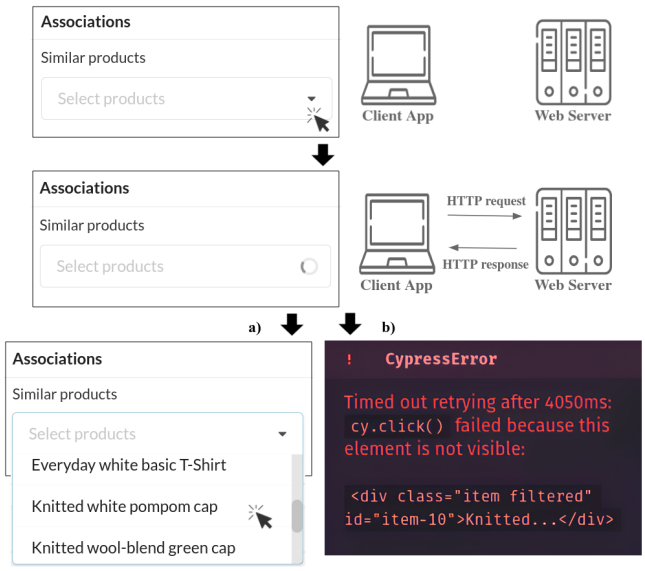

Figure 1: Example of a testing action that might fail due to synchronization issues.

The Cypress framework tries to attenuate this issue by executing scripts on the browser side, discerning the application's transitional state after an action, and waiting for the complete page to load. However, the *synchronization challenge* remains when the test script interacts with elements that trigger asynchronous calls [11, 27, 39]. Such calls occur whenever a page interaction requires data retrieval from a Web server. Figure 1 exemplifies such a scenario. Suppose a Cypress test script tries to interact with the tenth product that appears after accessing the *Select products* drop-down input. The *click* action triggers an asynchronous call that returns the list of registered products. The *select* action becomes viable only after the request is fulfilled and the application lists the found product (Figure 1-a). However, if the test does not properly handle synchronization, the test script might try to click on a product before the list is loaded, leading to a test breakage (Figure 1-b). Reasons for an application delay that may lead to synchronization issues are numerous (e.g., network latency, slow server response, complex database queries) and often cannot be controlled or properly anticipated by the tester.

Despite being the state-of-the-art framework for E2E web testing and including a native waiting strategy, Cypress cannot avoid synchronization issues. This highlights that synchronization remains an open problem that requires proactive measures from testers. In this paper, we discuss a catalog of waiting mechanisms to avoid synchronization issues in Cypress tests, and we propose a new mechanism, the *Network Wait*. Moreover, we evaluate the impact that synchronization issues may generate in practice, and we investigate the effectiveness of the mechanisms in dealing with them. By doing so, we offer new insights and ways to help testers to better construct E2E suites, we provide evidence of the practical effectiveness of different waiting mechanisms, and we highlight the importance of adopting advanced strategies to address asynchronously loaded content [27].

The contributions of this work are fourfold:

- A catalog of waiting mechanisms for Cypress tests;
- A novel waiting mechanism (*Network Wait*) for Cypress;
- Empirical studies that demonstrate the deterioration that synchronization issues can generate on two E2E web test suites; and
- Empirical studies that evaluate the efficiency of four waiting mechanisms in addressing synchronization issues.

The remainder of the paper is structured as follows. Section 2 presents a catalog of waiting mechanisms for Cypress that can be employed to address synchronization issues. Section 3 focuses on a series of empirical studies designed to demonstrate the impact of the synchronization challenge and to evaluate the effectiveness of different waiting mechanisms. In Sections 4 and 5, we discuss potential threats to validity and related work, respectively. Finally, in Section 6, we present the concluding remarks and potential avenues for future research.

2 A CATALOG OF WAITING MECHANISMS

The *synchronization challenge* is well-known in automated E2E tests. In this context, waiting mechanisms can act as traffic lights for testing, controlling when to stop, when to go, and when to wait. This orchestration may help avoid synchronization issues.

To systematically identify relevant waiting mechanisms, we conducted a literature mapping using common keyword searches combined with snowballing, a technique in which references from selected articles are reviewed to identify additional relevant studies, ensuring a comprehensive review [40]. We subsequently validated the importance of these mechanisms among experienced testers from a partner company. Our review of the literature identified four waiting mechanisms applicable in this context: *Implicit Wait*, *Static Wait*, *Explicit Wait*, and *Fluent Wait* [3, 12, 26]. However, the referenced works focus solely on Selenium test scripts. An exploration of the grey literature introduced a new mechanism specific to Cypress: *Stable DOM Wait*[5]. Furthermore, we introduce a novel mechanism exclusive to Cypress: the *Network Wait*. As a result, we compile a catalog of waiting mechanisms for Cypress tests[*].

[*]Also referred to as the *asynchronous* or *race condition challenge*.

[5]https://github.com/narinluangrath/cypress-wait-for-stable-dom

[*]A detailed version of the catalog is available on our website: https://noto.li/mhRfQe

Although some mechanisms are not native to Cypress, they can be implemented. For instance, we discuss the implementation of *Explicit Waits* using external Cypress dependencies. Additionally, mechanisms such as *Stable DOM*, which have not been addressed in any related work using Selenium, are currently exclusive to Cypress. Furthermore, the proposed mechanism incorporates a concept unique to Cypress: intercepting requests. Consequently, this new mechanism is only supported by Cypress.

In this section, we present our catalog by discussing each mechanism, exploring its pros and cons, relevance, and implementation details within the Cypress context. To the best of our knowledge, our catalog is the first to compile and demonstrate the use of these waiting mechanisms in the Cypress framework. For each mechanism, we provide a general description, followed by an example of how to implement it with Cypress, and a discussion on implication, benefits, and possible drawbacks. We hope this catalog aids testers in gaining a better understanding and handling of synchronization issues in Cypress tests.

2.1 Implicit Wait

2.1.1 Description. The *Implicit Wait* acts as a single traffic light overseeing waits for the entire test script. Unlike other waiting mechanisms inserted before specific commands, the *Implicit Wait* is a global setting that uniformly impacts all commands. It enforces a time limit for every interaction, ensuring that no command proceeds until it is either feasible to continue or a specified waiting period has elapsed. Selenium employs an *Implicit Wait* [12]. For Cypress, although this mechanism is not officially labeled as such, it includes a general automatic waiting feature.

