Non-Flaky and Nearly-Optimal Time-based Treatment of Asynchronous Wait Web Tests

YU PEI, SnT, University of Luxembourg, Luxembourg
JEONGJU SOHN*, Kyungpook National University, Korea
SARRA HABCHI, Ubisoft, Canada
MIKE PAPADAKIS, SnT, University of Luxembourg, Luxembourg

Asynchronous waits are a common root cause of flaky tests and a major time-influential factor of web application testing. We build a dataset of 49 reproducible asynchronous wait flaky tests and their fixes from 26 open-source projects to study their characteristics in web testing. Our study reveals that developers adjusted wait time to address asynchronous wait flakiness in about 63% of cases (31 out of 49), even when the underlying causes lie elsewhere. From this, we introduce *TRaf*, an automated time-based repair for asynchronous wait flakiness in web applications. *TRaf* determines appropriate wait times for asynchronous calls in web applications by analyzing code similarity and past change history. Its key insight is that efficient wait times can be inferred from the current or past codebase since developers tend to repeat similar mistakes. Our analysis shows that *TRaf* can statically suggest a shorter wait time to alleviate async wait flakiness immediately upon the detection, reducing test execution time by 11.1% compared to the timeout values initially chosen by developers. With optional dynamic tuning, *TRaf* can reduce the execution time by 16.8% in its initial refinement compared to developer-written patches and by 6.2% compared to the post-refinements of these original patches. Overall, we sent 16 pull requests from our dataset, each fixing one test, to the developers. So far, three have been accepted by the developers.

Additional Key Words and Phrases: Flaky Test Failure, Automatic Program Repair, Web Front-end Tests

1 INTRODUCTION

Regression testing ensures the system under test (SUT) behaves as expected when software is changed: a test failure implies that the latest changes introduce faults into the SUT [13, 15, 49]. In practice, however, some tests fail intermittently during regression testing, even in the absence of fault-introducing changes and sometimes on the same version of the system. These tests, called flaky tests, force developers to waste their effort debugging them, only to discover later that they are false alarms. This causes developers to overlook even the true failure alarms due to their past experiences with flaky tests.

Flaky tests are quite common in modern software systems [9, 15, 19, 22, 27, 39]. Labuschagne et al. showed that 13% of failed test executions in open-source projects using Travis CI were flaky [15]. At Google, despite the persistent effort to remove flaky tests, around 16% of the tests still exhibit a certain flakiness degree, and 84% of transitions between Pass and Fail were caused by flaky tests, indicating the importance of flakiness issues in CI

*Corresponding author

This work is supported by the Luxembourg National Research Funds (FNR) through the CORE project grant C20/IS/14761415/TestFlakes.
Authors' addresses: Yu Pei, yu.pei@uni.lu, SnT, University of Luxembourg, Luxembourg; Jeongju Sohn, jeongju.sohn@knu.ac.kr, Kyungpook National University, Korea; Sarra Habchi, sarra.habchi@ubisoft.com, Ubisoft, Canada; Mike Papadakis, michail.papadakis@uni.lu, SnT, University of Luxembourg, Luxembourg.

ACM 1557-7392/2024/9-ART
https://doi.org/10.1145/3695989

environment [22, 31]. While flaky tests appear inevitable, manually addressing them is infeasible because of their sheer number and ambiguous nature. Hence, many studies have focused on automatically resolving them, either entirely removing root causes or reducing the chance of being flaky [3, 6, 10, 17, 25, 42]. As various factors (i.e., root causes) can incur flaky tests, the proposed repair techniques often focus on a single category of flaky tests caused by a specific root cause rather than attempting to cover different categories of flaky tests.

Asynchronous call/wait is one of the most common causes of flaky tests [17, 27, 39]. Nevertheless, compared to other categories of flakiness, such as order-dependency, not much work has been dedicated to automatically repairing this specific category of flaky tests. Recently, Lam et al. introduced FaTB, which stands for Flakiness and Time Balancer, that refines developer-written fixes for flaky tests caused by asynchronous calls/waits (i.e., async wait flaky tests) [17]; the objective of the refinement is to reduce test execution time while retaining the same flake rate as the original fixes. Their study on six large-scale proprietary projects in Microsoft showed the prevalence of async wait flaky tests and the developers' tendency to address them simply by increasing the wait time. Based on these observations, the authors conducted a small study on the effectiveness of FaTB in refining the wait time in developer fixes to balance between the decreased flake rate and increased test execution time; this study demonstrates the potential benefits of *collective reduction in test execution time by using more efficient time values for waiting in the async wait flaky tests*. Prior studies in web testing revealed that web developers often employ large wait times to mitigate the flakiness caused by asynchronous calls [32, 35]. This paper also proposes a technique that assigns a wait time to handle the async wait flakiness, but without any involvement of developers, aiming to completely automate the repair that can be applied as soon as the detection of flakiness.

Testing in web environments is particularly challenging as testing goes beyond unit and integration tests, including front-end testing. Web front-end testing often interacts with users through the graphical user interface (GUI). Due to the complexity of user interactions, network instability, and the inherently asynchronous nature of web applications, web front-end testing is more subject to flaky tests than traditional unit testing. Yet, the research community paid less attention to flaky tests in web front-end testing than traditional unit tests [12].

In this paper, we investigate flaky tests in web front-end testing, mainly those flaked by asynchronous waiting between calls (i.e., async wait flaky tests). To get a general idea of how async wait flaky tests have been handled by developers in this specific test domain, we conduct a study focused on the practice of developers to resolve async wait flaky tests in web front-end testing. We study 49 async wait flakiness from 26 web projects, examining the flakiness in test code and their corresponding developer-written repairs. From this study, we find that in approximately 63% of the cases (31 out of 49), developers repaired async wait for flaky tests by adding or increasing the time to wait, mostly concurring with the previous findings on the fixes in traditional testing [12, 17, 27, 28, 33, 39].

Based on the findings of our preliminary study, we propose *TRaf*, short for Time-based Repair of Async wait Flaky tests, a new repair approach that automatically addresses async wait flaky tests in web front-end testing. *TRaf* uses code similarity and past change history to find the probable wait time to repair a given flaky test. Our basic idea is that fixing ingredients for the repair, in our case, new time values, exists in the past or current codebase since developers are more likely to make similar mistakes concerning similar functionality and test behaviors. *TRaf*, by default, statically suggests a set of potential wait times and does not involve any test run until validating the generated repair with the suggested time value, maximizing its usability; it allows users to dynamically tune the new wait time further to obtain the optimal time between the original flaky time and the newly proposed one. *TRaf* is practical and lightweight; it identifies better wait time value candidates without any interpretation by developers. Hence, it can be used as hotfixes to ensure continuous and timely software delivery.

Experimental results with 31 Async Wait tests originally repaired by adding or increasing the timeout (out of 49 studied tests) show that *TRaf* can propose an efficient wait time that can balance between two conflicting goals: *reducing the test execution time* and *relieving test flakiness*. Without and with the additional dynamic optimization, *TRaf* saves the test execution time an average of 11.1% and 20.2%, respectively, compared to developer-fixed

tests while alleviating the test flakiness. We sent 16 pull requests to the developers, each fixing one test. So far, three pull requests have been accepted by the developers, six are pending, and seven have been rejected. These results demonstrate that our approach can reduce the flakiness of tests by proposing an appropriate waiting time, thereby reducing the execution time. Of the seven rejected pull requests, the developers mostly acknowledged the flakiness and chose to reduce the problem custom to the projects. Developers tend to prioritize stability, opting for large wait time fixes as long as they are sufficient to ensure continuous delivery of stable applications, especially projects with high complexity and large scale.