2.1.2 Usage Scenario with Cypress. When a tester utilizes commands like *click*, *type*, or *clear*, Cypress implicitly enforces those commands to wait for the respective operations to complete. The default timeout for Cypress commands is 4000 milliseconds, but it can be customized based on specific requirements, as exemplified in Listing 1 (line 1). In this example, the `defaultCommandTimeout` is set to two seconds. Consequently, when a test script interacts with a page element, Cypress waits two seconds to acquire the element before proceeding with the *click* command.

```
1 Cypress.config('defaultCommandTimeout', 2000);
2
3 cy.get('[id="product-select"]').click();
4 cy.get('[id="item-10"]', { timeout: 5000 }).click();
```

Listing 1: Example of Implicit Wait configuration in Cypress.

Additionally, Cypress offers a timeout option within commands, allowing custom waiting times for individual commands. For instance, in Listing 1 (line 4), the timeout for the command that acquires the desired item was set to five seconds. This gives the tester control over how long to wait for the asynchronous call to be fulfilled and for the item to appear on the page. It is important to note that altering the timeout for a specific command extends the basic characteristics of an *Implicit Wait*, as it solely evaluates the action of the command without considering any other condition.

2.1.3 Implications, Benefits and Drawbacks. The use of *Implicit Waits* can simplify wait management by offering a consistent approach across test commands, which may reduce the need for extensive wait coding and improve script readability and maintenance. However, they may cause inefficiencies and unreliable test outcomes due to the use of fixed waiting times that may not match the actual application response times, potentially slowing the tests and leading to flakiness [33, 36]. Complex synchronization issues, such as waiting for visible elements with specific attributes, may require alternative strategies for more precise handling.

2.2 Static Wait

2.2.1 Description. In Selenium, testers often use `Thread.sleep()` to pause the execution of a test script for a specific duration. This pause is not dependent on any conditions or external factors, and the script resumes only after the pre-defined time has elapsed. For Cypress, a similar behavior is achieved using the `wait` command.

2.2.2 Usage Scenario with Cypress. In Listing 2, we exemplify the use of this mechanism. After interacting with the *Select products* element, we use the `wait` command to pause the test execution for one second (line 2). After the waiting period is over, the test execution performs the click on the item (line 3).

```
1 cy.get('[id="product-select"]').click();
2 cy.wait(1000);
3 cy.get('[id="item-10"]').click();
```

Listing 2: Example of Static Wait use in a Cypress test script.

2.2.3 Implications, Benefits and Drawbacks. The use of *Static Waits* can impact reliability and efficiency. Although it provides a straightforward synchronization method by pausing execution for a fixed duration, this can lead to inefficiencies in dynamic web applications with varying loading times. Fixed wait times may cause unnecessary delays if the application is ready early or result in test breakages if it takes longer than expected, increasing test flakiness [26, 36]. It may also mask underlying synchronization issues, making it less effective in dynamic testing environments. *Static Waits* should be used moderately and combined with adaptive waiting strategies.

2.3 Explicit Wait

2.3.1 Description. In Selenium, *Explicit Waits* are a powerful mechanism for ensuring that a test script proceeds only when specific conditions are met [34, 37]. This feature is indispensable when dealing with web elements that might take an unforeseeable amount of time to load or become interactive.

For implementing *Explicit Waits* testers often use external dependencies such as *cypress-wait-until*[7]. This dependency introduces a custom command called `waitUntil` that enables implementing *Explicit Waits*. This command helps the tester to use common conditions (e.g., the presence of an element, specific values for global variables) or custom conditions.

2.3.2 Usage Scenario with Cypress. Listing 3 (line 3) uses `waitUntil` to make the test execution wait up to 10 seconds for the desired item to exist on the web page. The arrow function encapsulates the condition, ensuring that the button is both present and acquirable.

[7] https://www.npmjs.com/package/cypress-wait-until

Upon meeting this condition, the script subsequently simulates a user click action on the button using the `click` function (line 8). This approach ensures reliable user interaction simulations in dynamic web environments.

```
1 cy.get('[id="product-select"]').click();
2
3 cy.waitUntil(() =>
4   cy.get('body').then(($body) =>
5     $body.find('[id="item-8"]').length > 0),
6       { timeout: 10000 });
7
8 cy.get('[id="item-10"]').click();
```

Listing 3: Example of Explicit Wait use in Cypress.

2.3.3 Implications, Benefits and Drawbacks. *Explicit Waits* can enhance test reliability and accuracy by allowing tests to wait for specific conditions. However, this approach also increases script complexity and maintenance requirements, as defining and managing waiting conditions can make test scripts more challenging to read and maintain [35]. Additionally, if not properly implemented, *Explicit Waits* can still result in timeouts or missed conditions, leading to test breakages despite their improved control capabilities.

2.4 Fluent Wait

2.4.1 Description. A *Fluent Wait* can be seen as a customizable version of an *Explicit Wait* [26]. It leverages the same commands as *Explicit Waits*, but with additional parameters that testers can tweak to tailor the waiting strategy. These parameters include options such as polling frequency and custom error messages for timeouts.

2.4.2 Usage Scenario with Cypress. In Listing 4, the tester wants to ensure that the execution of the test continuously checks for the presence of a specific item on the web page over a 10-second period. It evaluates this condition at one-second intervals. The `waitUntil` command (line 3) encapsulates this condition and introduces the `interval` and `errorMsg` parameters. The `interval` parameter specifies that the evaluation should happen every one second, and the `errorMsg` parameter provides a custom error message in case of a timeout.

```
1  cy.get('[id="product-select"]').click();
2
3  cy.waitUntil(() =>
4    cy.get('body').then(($body) =>
5      $body.find('[id="item-10"]').length > 0),
6        { timeout: 10000,
7          interval: 1000,
8          errorMsg: 'This is a custom error message'
9        });
10
11 cy.get('[id="item-10"]').click();
```

Listing 4: Example of Fluent Wait use in Cypress.

2.4.3 Implications, Benefits and Drawbacks. *Fluent Wait's* flexibility enables customization of wait durations and check frequencies, enhancing test efficiency and reducing flakiness by adapting to dynamic application behavior. To fully exploit this flexibility, it is important to test various parameter configurations [26]. However, this adaptability can also increase script complexity and maintenance challenges. Poorly designed polling strategies may result in excessive resource use or longer waiting times, affecting overall performance. Thus, while *Fluent Wait* offers significant benefits, careful management is crucial.

2.5 Stable DOM Wait

2.5.1 Description. This mechanism addresses a critical need in E2E testing, which is to ensure the stability of the Document Object Model (DOM) for a specified duration before allowing the test flow to proceed. This is achieved by using the MutationObserver[8] interface to detect alterations in the DOM tree. This mechanism is particularly useful for visual regression testing, where minor DOM changes can impact a web page's visual appearance. Ensuring a stable DOM helps in capturing accurate visual snapshots.