In summary, the main contributions of this paper are:

- We derive a reproducible dataset of 26 JavaScript open-source web projects with 49 tests exhibiting async wait flakiness.
- We manually inspect common fixing practices for tests exhibiting async wait flakiness in web front-end testing and confirm that adding or increasing the wait time, used in 31 of the 49 tests studied, is the most common practice by developers.
- We propose *TRaf*, a new repair technique for async wait flaky tests in web front-end testing. *TRaf* statically and efficiently finds an adequate wait time using the code similarity and version history of the code under testing.
- Experimental results show that *TRaf* can statically find a waiting time that can alleviate the flakiness while saving an average of 11.1% in the test execution time compared to using time values selected by developers. The additional dynamic tunning further saves the execution time by 16.8% from the first successful refinement (within two trials) and by 20.2% when tuning to the end. Compared to an existing post-refinement of developer-written patches, it reduces the execution time by 6.2%.

The remainder of the paper is organized as follows: Section 2 explains the background of our approach. Section 3 presents the results of our preliminary study, and Section 4 describes the methodology of *TRaf* in detail. Section 5 presents our research questions and the experiments, which are evaluated in Section 6. Section 7 discusses the potential impact of our findings in practice. Section 8 and Section 9 state the threats and related work. We conclude our paper in Section 10.

2 ASYNC WAIT FLAKY TESTS IN WEB FRONT-END TESTING

Continuity of program execution during a call process does not guarantee that the result will be accessible when needed. For instance, a program may request a resource that is not fully rendered or attempt to process data before the arrival due to the network latency. A common approach to deal with such cases is to wait until the call/action is completed and then proceed with subsequent actions. However, because of the inherently unpredictable nature of web and the diversity of web resources, the web access load is difficult to estimate and likely varies on each call [24, 44, 48]. This is when asynchronous wait flakiness incurs, causing a test to pass and fail non-deterministically depending on the environment/circumstance of a given time.

Asynchronous (Async) wait flaky test failures typically occur in the following scenario: a test interacts with the DOM element, waits for resources to be rendered or loaded, and performs subsequent actions, such as transition and animation [39]. Let us consider an example in Figure 1a, which describes a test trying to retrieve an element before it is fully rendered (at Line 04). For this test to consistently pass, the *d3-flame-graph* element should be guaranteed to be accessible before the retrieval; otherwise, the test will fail non-deterministically because of asynchronous call to the *d3-flame-graph* element.

In web front-end testing, *DOM* and *Time* are the primary concerns of developers and testers since these two directly influence how web pages display and load [7, 23]. Indeed, the recent work on UI-based flaky tests in web testing shows that async wait flaky tests, which account for approximately 45% of the studied flaky tests, are

often caused by the timing differences or the availability of UI components to interact [39].[1] The authors further demonstrated that many of these async wait flaky test failures can be resolved simply by adding or increasing delays. Motivated by such observations, although other factors, such as network or API requests, may incur asynchronous wait in web testing, our preliminary study in Section 3 mainly focuses on the role of time and DOM issues in triggering and addressing asynchronous wait flaky test failures.

```
01.   it('test flamegraph shows when valid profile data exists', async () => {
02.     await page.$eval('[data-refresh-button]', elem => elem.click());
03. +   await page.waitForSelector('.d3-flame-graph');
04.     const flamegraphElement = await page.$eval(
05.       '.d3-flame-graph',
06.       elem => elem.textContent,
07.     );
```

(a) Dom-caused/Dom-based fix

```
01.     await componentAct(async () => {
02.       errStream.error("controlled error")
03. -     await wait(0)
04. +     await wait(50)
05.     })
06.
07.     expect(screen.queryByText("Loading...")).toBeNull()
```

(b) Time-caused/Time-based fix

```
01. test('opens a dialog with multiple results', async () => {
02.     ...
03.     fireEvent.mouseDown(input)
04.     fireEvent.change(input, { target: { value: 'seg02' } })
05.     fireEvent.keyDown(auto, { key: 'Enter', code: 'Enter' })
06. +   await screen.findByText('Search results', {}, { timeout: 10000 })
07.     expect(state.session.views[0].searchResults.length).toBeGreaterThan(0)
08. })
```

(c) Dom-caused/Time-based fix

Fig. 1. Cases from our study include cause and fix categories. Figure 1a shows the case caused by interaction between the DOM and fixed by a Dom-based fix strategy, which fixes the synchronization point to a specific DOM element. Figure 1b shows the case caused by an insufficient wait time and fixed by a Time-based fix strategy, i.e., increases the wait time. Figure 1c shows a Dom-caused case fixed by a Time-based fix strategy, i.e., add a timeout to the asynchronous call to a DOM element.

3 PRELIMINARY STUDY

This section explores how developers introduce and handle async wait flaky tests in web front-end testing, mainly concerning the roles of the DOM and time issues.

[1]The network delays or the improper scheduling often result in interacting with elements that have not been fully loaded, resulting in flaky test failures.

3.1 Data Collection

The asynchronous wait mechanism (i.e., async wait flaky tests) in web applications includes various asynchronous events, such as user interaction with the web interface [39]. To collect a set of async wait flaky tests, we followed a procedure similar to that of Gruber et al., searching for commits that fix flakiness in open-source projects on GitHub with certain keywords [9]. We then rerun the tests in these commits a fixed number of times, checking for the inconsistency in the test state across the runs.

Specifically, we first looked for the keywords *"e2e"*, *"flaky"*, *"flakiness"*, and *"Intermittent"* in commit messages through GitHub search, obtaining over 800 distinct repositories related to these keywords. We limit the search to repositories mainly written in JavaScript, one of the most common web development languages, narrowing our scope of web front-end testing. We further filtered out the commits using the keywords *"async"*, *"wait"*, *"timeout"*, *"DOM"*, and *"delay"* to ensure them to be related to async wait flakiness. We then manually inspected the remaining commits that modified the test code[2] and identified those that (could) fix the async wait flakiness in the test code. We eventually found around 300 projects with commits potentially fixing async wait flakiness.

To obtain a reproducible set of async wait flaky tests, we cloned the projects from GitHub and reran relevant tests modified in the resulting commits from the previous step. In practice, developers often run a test several times to ensure that it is not flaky. Similarly, many previous researches have demonstrated that 100 reruns are typically enough to determine whether a test is flaky or not [4, 5, 16, 17, 34]. To capture the flakiness, we executed each candidate test 100 times and labeled it as flaky if its pass/fail outcome changed in any of its executions; we saved the error messages for later analysis. Most of the 300 projects with flaky commits could not be executed in our local environment due to dependency issues and inconsistencies between operating systems. For many of those for which we successfully recreated the testing environment, we failed to reproduce the flakiness reported by developers within 100 test reruns; the failures might be attributed to the difficulty of reproducing and some unknown issues. We ended up with 26 projects, as shown in Table 1. With the manual inspection of test outcomes and corresponding error messages, we eventually acquired 49 reproducible flaky tests among 1,869 suspicious flakiness-related commits across 26 web application projects.

3.2 Categorization

We manually examined the collected async wait flaky tests and their fixes, mainly in how they relate to the *Time* and *DOM* factors. We define four categories, two in terms of their causes and two in terms of how they were addressed by developers, by inspecting the related commits, test code, and error messages. The manual inspection and categorization took roughly 40 working days to complete.

3.2.1 Categorization by Different Causes. Asynchronous (Async) wait flaky tests refer to the tests that fail non-deterministically by the mismatch between calls at the synchronization point or the absence of it, which often leads to interacting with elements or resources from the previous call before they are completed. We thus categorize the async wait flaky tests based on the type of call at the (synchronization) point where the output flakes, specifically depending on whether the call is related to *DOM* or the *Time*.

Dom-caused. If a test contains an explicit call to a DOM element, such as `waitForElements` or `findByText`, at the place where flaky behavior occurred, we call this test *DOM-caused*. Figures 1a and 1c present examples of async wait flaky tests categorized as DOM-caused. In Figure 1a, the absence of a synchronization point at Line 03 resulted in accessing the *d3-flame-graph* element before it was completely generated. This code snippet is from an async wait flaky test we examined in the Shopify-theme-inspector; by rerunning this test, we got an error message, ``Failed to find an element matching the selector d3-flame-graph''. Similarly,

[2]We aim to address the flakiness caused by asynchronous waiting between calls in tests. Thus, we excluded the commits where only source code was modified

Table 1. Subjects. $\#_{all}$ and $\#^{related}_{flake}$ are the number of total commits and flakiness-inducing commits. $\#^{rerun}_{flake}$, $\#^{dom}_{cause/fix}$ and $\#^{time}_{cause/fix}$ are the number of reproduced flaky tests and the number of these flaky test caused/fixed by DOM-based and Time-based factors. The final number of flaky tests for the preliminary study is highlighted in dark gray.

project	Commits			Cause		Repair	
	$\#_{all}$	$\#^{related}_{flake}$	$\#^{rerun}_{flake}$	$\#^{dom}_{cause}$	$\#^{time}_{cause}$	$\#^{dom}_{fix}$	$\#^{time}_{fix}$ ($\#^{dom}_{cause}/\#^{time}_{cause}$)
jbrowse-components	8,434	79	6	6	0	0	6 (6/0)
elem-ing-software	579	75	5	5	0	5	0 (0/0)
react-uploady	668	21	4	0	4	0	4 (0/4)
flashmap-production	7,325	105	4	4	0	0	4 (4/0)
react-rxjs	387	12	3	0	3	0	3 (0/3)
puppeteer-core-controller	516	77	2	0	2	0	2 (0/2)
predictive-text-studio	1,198	22	2	0	2	0	2 (0/2)
keptn	8,273	189	2	2	0	1	1 (1/0)
dom-testing-extended	171	24	2	2	0	2	0 (0/0)
preact-devtools	1,416	43	2	2	0	1	1 (1/0)
coinscan-front	184	11	2	2	0	1	1 (1/0)
material-ui	21,337	331	1	1	0	0	1 (1/0)
dotcom-rendering	24,129	238	1	1	0	0	1 (1/0)
codacollection-e2e	16	3	1	1	0	0	1 (1/0)
elm-select	515	17	1	1	0	0	1 (1/0)
eui	5,018	70	1	1	0	1	0 (0/0)
edelweiss-ui	4,007	68	1	0	1	0	1 (0/1)
racp	1,979	129	1	1	0	1	0 (0/0)
wonder-blocks	1,495	53	1	1	0	1	0 (0/0)
client	13,444	10	1	1	0	0	1 (1/0)
liquid	1,191	71	1	1	0	1	0 (0/0)
shopify-theme-inspector	216	2	1	1	0	1	0 (0/0)
azure2jira	20	5	1	1	0	1	0 (0/0)
beacons	6,638	136	1	1	0	1	0 (0/0)
js-libp2p	1,324	64	1	1	0	1	0 (0/0)
cordless	138	14	1	0	1	0	1 (0/1)
total	110,618	1,869	**49**	**36**	**13**	**18**	**31 (18/13)**

in Figure 1c, the test flaked at Line 06 where the results were accessed before fully loaded. In our dataset, 36 out of 49 tests are categorized as *DOM-caused*, as shown in $\#^{dom}_{cause}$ column in Table 1.

Time-caused. Compared to the tests categorized as *DOM-caused*, where many tests are flaky due to the absence of a synchronization point, a test belonging to the Time-caused category often contains an explicit call for synchronization where it waits for a certain amount of time. In Figure 1b, the test flakiness was caused by insufficient waiting time during the execution of the element action and function statements (Line 04). We observed 13 tests flaked by the timing issues out of the 49 async wait tests we studied ($\#^{time}_{cause}$ in Table 1).

3.2.2 Categorization by Developer Fixes. We noted that developers typically increase or add a timeout to the test or modify it to wait for a specific DOM element to be available to resolve flakiness. Hence, we define two repair strategies for async wait flakiness, *Time-based* and *DOM-based*, similar to what we did for the cause.

DOM-based fixes Implementing a `waitFor` with a DOM condition is a common way to fix the async wait flakiness of a test. This type of fix, i.e., *DOM-based*, covers different DOM-related conditions, such as presence, visibility, and availability. In summary, DOM-based fixes repair or introduce a synchronization point that depends

on the rendering state of a specific DOM element. The main objective of DOM-based fixes is to ensure that tests continue execution only after the rendering of previously accessed web page elements, whether for attributes, states, or data, has been completed. One such example is shown in Figure 1a, where a test flaked by the absence of synchronization point for *d3-flame-graph* element between the rendering (Line 02) and the access (Line 04). The developer repaired this flaky test by adding `waitForSelector` condition to fully load the 'd3-flame-graph' element before the assertion. 18 of 49 tests were fixed with the DOM-based fix strategy, as shown in column $\#^{dom}_{fix}$ in Table 1.

Time-based fixes The most common way to resolve async wait flakiness is by increasing or adding the wait time. The Time-based fixes address the flakiness by explicitly waiting for a certain amount of time at the synchronization point. Figure 1b presents the code snippet of the fix, categorized as *Time-based*, from React-rxjs. When we reran this test, we received the *"Loading..."* still exists in the context, which is unexpected. Developers addressed this flakiness failure by increasing the wait time from `wait(0)` to `wait(50)` to guarantee the completion of the loading element unmount before the assertion expect function at Line 04. On the other hand, Figure 1c provides an example of a *Time-based* fix for a test that experienced flakiness due to a Dom-caused issue with an error message stating, *"Unable to find an element with the text: Search results"*. In response, the developer resolved this flakiness by adding a timeout to ensure the retrieval of the *"Search results"* element at Line 06. Out of 49 tests, the *Time-based* fixing strategy was applied to 31 tests (18 for *DOM-caused* and 13 for *Time-caused*) by developers.

3.3 Observations

Table 1 shows that the time-based repair strategy was adopted for more than half of the async wait tests we studied, i.e., 31 out of 49 (63.3%). Among these tests, around half of them (18 out of 31) were flaky for DOM-based causes (i.e., $\#^{dom}_{cause}$ in $\#^{time}_{fix}$ column), while the remaining half were originally flaky for Time-based causes (i.e., $\#^{time}_{cause}$ in $\#^{time}_{fix}$ column). We speculate that this dominance of the time-based repair strategy across different causes is related to the difficulty of identifying for which element to wait; depending on the design and complexity of the system, pinpointing the right component to wait for may not be straightforward [17, 39].

For this dominating *Time-based* fixing strategy, we further inspect how developers have selected new waiting times to address the flakiness. Thus, we compare the time values in the original flaky tests with those chosen by developers to resolve the flakiness for the flaky tests incurred and resolved by the Time-based cause and strategy. Overall, the wait time in flaky tests ranges from 0 to 1500 ms with an average of 466 ms[3], whereas the time in the developer fixes is between 50 and 30000 ms with an average of 1282 ms. We further found that most of the ratios between the fixed wait time chosen by developers and the original flaky time were concentrated between two and four times, with an average ratio of 3.43. From these observations, we conclude that developers often choose long wait times to reduce test flakiness.