2.5.2 Usage Scenario with Cypress. For Cypress users, this mechanism is accessible through an external library known as *cypress-wait-for-stable-dom*[9]. After importing this library, a tester can access the `waitForStableDOM` function, as shown in Listing 5.

```
1 cy.get('[id="product-select"]').click();
2 cy.waitForStableDOM({ pollInterval: 500, timeout: 5000 })
3 cy.get('[id="item-10"]').click();
```

Listing 5: Example of Stable DOM Wait use in Cypress.

In this example, the `waitForStableDOM` function is called with various parameters, including `pollInterval`, which specifies the duration for which the DOM must remain stable, and `timeout`, which sets the time limit for waiting. Importantly, the code in line 3 is executed only when the preceding command has completed its wait execution. This ensures that a test script interacts with a stable DOM, enhancing the reliability of its results.

2.5.3 Implications, Benefits and Drawbacks. *Stable DOM Waits* play a crucial role in test automation, particularly in scenarios requiring visual consistency and accurate DOM interactions. By ensuring that the DOM remains stable for a specified duration, this approach reduces false positives from transient changes and provides reliable test results. However, if the DOM is frequently updated, this mechanism may introduce delays in test execution. Balancing poll intervals and timeouts is essential to maintain both reliability and efficiency.

2.6 Network Wait

2.6.1 Description. *Network Wait* is a novel mechanism that closely observes requests made during test execution. Its purpose is to ensure that a test proceeds only when all these requests are completed. As new requests are initiated, a counter is incremented; upon request completion, the same counter is decremented. The test execution resumes its course when this counter reaches zero, remaining in this state for a predetermined period of time.

To the best of our knowledge, the *Network Wait* mechanism is unique to Cypress because it can intercept requests - a feature not available in Selenium. This mechanism monitors client-server communication during test execution, whereas current wait mechanisms focus solely on the DOM. Therefore, existing methods cannot replicate *Network Wait* behavior. We developed and published our own external NPM dependency for *Network Wait*, which is available to the Cypress testing community[10].

[8] https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver
[9] https://www.npmjs.com/package/cypress-wait-for-stable-dom
[10] https://www.npmjs.com/package/cypress-network-wait

2.6.2 Usage Scenario with Cypress. Listing 6 presents an implementation of this mechanism through Cypress native `intercept` command. The `routeHandler` function manages the `pendingCount` variable, which decreases with each response, ensuring patient waiting until all requests are completed. In our example, intercepts are configured to cover all URLs and HTTP methods (line 7), allowing the process to wait until the counter reaches zero before the test proceeds (lines 10-13).

This *Network Wait* is recommended in situations where the subsequent testing step is uncertain and a more generic waiting strategy is required, which is often the case for scriptless E2E testing [4]. It dynamically adapts to network conditions, reducing waiting times on faster networks and extending them on slower networks. This adaptability may lead to an effective balance between precision and efficiency during test execution.

```
1  let pendingCount = 0;
2  function routeHandler(request) {
3    pendingAPICount++;
4    request.on('response', () => pendingCount--);
5  }
6
7  cy.intercept('*', '*', routeHandler);
8
9  Cypress.Commands.add('waitNetworkFinished', () => {
10   while (pendingCount > 0) {
11     cy.log('Waiting for pending requests.');
12     cy.wait(500);
13   }
14   cy.log('All requests completed!');
15 });
16
17 cy.get('[id="product-select"]').click();
18 cy.waitNetworkFinished();
19 cy.get('[id="item-10"]').click();
```

Listing 6: Simplified implementation of Network Wait and its use in Cypress.

2.6.3 Implications, Benefits and Drawbacks. The *Network Wait* mechanism dynamically adjusts to network conditions, ensuring tests are executed accurately without premature execution or unnecessary delays. This approach is beneficial for scenarios with unpredictable network behavior and multiple asynchronous requests. However, it requires careful monitoring to manage edge cases like incomplete or stalled requests. Additionally, the use of polling loops (e.g., `cy.wait(500)`) can introduce minor delays, affecting overall test execution time.

3 EMPIRICAL STUDIES

In this section, we present the empirical studies performed to assess the impact that synchronization issues can cause in test suites. We also evaluate the efficiency of various waiting mechanisms to address this challenge. Our investigation is guided by the following research questions:

- RQ_1: How much of a test suite can break due to synchronization issues?
- RQ_2: How effective are waiting strategies to mitigate synchronization issues?

RQ_1 relates to our hypothesis that synchronization issues may cause test breakages due to temporal misalignments between test execution and system response, especially in repeated executions (e.g., 50 times) or less controlled environments. By addressing RQ_1, we hope to provide a practical understanding of how synchronization issues affect test suites. Our second hypothesis, related to RQ_2, is that waiting mechanisms could minimize synchronization issues in Cypress tests. Therefore, we aim to assess the performance of various waiting strategies in resolving these challenges.

To answer the research questions, we conducted empirical studies in two different scenarios. The first considers a test suite developed for an open-source scalable online store application, while the second considers a suite from an industrial project. In both cases, we investigated the impact of synchronization issues on the test suites and compared the effectiveness of different waiting mechanisms. Specifically, we compared *Static Wait* (with a default one-second delay), *Stable DOM Wait* (with a default one-second interval), *Network Wait*, and *Explicit Wait*. Although *Fluent Wait* and *Implicit Wait* were listed in our catalog (Section 2), we did not include them in our studies because the former can be seen as a customizable *Explicit Wait*, and the latter is an inherent setting in Cypress, automatically applied to all test scripts.

In our studies, we used Cypress version 12.17.2 in its default configuration (default timeout command of four seconds). The empirical studies were executed on a Desktop equipped with an Intel Core i7 10700KF processor, 32GB DDR4 3200MHz RAM, Nvidia GTX 1060 6GB GDDR5 graphics card, and a 1TB SATA SSD with 500Mbps/s.

3.1 Investigating the Impact of Synchronization Issues

For the first study, we selected the Sylius Standard[11] (version 1.12.4). Sylius is an open-source eCommerce framework known for its modular and flexible architecture, making it well-suited for developing customized online stores. It offers extensive configuration options and advanced features to manage catalogs, orders, payments, and shipping, among other aspects of eCommerce. We selected this application as object due to its wide range of functionalities. Additionally, this project is easy to use and popular, as evidenced by its high number of stars and active community engagement on GitHub. During the study, we used the sample data provided by the framework to populate the database and executed the entire web application within a single Docker container using an image built from its repository[12].

Synchronization issues often occur in applications where modules are potentially hosted on distinct machines, including the front-end, back-end, and database. Another potential scenario involves complex functionalities that trigger asynchronous calls and lead to prolonged database queries. To have a controlled experiment in which we investigate the potential degradation of the test suites, we emulate such scenarios. We deliberately introduced various levels of network delays into our testing environment. This process was inspired from a related work that discusses how varying loading times can contribute to test flakiness [26]. For that, we used the command-line utility tool Traffic Control[13]. Traffic Control is instrumental for network traffic management and manipulation, with a specific focus on bandwidth regulation and control.