This large gap between the waiting time in flaky tests and developer fixes implies two conflicting goals during web front-end testing: *1) to reduce execution time* (cost) and *2) to alleviate test flakiness* (stability). Our study shows that developers often set a short wait time to reduce test execution costs and switch later to a longer time to alleviate flakiness. A recent study by Lam et al. demonstrated that developers often increase wait times to fix async wait flaky tests and that these time values can be refined to reduce test effort, i.e., execution time [17]. In this paper, motivated by these findings, we propose an efficient way to repair async wait flaky tests in web front-end testing by finding a fair wait time to reduce flakiness and execution time in advance of developers.

[3]Wait 0 means a task is executed at the earliest available idle time of the main thread. This is due to JavaScript in a browser executing on a single thread.

Of 49 asynchronous wait flaky test failures, 63% were addressed through the Time-based strategy of adding or increasing delay regardless of their causes (DOM or Time issues). The large gap between the time in flaky tests and the developer's fixes implies two conflicting goals between the cost and stability during the web testing.

4 AUTOMATED REPAIR OF ASYNC WAIT FLAKY TESTS

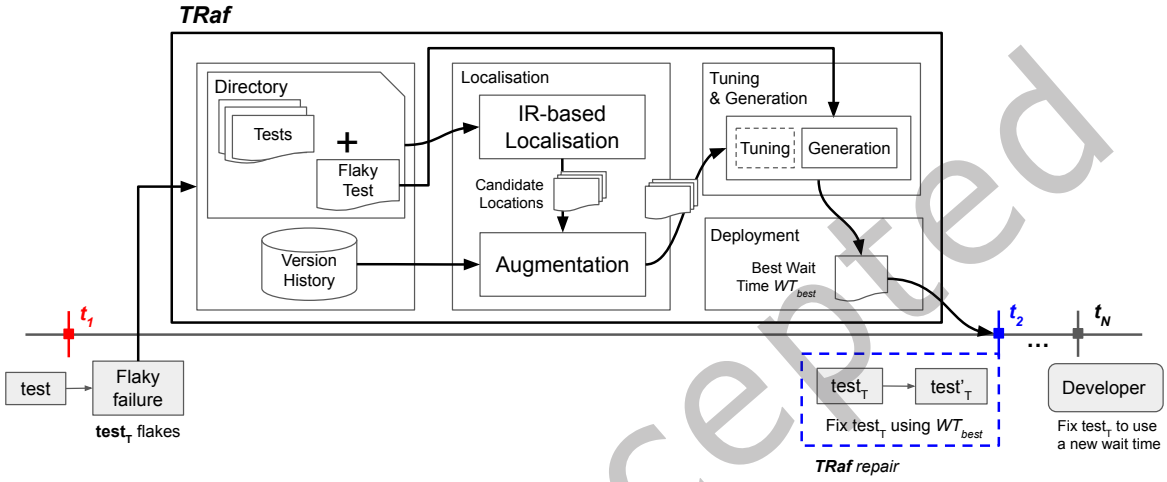


Fig. 2. Overall architecture and application of *TRaf*. The dotted borderline means users can skip the Tuning. *TRaf* can be used right after the occurrence of flaky failure (t_2), without waiting for developers to inspect the test flakiness (t_N).

Motivated by the findings of our preliminary study, we propose *TRaf*, short for Time-based Repair of Async wait Flaky tests, a new automated time-based repair framework for async wait flaky tests in web front-end testing. Figure 2 shows the overall architecture and usage of *TRaf*. When a new flaky test arrives (at t_1), *TRaf* localizes a new wait time value from the past and current codebase and patches the flaky test using the obtained time value, which can be further optimized depending on the configuration. Developers can then continue with a newly repaired test without manually addressing them (t_2).

To sum up, *TRaf* consists of two main phases: 1) localization and 2) tuning and generation. In the localization phase, *TRaf* uses an information retrieval (IR)-based approach to identify relevant lines of code to extract candidates of a new wait time for a given flaky test. It then augments this candidate set by analyzing the version history of the code under inspection. In the tuning and generation phase, *TRaf* first ranks the candidate lines in descending order of their likelihood to contain an adequate wait time; it then sequentially extracts and uses new time values to generate patches from this ranked list or goes through an additional dynamic tuning of the time value before the repair.

4.1 Localization

To automatically repair a test failure, including async wait test flakiness, we need ingredients to compose a fix-patch in addition to fixing locations and operators. The plastic surgery hypothesis is one of the most adopted strategies to retrieve fixing ingredients in the Automatic Program Repair (APR) domain [1, 21, 26, 29, 46]. The underlying intuition is that the changes made to the codebase contain existing code fragments and thus can be reconstituted using these fragments [1, 29]. Many existing works on search-based APR build their techniques

upon this hypothesis [26, 46]. Wen et al. showed that effective fixing ingredients likely exist in the code similar in context to the target fix location [46]. Liu et al. demonstrated the crucial role of fixing ingredient retrieval in template-based APR [26].

In Section 3.2.2, we observed that developers often resolve async wait flakiness by adding or increasing the wait time. With further analysis, we found that developers often select a long wait time, around three times longer than the time in the flaky test on average, for the repair. *TRaf* addresses asynchronous wait flakiness by replacing the original flaky wait time in a test with a new time value that is sufficient for the asynchronous call to complete while avoiding wasting resources. Hence, the main objective of the localization phase of *TRaf* is to identify the location (i.e., line) to extract these efficient fixing ingredients of *TRaf*, which will be the new wait time values.

4.1.1 Information-Retrieval based localization. Considering test flakiness as a specific type of test failure that flakes the outcome, *TRaf* adopts an IR-based Fault Localization (IRFL) approach to identify the lines of code to obtain adequate wait time values. IRFL approaches leverage the lexical similarity between the code and bug reports to localize faults in code [20, 40, 47]: *the more similar the code element is to a given bug report, the more likely it is to include a fault.* In this study, we reformulate the idea behind IRFL for our problem and hypothesize: *the more similar a line of code is to the code that causes test flakiness, the more likely the line is to contain an alternative time value (i.e., fix ingredients)*

Prior studies found the majority of fixes (71%) for flaky tests are located in the test code [17, 27]. In the domain of web testing, many functions focus on manipulating the Document Object Model (DOM) and interacting with various web events, such as a user interaction (e.g. a user hovering over a button element) or another event-triggering action, (e.g. the browser finishing loading a web page) [8, 38]. The recurrent nature of these activities yields similar patterns in the form of event actions and waiting strategies. In general, We found that many test files used a couple of similar statements related to async waits and wait time values, especially in the test folders. Moreover, these wait time values often vary within a certain range of time values. Consequently, similar codes and functions will appear consistent in distinct tests conducted within the same project.

TRaf employs a simple IRFL method based on the Vector Space Model to compute the lexical similarity between the fix location (i.e., the line that triggers async wait flakiness). We assume the fixed locations to modify/insert a wait statement to be given.[4] To provide context of the flakiness, one line before and one line after the fixed location, more precisely, the line of the asynchronous call, are processed together to generate a query. All other lines become documents to retrieve. We limit our inspection to the lines of test files in the same directory as the flaky test file.

For the preprocessing, we remove JavaScript keywords from the vocabulary, a collection of unique words in documents. Both the query and the documents are then converted into fixed-length weighted vectors. Each weight of this vector corresponds to the term frequency-inverse document frequency (*tf-idf*) of a unique term. The term frequency (tf) measures how frequently term t appears within document d, while inverse document frequency (idf) evaluates how rare the term t is across all documents D. Together, *tf-idf* assesses the importance of the term t in the document d: if a rare term appears frequently in a given document, this term is likely a keyword of the document. The below equations detail how *tf-idf* is computed for term t, document d, and a document set D.

[4]The fix locations were often easily obtainable, as most failures occurred at waiting statements or assertions, as shown in Section 3.2.2. In other cases, we simply used the last statement in the stack trace.

$$tf(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}},\ f_{t,d} = \text{the frequency of } t \text{ within } d$$

$$idf(t,D) = log\frac{|D|}{|d \in D : t \in d|},\ \ tf\text{-}idf(t,d,D) = tf(t,d) \cdot idf(t,D)$$

We compute the cosine similarity between weight vectors of the fixed location (i.e., the flaky line) and every candidate line.

4.1.2 History-based Augmentation. Developers often make similar mistakes more than once, for instance, due to the incomplete understanding of the changes they made [14]. Previous studies of fault localization and defect prediction exploit the repetition of similar failures to improve the accuracy, either directly or indirectly, by mining the project version history [36, 43, 47]. *TRaf* also investigates the version history of a project, inspecting the past changes to augment the pool of candidate locations to find new waiting times. We assume that developers made or fixed asynchronous wait flaky tests that failed in a similar context and, thereby, can obtain valuable information about the repair from the past. Hence, *TRaf* tracks down past changes of the lines identified from the previous IR-based approach and collects past versions of these lines with time values different from those currently have; it also examines the past changes of the *fixed location* since it might undergo (and be treated for) similar asynchronous wait issues before.

After retrieving all the candidate lines from the present and the past, *TRaf* extracts waiting time values, i.e., its fixing ingredients, from these lines; the values smaller than the time used in the flaky test are excluded as they are out of interest. *TRaf* assigns each candidate time value the cosine similarity value of its origin line; for cases where the same time value exists in more than one line and thus has multiple similarity values, the largest is considered.

4.2 Tuning and Generation

The main task of the Tuning and Generation phase is divided into two folds: 1) tune the wait time value and 2) generate a patch with it. *TRaf* allows developers to configure the amount of effort spent on the tuning, from only a few trials to finding the optimum, depending on the available resources. Under a fast-release cycle, developers can skip this step and go directly to the generation, as shown in Figure 2.

TRaf refines the time value by systematically trying out different values between the original flaky time (the left boundary B_l) and the time value obtained from the localization step (the right boundary B_r). In the absence of an explicit time delay in the flaky test, such as the tests being flaky for DOM-based causes but addressed by the Time-based fix strategy, B_l is set to be 0. For the right boundary B_r, *TRaf* first sorts the time values in descending order of the similarity score of their source line to the fixed location. It then generates the patch, for instance, via the wait-time replacement, using the time value at the top, and validates whether this patch resolves the test flakiness by monitoring 100 test reruns as we did in the preliminary study. *TRaf* continues to do this until encountering the time value that resolves the flakiness, the one consistently passing across 100 runs.

By default, *TRaf* directly returns B_r and the corresponding fix to the users; we call *TRaf* without the extra tuning as *static*, as it does not need any test run until the patch validation; we call the other case with tuning *dynamic*. In most cases, the candidates from the localization step include at least one valid time value that addresses a flaky test. Nevertheless, this is not always guaranteed: all the obtained time values can be smaller than the wait time in the flaky test. *TRaf* marks such cases as *"non-applicable"*, counting them as failures.

Under the dynamic tuning mode, *TRaf* conducts a binary search between the obtained search boundary until the time difference between the left (B_l) and right (B_r) search boundaries is less than the threshold thr_{time} or when it reaches the maximum number of searches. Specifically, for each iteration, *TRaf* sets the new wait time,

wt_{new}, to be the middle of the current search boundary. It then patches the flaky test using wt_{new} and checks whether the test is no longer flaky, i.e., constantly passing for a predefined number of test reruns. If it is still flaky, *TRaf* uses wt_{new} as the new left boundary and continues to search. The search ends either when the search range becomes smaller than a given threshold thr_{time} or when the maximum number of searches has been met.

4.2.1 Repair. To resolve test flakiness, *TRaf* replaces the wait time value of an asynchronous call in a flaky test with the new time value it found. For the cases without an explicit wait time, *TRaf* adds the obtained wait time as an additional timeout to the asynchronous call to a DOM element, similar to the example in Figure 1c, or inserts an additional wait function after the call. In our study, we take the last statement in the stack trace as the fixed locations. As shown in Figure 3, this information can be easily obtained in most cases. For cases where a test is flaky by the interactions of multiple asynchronous calls,[5] we take the last asynchronous call as the fixed location.

```
 Unable to find an element with the text: Search results. This could be because the text is broken up by multiple elements.
In this case, you can provide a function for your text matcher to make your matcher more flexible.
       140 |
     > 141 |  await screen.findByText('Search results')
         ...
      at waitForWrapper (node_modules/@testing-library/dom/dist/wait-for.js:187:27)
      at findByText (node_modules/@testing-library/dom/dist/query-helpers.js:101:33)
      at _callee$ (products/jbrowse-web/src/tests/TextSearching.test.tsx:141:16)
      at tryCatch (node_modules/regenerator-runtime/runtime.js:63:40)
      at Generator.invoke [as _invoke] (node_modules/regenerator-runtime/runtime.js:294:22)
      at Generator.next (node_modules/regenerator-runtime/runtime.js:119:21)
      at asyncGeneratorStep (node_modules/@babel/runtime/helpers/asyncToGenerator.js:3:24)
      at _next (node_modules/@babel/runtime/helpers/asyncToGenerator.js:25:9)
```

Fig. 3. Error message with location info of the repair case.

5 EXPERIMENTAL SETTINGS

5.1 Research Questions

To evaluate the performance and usefulness of *TRaf*, we ask the following three research questions.

RQ1 Effectiveness: How effective is *TRaf* in finding the right wait time to repair async wait flakiness?

We evaluate the effectiveness of *TRaf* in terms of the wait time and the test execution time, mainly inspecting how much time it can save *beforehand* compared to the developer-written patches. *TRaf* runs in static mode (*TRaf*^{S}) by default and can be further configured to dynamically tune the obtained wait time (*TRaf*^{D}), as indicated in Figure 2 with the dashed outlined box. The dynamic tuning of *TRaf* can optimize the time values from the previous localization. However, this may not always be feasible, especially for projects under the fast release cycle, requiring rerunning tests for every optimization attempt. Thus, we examine an additional tuning configuration that is positioned between each end, i.e., the default, which is no tuning at all (*TRaf*^{S}), and tune-till-end, i.e., *TRaf*^{D}, by allowing *TRaf* to try out a few shots of tuning until meeting the first valid time value that resolves the flakiness with a shorter wait time than the developers' fixed time. We call this configuration *first-match*, as the optimization ends immediately after the first successful attempt; we refer to *TRaf*^{D} with the *first-match* tuning as *TRaf*$^D_{FM}$. By trying out the different configurations, we assess the performance of *TRaf*, investigating *whether it can statically find time values that can address async wait flakiness* and *how the dynamic tuning can further reduce the execution time.*

For this analysis of *TRaf*'s effectiveness, we compared the average execution times of 100 repeated test runs after repair, accounting for the inherent stochastic nature of the test runs. To confirm the statistical significance, we compared the execution times of these runs between *TRaf* and two baselines: developers' fixes for the static

[5]Only two tests out of 31 flaky tests that we studied were flaky by the interactions of multiple asynchronous calls.