[11] https://github.com/Sylius/Sylius-Standard, https://github.com/Sylius/Sylius
[12] https://gitlab.com/lsi-ufcg/cytestion/sync-study/sylius-showcase
[13] https://linux.die.net/man/8/tc

A total of five Sylius application instances were created for the purpose of our study. Four of these were equipped with Traffic Control to introduce delays, resulting in the following treatments: *Without delay*, *500 ms delay*, *1000 ms delay*, *1500 ms delay*, and *2000 ms delay*. These values were introduced with a reasonably significant gradual increase to simulate different network delays. Each instance has a unique Docker image tag, facilitating effortless execution and repeated database restoration. The artifacts of our study are available on our website[14].

To create the test suite, we recruited 48 students. They are last year Computer Science students with a solid programming and testing background. Prior to the study, they went through a comprehensive two-hour training session on E2E testing and Cypress. The students generated test cases for 25 distinct Sylius features. Each feature corresponded to a different part of the application, such as products, payments, and more. To maintain flexibility and emulate real-world test suite creation, we intentionally refrained from imposing specific requirements on what elements should be tested within each section. Instead, each student had the freedom to create test cases based on their understanding of the assigned section and the system behavior.

From the test cases created by the students, we validated and selected 200 tests to be used in our study. This selection was manually done by the first author to create a suite with valid test cases. We considered valid test cases that include interactions with multiple actionable elements and assertions confirming the expected behaviors. Notably, some students incorporated static waits in their tests, which resembles the behavior of experienced testers who would try to predict and treat possible synchronization issues. Also, it is important to highlight that we neither intentionally designed the testing sections nor guided the test case creation to include synchronization issues, as those issues could potentially arise in any part of an application or test case.

For the five created Sylius instances (*Without delay*, *500 ms delay*, *1000 ms delay*, *1500 ms delay*, and *2000 ms delay*), we executed the created suite 50 times, resulting in a total of 10,000 test case executions per instance. By repeating a test case execution for a given system instance, we intended to be able to detect possible synchronization issues and evaluate the suite reliability. A similar approach was performed in related studies [26, 27].

3.1.1 Results. Out of the 200 test cases, 35 (17.5%) exhibited flakiness during at least one execution. In Figure 2, these 35 test cases are ordered by the total number of breakages per instance. Interestingly, we found ten test cases that experienced the highest breakage rates, which means they were the most affected by delays (test cases 1-10). As the delays increased, the number of executions experiencing breakage drastically increased for those tests. We manually investigated all 35 tests and found that tests 1-10 interact with elements that trigger asynchronous calls. For instance, some of these test cases involve selecting a product, which is listed only after clicking the *Select products* drop-down input, creating a test breakage similar to the one described in Section 1. Another example is a test case that opens and closes a large image. This image is loaded after a clicking action. This test encountered breakage when a close action was performed when the image was not yet fully loaded. Tests 11-35, on the other hand, present a lower number of breakages and their flakiness were due to issues not related to synchronization, such as logic faults and dependencies on randomly generated data. Therefore, our second study (Section 3.2) focuses only on the ten test cases that include synchronization issues (tests 1-10).

Even though delays were applied universally, the impact on the remaining 165 test cases may have been less evident due to Cypress's native waiting mechanisms for page loading and the default implicit waits. However, the presence of asynchronous calls can still introduce synchronization issues, emphasizing the need to address such problems to maintain the robustness of a test suite.

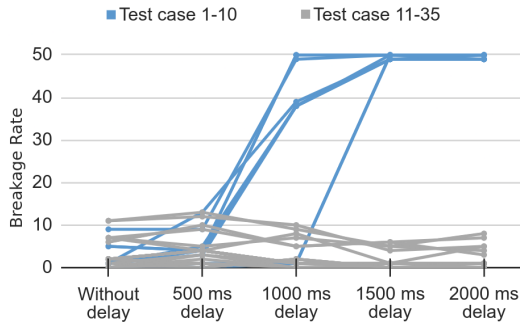

Figure 2: Number of test case breakages by Sylius instance.

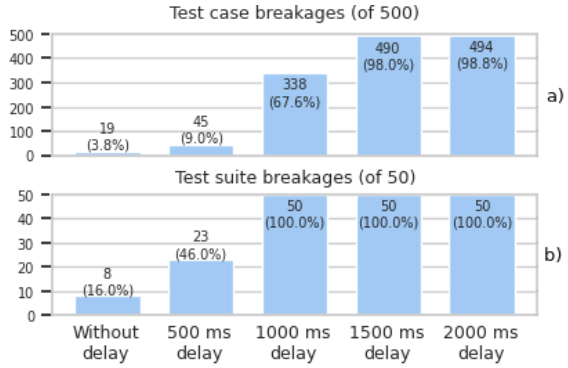

Figure 3: Breakage rates by test case and suite runs.

Synchronization issues often reflects in test flakiness. Therefore, the same test case may present different outputs (pass or fail) in different executions. Figure 3 illustrates test case and suite breakages for test cases 1-10 across different configurations. Each bar represents the breakage rate of test cases (out of 500 = 10 test cases x 50 runs) and test suite executions (out of 50 runs). While Figure 3-a refers to test case runs that experienced breakage, Figure 3-b presents test suite breakages. A test suite breaks when at least one of the 10 tests within the suite experiences breakage.

We can see that, even for the scenario without delays, synchronization issues were found (19 breakages across 8 suite executions), compromising the test suite reliability by 16%. When a delay of 500 ms was introduced, this rate escalated to 46% across 23 suite runs.

[14]https://gitlab.com/lsi-ufcg/cytestion/sync-study/execute-study

For greater delays (1000 ms, 1500 ms, and 2000 ms) all suite executions experienced breakage. These results evidence the substantial impact that synchronization issues might have on the reliability of a test suite (RQ_1). Even minor delays can cause significant increases in both individual test cases and overall suite breakages.

3.2 Evaluating Different Waiting Mechanisms

Based on the results of our first study, we selected the 10 test cases most affected by synchronization issues (test cases 1-10). For each of those tests, we created four refactored versions. In each version, we applied a different waiting mechanism (*Static Wait*, *Stable DOM Wait*, *Network Wait*, and *Explicit Wait*) trying to fix the found synchronization issues. Finally, we reran the suite 50 times on the five Sylius instances. The goal of this second study is to identify the best waiting mechanism that could help a tester cope with the synchronization challenge (RQ_2). It is important to highlight that the refactorings were manually applied by the first author in all locations identified as having a synchronization issue in our first study. The refactorings were later revised and confirmed by the third and fourth authors.