$TRaf$ ($TRaf^S$) and $FaTB_{FM}$ for the dynamic $TRaf$ ($TRaf^D_{FM}$); Section 5.2 illustrates the details of these baselines. We used $TRaf^S$ to see if $TRaf$ could produce acceptable results without additional adjustments. Without an upper bound, each refinement attempt converges to the same optimal value regardless of the initial wait time. To examine the impact of a better initial wait time, we used the *FM* variants for both $TRaf$ and the baseline ($TRaf^D_{FM}$ and $FaTB_{FM}$), forcing the dynamic optimization to stop at the first success. The Mann-Whitney U-test with a 95% confidence level was used for this analysis, with the alternative hypothesis that $TRaf$ consistently outperforms (i.e., reduces execution times) the baseline.

In Section 3, we define two root cause classes of async wait flaky tests based on the type of calls where flaky behavior has been observed: *Time-caused* and *DOM-caused*. $TRaf$ applies *Time-based* fixes regardless of the root causes. It is motivated by the preference of developers to adapt or introduce latency to resolve async wait flakiness. Nevertheless, the performance of $TRaf$ may still vary depending on the root cause characteristics. Hence, we investigate how $TRaf$ performs on different classes of root causes. For *DOM-caused*, there are some cases where the fix-location, i.e., the call where flaky behavior occurred, is ambiguous due to multiple calls to DOM elements being made before the flaky behavior was observed. Figure 4 shows such an example: in this code snippet, a test made four calls to different DOM elements (Lines 03,04,05,06) before it experienced flaky failure at the last call at Line 06. In this case, the flaky behavior can be caused by the last call on Line 06, but it can also be incurred by the intertwined outcomes of the previous four async wait calls. To inspect how $TRaf$ performs on such complex/ambiguous cases, we define another root cause class, *Multi-dom*, that contains async wait flaky tests where multiple asynchronous wait calls to DOM elements were executed before their flaky failures. For this class, $TRaf$ considers the last asynchronous call as the fixed location.

```
01.  export async function run() {
02.      ...
03.      await clickRecordButton(devtools);
04.      await click(page, "button");
05.      await clickRecordButton(devtools);
06.      await devtools.hover('[data-type="flamegraph"] [data-name="Counter"]');
07.      // Wait for possible flickering to occur
08.  +   await wait(1000);
09.      expect(log.filter(x => x.type === "highlight").length).to.equal(1);
10.  }
```

Fig. 4. An example of case: waitfor time after multiple element interactions.

RQ2 Efficiency: How efficient is $TRaf$ at repairing async wait flaky tests?

The cost of $TRaf$ mainly comes from test reruns. Hence, we investigate the efficiency of $TRaf$ by inspecting the number of test reruns during the entire repair process (n_{total}), i.e., from localization to repair. Test reruns occur when $TRaf$ validates a generated patch. Each rerun represents 100 executions with the same wait time value. We examine the results of 100 test executions, and if all of them pass, it is deemed that the patch is valid. However, if there is a failure, it indicates that the flakiness still exists. The patch validation takes place under two circumstances: 1) after the localization to select a potential candidate wait time and 2) for every dynamic tuning attempt. We differentiate the number of test reruns, i.e., the cost in these two circumstances to obtain a more comprehensive picture of the efficiency of $TRaf$, dividing n_{total}) into two (n_{loc} and n_{tuning}) as follows:

- n_{loc}: the number of test reruns to identify the candidates for the new time value for the flaky test during the *localization* stage. This is the default and the minimal cost of $TRaf$.
- n_{tuning}: the number of test reruns during the dynamic tuning. This is an *additional cost* if the user configures $TRaf$ to further tune the values obtained from the localization stage.

- n_{total}: the total number of test reruns during the entire execution of *TRaf*. For $TRaf^S$, this will be the same as n_{loc}, and for $TRaf^D$, this will be the sum of n_{loc} and n_{tuning}

We evaluate the efficiency of *TRaf* under the three tuning configurations used in RQ1 to evaluate the effectiveness, i.e., $TRaf^S$, $TRaf^D_{FM}$, and $TRaf^D$.

RQ3 Augmentation Contribution: How does the history-based augmentation contribute to effectiveness of *TRaf*?

One of the main intuitions of *TRaf* is that the hints for better wait time may exist in the past relevant code due to the repetitive nature of developers. To investigate how history-based augmentation contributes to *TRaf* in obtaining better waiting times, we compare the performance of *TRaf* with and without the augmentation with static and dynamic modes. More specifically, we compute the difference between the waiting time ($diff_{WT}$) and the execution time ($diff_{ET}$) acquired by *TRaf* with and without the augmentation, inspecting whether the augmentation allows *TRaf* to find a more *efficient* latency value to resolve an asynchronous issue.

The current codebase may not contain any valid time value, resulting in *TRaf* being *non-applicable*, as described in Section 4.2. Thus, in addition to wait time and execution time, we compare the Repair Rate (*RR*), which is the ratio of the number of cases where *TRaf* successfully finds a valid time value, i.e., *applicable*, over the total number of studied cases including the failed ones (i.e., *non-applicable*) between with and without the history-based augmentation: $RR = \frac{cnt_{applicable}}{cnt_{applicable}+cnt_{non-applicable}}$.

5.2 Baseline

TRaf aims to address async wait flakiness by proposing an efficient wait time for asynchronous calls without waiting for developers to examine the flakiness. Figure 2 depicts the timeline from the detection of flakiness to the repair. At t_1, upon the detection of async wait flakiness, *TRaf* instantly tries to resolve this flakiness; depending on the available resources, it further optimizes a new wait time dynamically (i.e., dashed box). With the resolved test, the development continues (at t_2).

The main strength of *TRaf* comes from its *application timing* and its ability to find *new wait time values statically*. Hence, we employ two baselines: 1) using the original wait time in developer-written fixes to evaluate the capability and effectiveness of *TRaf* in finding new wait time values and 2) using the refined version of this developer-chosen wait time via FaTB [17] further to assess its effectiveness with regard to the application timing. Both FaTB and *TRaf* are in the same condition that runs 100 times by default to evaluate the fix. The focus will be on how *TRaf* performs compared to the cases where the time that resolves async wait flakiness is already known (t_N at Figure 2) and how much effort it can save *in advance*.

5.2.1 The Wait time of developers. We use the original time values chosen by developers to address flakiness as the first baseline to investigate how comparable or efficient the repair of *TRaf* is compared to the developer-written patches. We obtained these time values from the commit that fixes the test flakiness; we validate the wait time values selected by developers by rerunning the test 100 times to check for flakiness, as we did for the validation of wait time in the preliminary study (Section 3) and the repair of *TRaf* (Section 4.2).

5.2.2 FaTB: Posterior Repair. FaTB is a *post-processing approach* that aims at reducing test execution time while retaining the initial decreased flake rate of the developer's fixes for async wait flakiness. To achieve this, FaTB refines the wait time in developer-written fixes. It systematically tries out different time values between the original flaky time and the time chosen by developers for the fix, checking the middle of two boundary values to decide which direction to proceed further, similar to how *TRaf* refines a new wait time under dynamic tuning configuration. Although our dynamic tuning, $TRaf^D$ operates similarly to FaTB and will converge to the same value close to the optimal when they are allowed to refine the time until the end, their cost and paths to achieve such performance likely differ. Thus, we further compare the performance of FaTB and $TRaf^D$ on their first

successful refinement, implementing FaTB to stop at the first successful refinement. We call this variation of FaTB FaTB_{FM}; we changed FaTB_{FM} to use the resolution of flakiness instead of the flake rate, configuring its tolerance of flaky test failures as zero out of 100 test reruns. We compare $TRaf^{D}_{FM}$ and FaTB_{FM} in terms of the effectiveness and the efficiency.