3.2.1 Results. We present the results of our second study in Figure 4 that depicts the occurrences of test case and test suite breakages, along with the average test suite execution time. These results are showcased for the five delay settings (Without delay, 500 ms, 1000 ms, 1500 ms, and 2000 ms) and four waiting mechanisms (*Static Wait*, *Stable DOM Wait*, *Network Wait*, and *Explicit Wait*).

For the *Without delay* scenario, only the test artifacts that used *Static Wait* exhibited test case (0.02%) and suite breakages (2.0%). However, for the *500 ms delay*, all waiting mechanisms mitigated all synchronization issues, showing 0.0% of breakage rate for test cases and test suite.

As for the *1000 ms delay* setting, *Static Wait* experienced breakage rates of 16.8% for test cases and 70.0% for test suites. Furthermore, *Stable DOM Wait* showed breakages at 2.0% for test cases and 16.0% for test suites. On the other hand, *Network Wait* and *Explicit Wait* remained with no breakages.

For the *1500 ms delay*, breakages notably increased: 44.2% and 98.0% test case and suite breakage rates, respectively, for *Static Wait*. *Stable DOM Wait* rates increase to 27.0% for test cases and 94.0% for the test suite. On the other hand, *Network Wait* and *Explicit Wait* remained without breakages.

Finally, for the most degraded setting (*2000 ms delay*), breakage rates rose to 60.2% for *Static Wait* and 98% for test cases and test suite, respectively. Similarly, the *Stable DOM Wait* mechanism increased breakage rates to 55.8% for test cases and 100% for the test suite. Again, *Network Wait* and *Explicit Wait* maintained their effectiveness without presenting any breakages.

We conducted a manual investigation of these breakages. For *Static Wait*, most of the breakages occurred because the set waiting time was not sufficient to overcome the synchronization issues. As for the *Stable DOM Wait* breakages, the common found issue was due to the solution inherent assumption of DOM stability while rendering the loading of the executed action. For extended periods, it led to premature execution release and breakages.

To assess breakage rate differences, we applied the Fisher's exact test, along with odds ratios for effect size evaluation [2]. For the scenarios *Without delay* and *500 ms delay*, all mechanisms showed similar performance (p-value = 1). However, at delays of 1000ms and 1500ms, *Network Wait* and *Explicit Wait* outperformed *Stable DOM Wait*, which in turn surpassed *Static Wait* (p-value < 0.05). The same pattern occurred at a *2000 ms* delay, except where *Stable DOM Wait* equaled *Static Wait* (p-value = 0.1784347).

Network Wait and *Explicit Wait* exhibited no breakages during the experiment. However, it is essential to weigh trade-offs. On average, *Network Wait* took slightly longer to execute the test suite compared to *Explicit Wait* across all delay settings, with a range of 65 to 137 seconds. When evaluating waiting mechanisms, it is crucial to consider both breakage rates and execution time, which can vary across projects. The prolonged execution times with *Network Wait* can be attributed to waiting for non-essential requests. Although effective in reducing breakages, it may come at the cost of longer execution times.

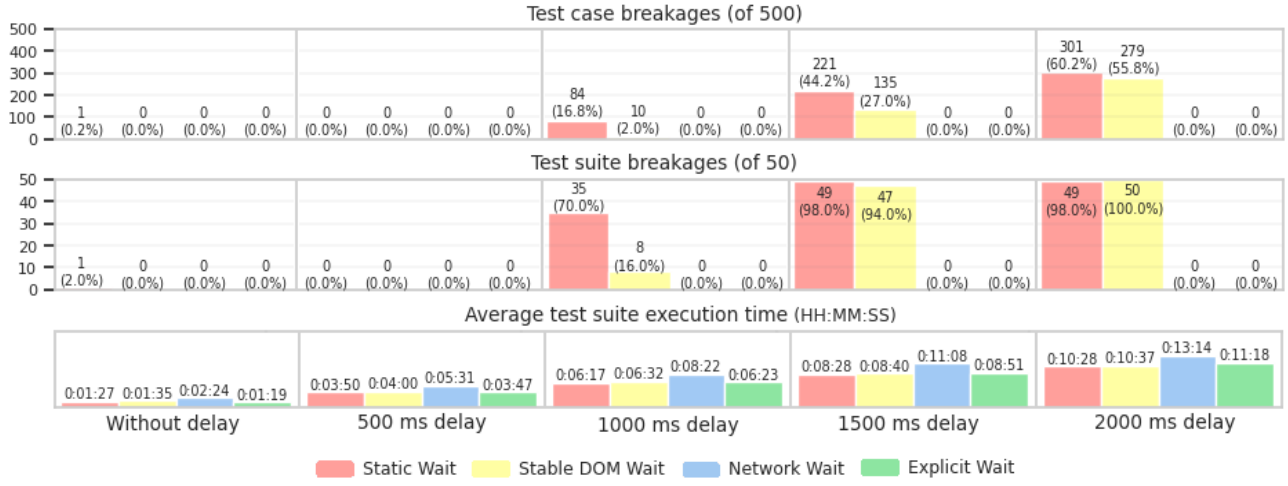

Figure 4: Results from the Sylius study: test case breakages, suite run breakages, and average suite execution time.

We conducted a Mann-Whitney U test for each delay configuration to evaluate execution time differences between *Network Wait* and *Explicit Wait* [2]. In all delay configurations, significant p-values (all $< 10^{-17}$) were observed, indicating substantial differences. Vargha-Delaney effect sizes ranged from -16.28471 to -8.055329, indicating consistently longer execution times for Network Wait compared to Explicit Wait.

These results contribute to answer RQ_2 by highlighting the diverse effectiveness of waiting mechanisms in handling synchronization issues. Notably, both *Network Wait* and *Explicit Wait* demonstrate resilience, exhibiting no breakages across various delay scenarios. However, the observed differences in execution times underscore the significance of a strategic selection tailored to each scenario.

3.3 A Case Study with an Industrial Application

To complement the previous results and assess the impact of synchronization related issues and the use of various waiting mechanisms in a real-world scenario, we conducted a case study involving an industrial application from a partner company. This project is a React-based[15] application designed for managing government auditing processes and features a robust Cypress test suite that runs multiple times a day. It works as a regression suite that the development team runs before any modification is integrated into the main codebase.

Table 1 presents the project metrics, including lines of code, lines of test code, test cases, and waiting points. Waiting points are specific code areas in the test suite where synchronization between the test script and the AUT is necessary. The reported 866 waiting points were identified by the project testing team. It was reported that synchronization issues have caused several practical problems to the team such as release delays and wasted efforts evaluating false bugs. To address this, the team applied a one-second *Static Wait* to each identified waiting point.

Metric	Value
Lines of Code	136,128
Lines of Test Code	5,933
Test Cases	169
Waiting Points	866

Table 1: Metrics for the study object.