Figure 2 depicts the occurrence of async wait flakiness and how *TRaf* and developers address it. Unlike *TRaf*, which is applied immediately upon detecting flakiness, FaTB is executed after developers have resolved the flakiness (at t_N), with the intention of improving the developer-written patch. Hence, our objective of using FaTB as the baseline is to study how the *TRaf* performs compared to the post-processing where the time developers used for the repair is known.

5.3 Version History Collection

TRaf expands its pool of candidate lines for extracting time values by further examining their previous versions, assuming developers repeatedly make similar mistakes. To obtain these lines in previous versions, we tracked down their past changes using Git v.2.37.0 [2] and `difflib` module of Python 3.9.15. SequenceMatcher determines the line deletion, insertion, and modification.

5.4 Configuration

TRaf allows users to configure the effort spent on the tuning via the time threshold thr_{time}, which is the minimum search range between the left and right ends. We set this threshold to 5 ms in our experiment, investigating how much test execution time can be saved at the maximum if there are enough resources for tuning. When the threshold (thr_{time} = 5 ms) is reached, *TRaf* terminates tuning and returns the best wait time, i.e., saving execution time the most while still resolving flakiness, to developers.

5.5 Implementation and Environment

TRaf is implemented in JavaScript and runs on Node.js v17.5.0 and NPM v8.4.1. All experiments were conducted on a machine equipped with an Apple M1 Max and 32 GB RAM. To evaluate our approach, we added an additional machine with a configuration of an Apple processor 2.3 GHz Dual-core Intel Core i5 and 8 GB of RAM. The replication package and the dataset are publicly available at https://doi.org/10.5281/zenodo.10728548.

6 RESULTS

6.1 RQ1. Effectiveness

Table 2 compares the effectiveness of *TRaf* in static ($TRaf^{S}$) and dynamic ($TRaf^{D}$) modes with the developer-written fixes (*developer*) and FaTB (FaTB_{FM}), in terms of the wait time and the test execution time. *optimal* in the table refers to the time value that minimizes the wait time and thereby the test execution time while resolving the flakiness; with sufficient resources, both *TRaf* ($TRaf^{D}$) and FaTB converge to this *optimal* value at the end of their refinement process.

TRaf runs without dynamic tuning ($TRaf^{S}$) by default. $TRaf^{S}$ outperformed developer-written patches (*developer*) in 17 out of 31 cases (54.8%), as shown in gray values, and ties in 13 cases (41.9%) by obtaining the same wait time value with developers. More specifically, as shown in the final row *Total*, *TRaf* obtained the wait time values shorter than those of developers by 47.4% (3094.9 ms) on average, reducing the average test execution time by 11.1% (186.6 ms) compared to the initial test execution time of the developer-written patches. These results imply that *TRaf* is capable of identifying comparable and often better wait times than the ones selected by developers to tackle the flakiness. With dynamic tuning, *TRaf* achieved further reductions in both wait time and test execution time; on average, it reduced wait time by 90.9% (5935.3 ms) and test execution time by 20.2%

Table 2. Comparison of wait time and test execution time between developers' fixes, *FaTB*, and our approach, *TRaf*, on a 32GB machine. *Avg* refers to the average within each root cause, and *Total* denotes the average without differentiating root causes. *optimal* denotes the optimal wait time value to which *FaTB* and dynamic *TRaf* ($TRaf^D$) eventually converge. *dev* denotes the wait and execution times under the developer's chosen values. *p-value* is to check statistical significance; *p-value* smaller than 0.05 are written in bold, and *<0.001* denotes the p-values less than *0.001*. Those obtaining the same performance as the baseline are highlighted in light orange in addition to those bold by outperforming the baselines.

			Wait time (ms)						Execution Time (ms)						
Root		Pass	Baseline			TRaf			Baseline			p-value		p-value	
cause	project[1]	/Fail	flaky	dev	$FaTB_{FM}$	$TRaf^S$	$TRaf^D_{FM}$	optimal	dev	$FaTB_{FM}$	$TRaf^S$	vs dev	$TRaf^D_{FM}$	vs $FaTB_{FM}$	optimal
Time	puppeteer	100/0	500	2000	1250	1000	938	919	2390.2	1679.9	1396.5	**<0.001**	1335.5	**<0.001**	1319.4
	puppeteer	100/0	300	2000	1250	500	475	463	2355.8	1592.1	836.2	**<0.001**	810.2	**<0.001**	803.6
	r-uploady	100/0	200	1500	850	800	500	229	1764.2	1192.7	1018.1	**<0.001**	702.7	**<0.001**	397.7
	r-uploady	100/0	200	1500	850	800	500	229	1714.8	1100.3	1000.2	**<0.001**	696.7	**<0.001**	405.5
	r-uploady	100/0	60	200	165	200	165	159	1674.6	1611.9	1674.6	0.732	1611.9	0.458	1607.7
	r-uploady	100/0	1500	3000	2250	2500	2344	2247	3765.8	3031.6	3104.9	**<0.001**	2961.2	**<0.001**	2866.9
	react-rxjs	100/0	100	110	105	110	105	104	127.1	126.6	126.7	0.467	126.2	0.424	126.0
	react-rxjs	100/0	0	50	25	10	5	5	123.3	113.6	111.8	**<0.001**	99.4	**0.003**	99.4
	react-rxjs	100/0	100	110	108	110	108	108	134.0	127.8	134.0	0.405	127.7	0.499	127.7
	predictive	100/0	1000	2000	1500	1500	1250	1133	1306.6	1191.7	1191.7	**<0.001**	1109.2	**<0.001**	1088.6
	predictive	100/0	1000	2000	1500	1500	1250	1165	1092.9	1074.2	1074.2	**<0.001**	985.3	**<0.001**	974.1
	edelweiss	100/0	100	200	150	200	150	141	986.6	857.7	986.6	0.412	856.3	0.476	832.4
	cordless	100/0	1000	2000	1500	2000	1500	1024	937.1	862.6	937.4	0.561	862.6	0.534	760.2
Avg.	-	-	466.2	1282.3	884.8	863.8	714.6	609.7	1413.2	1120.2	1045.6	-	945.0	-	877.6
Dom	dotcom	100/0	-	30000	15000	30000	15000	587	2401.2	2162.0	2401.2	0.633	2162.0	0.341	2134.0
	jbrowse	100/0	-	20000	10000	10000	5000	821	2267.9	2243.0	2243.0	**<0.001**	2220.9	**<0.001**	2204.3
	jbrowse	100/0	-	10000	5000	1000	500	477	1064.0	1049.9	1056.7	0.101	1043.8	**<0.001**	1041.7
	jbrowse	100/0	-	10000	5000	1000	500	258	929.1	905.1	839.7	**<0.001**	818.6	**<0.001**	817.0
	jbrowse	100/0	-	10000	5000	10000	5000	1172	1677.2	1662.7	1677.2	0.487	1662.7	0.500	1639.7
	jbrowse	100/0	-	25000	12500	10000	5000	1094	1512.9	1500.6	1482.2	**<0.001**	1469.9	**<0.001**	1460.7
	jbrowse	100/0	-	20000	10000	10000	5000	1485	1027.5	988.6	988.6	**<0.001**	962.5	**<0.001**	916.9
	coda	100/0	-	1000	500	1000	500	225	7820.7	7703.3	7820.7	0.559	7703.3	0.538	7662.8
	elm-select	100/0	-	100	50	100	50	13	233.5	180.6	228.8	**0.001**	178.4	0.415	142.7
	flashmap	100/0	-	10000	5000	1000	500	258	1023.4	996.1	910.6	**<0.001**	904.5	**<0.001**	880.3
	flashmap	100/0	-	10000	5000	1000	500	258	958.0	959.7	870.7	**<0.001**	864.6	**<0.001**	860.7
	flashmap	100/0	-	10000	5000	1000	500	477	655.0	635.4	646.0	0.430	640.9	0.492	640.6
	flashmap	100/0	-	20000	10000	10000	5000	821	2391.3	2365.1	2355.8	**<0.001**	2274.0	**<0.001**	2196.6
	coinscan	100/0	-	3000	2250	3000	2250	1985	3424.0	3299.0	3424.0	0.468	3299.0	0.480	3242.0
	keptn	100/0	-	500	250	500	250	16	1783.0	1059.0	1783.0	0.503	1059.0	0.567	762.4
	material	100/0	-	5000	2500	5000	2500	391	409.0	382.5	409.0	0.446	382.5	0.555	380.0
Avg.	-	-	-	11537.5	5815.6	5912.5	3003.1	646.1	1844.8	1755.8	1821.1	-	1727.9	-	1686.4
Multi	client	100/0	-	100	50	100	50	10	248.5	250.7	248.5	0.472	250.7	0.536	217.0
	preact	100/0	-	1000	500	500	250	102	3797.0	3172.0	3172.0	**<0.001**	3021.0	**<0.001**	2855.0
Avg.	-	-	-	550.0	275.0	300.0	150.0	56.0	2022.8	1711.3	1710.3	-	1635.8	-	1536.0
Total	-	-	-	6528.1	3390.4	3433.2	1859.4	592.8	1675.3	1486.4	1488.7	-	1393.7	-	1337.5