The web application operates entirely within a single Docker container using the designated code version. Database, front- and back-end components are preloaded before running the Cypress test suite. This scenario resembles the environment reported in our first studies with Sylius (Sections 3.1 and 3.2).

For this study, we used the following procedure: the existing project suite already uses the *Static Wait* mechanism, which we called the *Static Wait* version. We created three additional versions of this test suite, refactoring all waiting points to apply *Stable DOM Wait*, *Network Wait*, and *Explicit Wait*. Similarly to our previous studies, we executed each version of the suite 50 times within the AUT, and we registered the number of test case breakages, test suite breakages, as well as the average test suite execution time.

3.3.1 Results. Regarding RQ_1, the data obtained from this study reinforces that the synchronization challenge can significantly affect the reliability of a test suite. As shown by the test results in Figure 5, the *Static Wait* version (original version) experienced 70 test case breakages out of 8450 (0.83%). While this percentage might seem relatively low, it becomes concerning when considering that these breakages occurred in 16 different test suite executions, representing 32%. We interviewed the project developers and testers that confirmed the occurrence of these flaky tests on a daily basis and that their common practice to deal with that issue is to re-execute the test suite, which often is found as very costly. This underscores the impact of synchronization challenges on the reliability of the test suite.

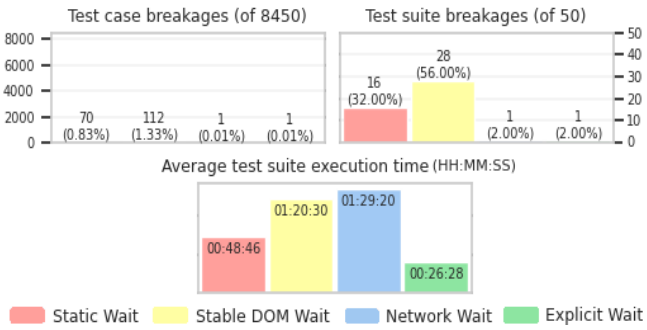

Figure 5: Case study results, including test cases breakages, suite run breakages and average test suite execution time.

Regarding RQ_2, while both the *Network Wait* and *Explicit Wait* versions resulted in identical results for test case and test suite breakages, significant differences emerged in average execution times. *Explicit Wait* outperformed with an average test suite execution time of 00:26:28, surpassing even the *Static Wait* version (00:48:26), while *Network Wait* exhibited the longest average execution time at 01:29:20.

However, it is crucial to consider the implementation complexities associated with these mechanisms. While implementing the *Network Wait* involved a simple replace operation, implementing the *Explicit Wait* was a very complex task. This complexity stemmed from the necessity to establish suitable wait conditions, requiring coordinated efforts from multiple authors over approximately four hours to address all 866 waiting points. Such complexities should be factored into the decision-making process when selecting the preferred mechanism.

Although the results regarding breakage rates were similar in our case study and in the Sylius one, they differed for execution time. We attribute this variation to the inherent nature of the *Network Wait* mechanism, which waits for all requests, including unnecessary ones, to be fulfilled before proceeding with the test execution. This difference may be attributed to the industrial case study inherently having more network requests than Sylius, influencing the overall execution time. In contrast, *Explicit Wait* sets conditions that indirectly require minimal or no requests before a specific element becomes explicitly available on the GUI.

[15] https://react.dev

Finally, we analyzed the two breakages that remained for both *Network Wait* and *Explicit Wait*. We observed that those breakages refer to waiting points that were not noticed/identified by the project testing team. They only became apparent after undergoing 100 test suite executions. We later included those points in the refactored versions of *Network Wait* and *Explicit Wait* and no breakages were found. This revelation was attributed to a temporary limitation of hardware resources during the execution, exposing a waiting point that went unnoticed because it normally loads quickly.

3.4 Learned Lessons

Taking into account the results of our studies, we can answer RQ_1 and RQ_2:

> ***RQ_1:*** *Synchronization issues can significantly impact a test suite reliability. For some scenarios, we found that up to 32% of a suite can fail due to those issues.*

> ***RQ_2:*** *Network Wait and Explicit Wait were the most effective mechanisms, with equal breakage rates for test cases and suites, but Explicit Wait had better execution times.*

We believe that the results achieved can be valuable for testers implementing E2E Cypress suite. They evidence the importance of caution when writing test scenarios involving asynchronous calls. By understanding the common pitfalls and sources of flakiness, testers can act proactively to reduce breakages, leading to more robust and reliable tests. Moreover, by knowing how to apply the different waiting mechanisms in Cypress scripts, testers can have the proper tools to deal with such challenges and fine-tune their test cases.

Explicit Wait is a well-known mechanism for Selenium suites. Our findings showed its effectiveness for Cypress suites as well. However, it is important to weigh the associated complexity of implementing it, requiring a thorough understanding of subsequent testing actions and conditions, which can be error-prone and affect the overall script efficiency.

Testers can identify waiting points reactively or proactively. Reactively, after a breakage, testers can rerun the test with Cypress's GUI, monitoring the logging to pinpoint and address synchronization problems. On the other hand, when working proactively, testers can identify waiting points before they cause breakages, especially for elements triggering asynchronous events like select input and dropdown interactions. Incorporating waiting mechanisms in relevant test cases can preempt potential issues.

We believe our results can also benefit tool builders and researches. New tools can integrate effective waiting mechanisms like *Network Wait* and *Explicit Wait*, providing testers with more reliable synchronization solutions. The new *Network Wait* mechanism enhances software testing with its generic, cost-effective approach, offering opportunities for further exploration by tool makers. For instance, scriptless E2E testing often deals with nondeterministic scenarios where the goal is to navigate through the application in search of visible failures [22]. Traditional waiting mechanisms often fall short due to the difficulty in determining the appropriate wait conditions. *Network Wait* addresses this challenge, providing a solution that does not require the definition of a specific condition. Additionally, researchers can use our studies to validate existing techniques and develop advanced synchronization algorithms and tools for automated testing practices, resulting in more robust solutions for real-world applications.

4 THREATS TO VALIDITY

In terms of external validity, our results cannot be generalized beyond the context of the specific projects used in our study. Since we based our study on two web applications and their test suites, one open-source and one industrial, we acknowledge that the sample lacks diversity. Nevertheless, given the substantial size of these systems and the number of test cases they encompass, we argue that they serve as good representatives of the broader web application and E2E testing domain. Furthermore, in the Sylius study, we utilize a test suite generated by students, whereas the second study employs a test suite developed by a team of professionals. Despite their differences, both suites include synchronization issues, indicating that tests with synchronization issues are not solely attributable to lack of experience and can manifest in any E2E test suite.

In terms of internal validity, we utilized the Cypress default settings to configure command timeout. This implies that our evaluations of waiting mechanisms can be seen as a blend of implicit and other waits. We acknowledge that different configurations for those settings could impact our results. Nevertheless, these settings are intrinsic to the framework's API that requires some value assignment. We opted to use Cypress default settings to resemble how most testers use it. Furthermore, our empirical study involved two manual actions by the first author, potentially threatening internal validity: the manual validation and selection of tests, as well as the subsequent manual refactoring to incorporate waiting mechanisms. To mitigate these threats, the selected tests and refactored code were reviewed and validated by the third and fifth authors.

In terms of conclusion validity, the use of external libraries, such as *cypress-wait-for-stable-dom* and *cypress-wait-until*, poses a potential validity threat to our study conclusions due to their impact on our results. We acknowledge that errors or limitations in these libraries could influence our findings, and the validity of our conclusions depends on this factor. To address this concern, we meticulously reviewed the documentation and source code of these libraries and performed additional validation testing.

In terms of construct validity, we compared *Static Wait* (one-second delay) and *Stable DOM Wait* (one-second interval). While different values could produce different outcomes, we chose one-second delays because of their common usage in the projects we studied. However, this choice might have affected our findings, and varying delay times could influence our conclusions.

5 RELATED WORK

Synchronization issues have been investigated in the context of automated web testing. Lam et al. [17] examined the life cycle of flaky tests across six extensive Microsoft projects and highlighted their detrimental effects. Their analysis revealed that flaky tests are most commonly associated with asynchronous calls lacking proper waiting.

Nass et al. [25] conducted a systematic literature review that identified key-challenges in GUI-based test automation. They classified these challenges as essential or accidental and mentioned them as an open research topic. The synchronization challenge is categorized as an accidental one.

Sousa et al. [37] analyzed causes and fixing strategies for flaky tests in automated GUI tests across 24 open-source projects. They found that race conditions account for 60% of flaky test causes. Among the fixing strategies, the addition of waiting mechanisms was the most prevalent, appearing in 53% of the related commits.

Garcia et al. [12] extensively analyze Selenium, particularly its waiting strategies. They advocate for design patterns like Page Objects and the Screenplay pattern to enhance test reusability and readability. The study emphasizes reducing maintenance efforts through efficient waiting strategies and highlights challenges such as test flakiness and troubleshooting.

Habchi et al. [13] conducted a systematic literature review and interviewed 20 software practitioners to explore flaky tests in software testing ecosystems. They report synchronization points in GUI tests as a significant cause of flakiness. The authors also emphasize the importance of implementing mitigation measures, enhancing testing infrastructure, and establishing testing practice guidelines.

Leotta et al. [19] conducted a survey with 78 industry experts. They found that synchronization issues (referred to as asynchronous) are widely recognized as a challenge for E2E testing with Selenium. Moreover, they found that testers frequently apply waiting strategies to address it. However, these strategies, while sometimes effective, can increase execution time and introduce testing flakiness. The authors emphasize the importance of choosing an appropriate waiting strategy and advocate for further research into developing more effective waiting strategies and tools.

Pei et al. [31, 32] conducted an empirical study on Async Wait flakiness in front-end testing. The study identified time-based and DOM-related issues as primary concerns. They recommended synchronizing on DOM elements rather than using explicit time-based mechanisms to reduce flakiness and enhance test reliability. Additionally, Pei et al. introduced TRaf, an automated method for mitigating flaky tests by recommending suitable waiting times for asynchronous calls based on code similarity and change history.

Feng et al. [9] present AdaT, a lightweight approach for accelerating Android GUI testing by dynamically adjusting event timing based on GUI rendering status. By leveraging deep learning and real-time GUI streaming, AdaT accurately infers rendering states and synchronizes testing events, achieving efficiency and effectiveness improvements in automated testing.

Olianas et al. [26, 27] focus on reducing flakiness in E2E testing with Selenium by replacing static waits with explicit waits. In the first paper, they provide a procedure to eliminate flakiness, while the second introduces a tool-based approach for refactoring test suites to minimize the usage of thread sleeps, reducing execution time, and improving test suite reliability.

Liu et al. [20] present WEFix, an automated tool that inserts explicit waits to address flaky tests in web e2e testing. By analyzing browser-side DOM mutations, WEFix generates wait oracles to predict client-side operations, effectively reducing flakiness and runtime overhead.

These studies highlight significant synchronization issues affecting test reliability and system quality. Notably, only the work of Liu et al. has explored this challenge within the Cypress context, focusing solely on applying the *Explicit Wait* mechanism. Our work goes further by compiling a catalog of waiting mechanisms for Cypress and introducing the novel *Network Wait*. This new mechanism is unique due to Cypress's ability to intercept requests, a feature not present in Selenium. Traditional waiting mechanisms focus on DOM-related events and miss server-side interactions or asynchronous actions post-page load. Therefore, existing methodologies cannot replicate *Network Wait*'s unique functionality. Additionally, our empirical studies evidence the impact of synchronization issues in Cypress suites and may guide testers in selecting the most adequate fixing strategies.

6 CONCLUDING REMARKS

In this paper, we delve into the synchronization challenge in Cypress tests, a critical issue for automated E2E testing. Synchronization issues can cause test cases to become flaky and challenge the reliability of a test suite. We compiled a catalog of waiting mechanisms reported in the literature and discussed how they can be applied in Cypress tests. We believe our catalog is the first in this context and that it can significantly assist Cypress testers in gaining a better understanding of synchronization issues and how to address them. Additionally, we introduced a new mechanism, *Network Wait*, and provided its implementation for testing community as an external Cypress dependency. Within days of publication, without any advertising, this dependency was downloaded almost 2000 times[*], underscoring its practical significance.

We also conducted three empirical studies that demonstrated the great impact that synchronization-related issues can have on different test suites. Finally, we compared the effectiveness of four waiting mechanisms to deal with the issues. The *Explicit Wait* and *Network Wait* yielded the best results. Although the first is faster, it requires the tester to have a precise understanding of subsequent testing actions and conditions to avoid synchronization issues. The *Network Wait* emerges as a valuable option, as it monitors server requests to ensure correct test execution. This proposed approach proved to be effective.

Regarding future work, we plan to: i) expand our empirical studies by including a larger number of open-source projects and industrial case studies; ii) assess how our results can be related to and/or adapted for scriptless E2E testing [22]; iii) create guidelines on how testers can prevent synchronization issues in Cypress tests; and iv) develop an IDE plug-in that detects synchronization issues in legacy test suites and automatically addresses them.

REFERENCES

[1] Nauman Bin Ali, Emelie Engström, Masoumeh Taromirad, Mohammad Reza Mousavi, Nasir Mehmood Minhas, Daniel Helgesson, Sebastian Kunze, and Mahsa Varshosaz. 2019. On the search for industry-relevant regression testing research. *Empirical Software Engineering* 24 (2019), 2020–2055.
[2] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.
[3] Satya Avasarala. 2014. *Selenium WebDriver practical guide.* PACKT publishing.

[*] Data collected in June 2024.

[4] Axel Bons, Beatriz Marín, Pekka Aho, and Tanja EJ Vos. 2023. Scripted and scriptless GUI testing for web applications: An industrial case. *Information and Software Technology* 158 (2023), 107172.

[5] Maura Cerioli, Maurizio Leotta, and Filippo Ricca. 2020. What 5 million job advertisements tell us about testing: a preliminary empirical investigation. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 1586–1594.

[6] Diego Clerissi, Maurizio Leotta, Gianna Reggio, and Filippo Ricca. 2017. Towards the generation of end-to-end web test scripts from requirements specifications. In *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*. IEEE, 343–350.

[7] Cypress.io. 2018/2023. Key Differences. https://docs.cypress.io/guides/overview/key-differences. Accessed: 10-10-2023.

[8] Edward Dunn Ekelund and Emelie Engström. 2015. Efficient regression testing based on test history: An industrial evaluation. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 449–457.

[9] Sidong Feng, Mulong Xie, and Chunyang Chen. 2023. Efficiency Matters: Speeding Up Automated Testing with GUI Rendering Inference. https://doi.org/10.1109/icse48619.2023.00084

[10] Cypress framework. 2018/2023. Why Cypress? https://docs.cypress.io/guides/overview/why-cypress . Accessed: 10-10-2023.

[11] Boni Garcia. 2022. *Hands-On Selenium WebDriver with Java.* " O'Reilly Media, Inc.".

[12] Boni García, Micael Gallego, Francisco Gortázar, and Mario Munoz-Organero. 2020. A survey of the selenium ecosystem. *Electronics* 9, 7 (2020), 1067.

[13] Sarra Habchi, Guillaume Haben, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2022. A qualitative study on the sources, impacts, and mitigation strategies of flaky tests. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 244–255.

[14] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. 2016. WATERFALL: an incremental approach for repairing record-replay tests of web applications. https://doi.org/10.1145/2950290.2950294

[15] Mouna Hammoudi, Gregg Rothermel, and Paolo Tonella. 2016. Why do Record/Replay Tests of Web Applications Break? https://doi.org/10.1109/icst.2016.16

[16] Md Hossain, Hyunsook Do, and Ravi Eda. 2014. Regression testing for web applications using reusable constraint values. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 312–321.

[17] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1471–1482.

[18] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2016. Approaches and tools for automated end-to-end web testing. In *Advances in Computers*. Vol. 101. Elsevier, 193–237.

[19] Maurizio Leotta, Boni García, Filippo Ricca, and Jim Whitehead. 2023. Challenges of end-to-end testing with selenium WebDriver and how to face them: A survey. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 339–350.

[20] Xinyue Liu, Zihe Song, Weike Fang, Wei Yang, and Weihang Wang. 2024. WEFix: Intelligent Automatic Generation of Explicit Waits for Efficient Web End-to-End Flaky Tests. *arXiv preprint arXiv:2402.09745* (2024).

[21] Fatini Mobaraya, Shahid Ali, et al. 2019. Technical Analysis of Selenium and Cypress as Functional Automation Framework for Modern Web Application Testing. *Department of Information Technology, AGI Institute, Auckland, New Zealand* (2019).

[22] Thiago Santos de Moura, Everton LG Alves, Hugo Feitosa de Figueirêdo, and Cláudio de Souza Baptista. 2023. Cytestion: Automated GUI Testing for Web Applications. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. 388–397.

[23] Waweru Mwaura. 2021. *End-to-End Web Testing with Cypress: Explore techniques for automated frontend web testing with Cypress and JavaScript.* Packt Publishing Ltd.

[24] Takao Nakagawa, Kazuki Munakata, and Koji Yamamoto. 2019. Applying modified code entity-based regression test selection for manual end-To-end testing of commercial web applications. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 1–6.

[25] Michel Nass, Emil Alégroth, and Robert Feldt. 2021. Why many challenges with GUI test automation (will) remain. *Information and Software Technology* 138 (2021), 106625.

[26] Dario Olianas, Maurizio Leotta, and Filippo Ricca. 2022. SleepReplacer: A novel tool-based approach for replacing thread sleeps in selenium webdriver test code. *Software Quality Journal* 30, 4 (2022), 1089–1121.

[27] Dario Olianas, Maurizio Leotta, Filippo Ricca, and Luca Villa. 2021. Reducing flakiness in End-to-End test suites: An experience report. In *International Conference on the Quality of Information and Communications Technology*. Springer, 3–17.

[28] Ana CR Paiva, Nuno H Flores, João P Faria, and José MG Marques. 2018. End-to-end automatic business process validation. *Procedia Computer Science* 130 (2018), 999–1004.

[29] Narayanan Palani. 2021. *Automated Software Testing with Cypress.* CRC Press.

[30] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–74.

[31] Yu Pei, Sarra Habchi, Renaud Rwemalika, Jeongju Sohn, and Mike Papadakis. 2022. An empirical study of async wait flakiness in front-end testing.. In *BENEVOL*.

[32] Yu Pei, Jeongju Sohn, Sarra Habchi, and Mike Papadakis. 2023. TRaf: Time-based Repair for Asynchronous Wait Flaky Tests in Web Testing. *arXiv preprint arXiv:2305.08592* (2023).

[33] Kai Presler-Marshall, Eric Horton, Sarah Heckman, and Kathryn Stolee. 2019. Wait, wait. no, tell me. analyzing selenium configuration effects on test flakiness. In *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*. IEEE, 7–13.

[34] Sujay Raghavendra. 2021. *Python Testing with Selenium: Learn to Implement Different Testing Techniques Using the Selenium WebDriver.* Springer.

[35] Filippo Ricca, Maurizio Leotta, and Andrea Stocco. 2019. Three Open Problems in the Context of E2E Web Testing and a Vision: NEONATE. , 89–133 pages. https://doi.org/10.1016/bs.adcom.2018.10.005

[36] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An empirical analysis of UI-based flaky tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1585–1597.

[37] Érica Sousa, Carla Bezerra, and Ivan Machado. 2023. Flaky Tests in UI: Understanding Causes and Applying Correction Strategies. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. 398–406.

[38] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual web test repair. https://doi.org/10.1145/3236024.3236063

[39] Tanja EJ Vos, Pekka Aho, Fernando Pastor Ricos, Olivia Rodriguez-Valdes, and Ad Mulders. 2021. testar–scriptless testing through graphical user interface. *Software Testing, Verification and Reliability* 31, 3 (2021), e1771.

[40] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. 1–10.