[1] The complete names of the projects are as follows: puppeteer-core-controller, react-uploady, react-rxjs, predictive-text-studio, edelweiss-ui, cordless, dotcom-rendering, jbrowse-components, codacollection-e2e, elm-select, flashmap-production, codacollection-e2e, keptn, material-ui, client, preact-devtools.

(337.8 ms) compared to developer-written patches by finding a wait time close to the optimum (i.e., the column *optimal*). The cost of obtaining such performance will be discussed in the subsequent RQ2.

In many cases, the resources for repair are limited. The column $TRaf^D_{FM}$ in Table 2 demonstrates how *TRaf* performs with limited resources by stopping at the first successful refinement (i.e., *FM*). We compare $TRaf^D_{FM}$ with $FaTB_{FM}$, i.e., the *FM* variation of FaTB described in Section 5.2.2, to investigate how the initial valid wait time affects a subsequent dynamic optimization process: $FaTB_{FM}$ optimizes *developer-chosen wait time values* whereas $TRaf^D_{FM}$ tunes *new wait time values* that it finds statically without knowing what will be used by developers for repair. The results show that $TRaf^D_{FM}$ outperformed $FaTB_{FM}$ in around half of the cases (i.e., 15 out of 31 cases, as shown in blue values), reducing the wait time by 45.2% (1531 ms) and the test execution time by 6.2% (92.7 ms) on average. For the remaining 16 cases, $TRaf^D_{FM}$ obtained the same wait time with $FaTB_{FM}$ in 13 cases. When comparing these refined wait times of $TRaf^D_{FM}$ and the corresponding test execution time with those of developers (*developer* column), the wait time and the test execution time decreased by 4668.7 ms (71.5%) and by 281.6 ms

(16.8%), respectively, in 29 cases, suggesting the usefulness of dynamic tuning under the limited resources. These findings further highlight the effectiveness of *TRaf*, especially regarding its inherent strength of being applicable right upon the async wait flakiness detection *without waiting for developers to examine the flaky tests.*

The rows *Time-caused*, *Dom-caused*, and *Multi-dom* in Table 2 differentiate the performance of *TRaf* on the studied async wait flaky tests by their respective causes. For tests flaked by *Time-caused* issues and those by *Dom-caused* issues, *TRaf* saved the average wait time 32.6% (418.5 ms) (*Time-caused*) and 48.8% (5625.0 ms) (*Dom-caused*) and test execution time 26.0% (367.6 ms) (*Time-caused*) and 1.3% (23.7 ms) (*Dom-caused*), respectively. Flaky tests in *Dom-caused* were frequently handled by *adding* a timeout to an asynchronous call to a DOM element, similar to the example in Figure 1c. Hence, we suspect the notable difference in the scale of improvement between wait time and test execution time in the *Dom-caused* flaky tests is due to function calls that often terminate before a given timeout by satisfying DOM-related conditions in the same call during test reruns. For those from Multi-dom, the wait time and the execution time decrease by 45.5% (250 ms) and 15.4% (312.5 ms). While *TRaf* saved the most execution time for flaky tests in the *Time-caused* category, it successfully repaired async wait flakiness regardless of their causes, including Multi-domcases where the fix-location is ambiguous. Based on these, we posit that *TRaf* can generate generally effective repairs for async wait flakiness.

The 13th and 15th columns (i.e., *p-value vs dev* and *p-value vs* $FaTB_{FM}$) of Table 2 present the p-values obtained from the Mann-Whitney U-test, evaluating the statistical significance of the observed improvements. In the comparison between $TRaf^S$ and the developers' fixes, we rejected the null hypothesis in 17 out of 31 cases (54.8%) (i.e., p-value < 0.05). Among the 15 cases where we failed to reject the null hypothesis, we observed that in 12 of these cases, highlighted in light orange, $TRaf^S$ selected the same wait time as the developers' fixes to mitigate flakiness. *TRaf* selects candidate wait times from the current and past codebase by computing the lexical similarity and reviewing the past changes. Since these time values were originally selected by developers for different tests or in the past, some of the candidates found by *TRaf* are expected to match those chosen by developers. The key advantage here is that *TRaf* identifies these candidates before developers attempt to fix the flaky tests. In the comparison between $TRaf^D_{FM}$ and $FaTB_{FM}$, 17 out of 31 cases (54.8%) demonstrated statistical importance in their observation, rejecting the null hypothesis. Nine cases, highlighted in light orange, were those where $TRaf^D_{FM}$ obtained the same average execution wait time as the $FaTB_{FM}$ baseline, in addition to the same wait time.[6] Combined, of 31 cases, 26 showed that *TRaf* successfully outperformed or matched the baseline without initial operating before the developers' involvement, unlike the baseline developers (using developers' chosen time) and FaTB (refining developers' chosen time); this automation via *TRaf* implies the potential to reduce developers' effort in addressing test flakiness.

Figure 5 summarizes the effectiveness of *TRaf* compared to the developer-written patches for async wait flaky tests; the y-axes of Figures 5a and 5b are the ratios of the new wait time and the respective test execution time to the time selected by developers and the corresponding execution time, respectively: *the smaller the ratio is, the more effective TRaf is.* Overall, *TRaf* reduced the wait time and, thus, the execution time for most flaky tests, as summarized in the last three plots in these figures (*All*). For *Dom-caused* flaky tests, there were a few cases where the execution time sometimes degraded after the repair. By manually inspecting such cases, we found that they were due to the combined impact of early termination by satisfying DOM-related conditions in async wait calls and using the average execution time of repeated test reruns as the final test execution time.[7] Nevertheless, in most of the studied flaky tests, *TRaf* discovered more efficient wait time values than developers, with or without the tuning, showing the existing and potential benefits of efficiently handling async wait flakiness. The wait time

[6]We only highlight the cases where we obtained *exactly* the same average for the execution time for clarity, here; this number, i.e., nine cases, can increase if we further investigate the similarity between the distribution of execution time values, by including the cases where *TRaf* computes the same wait time.

[7]The randomness of test execution often resulted in different test execution times with the same wait time, especially when combined with the early termination by satisfying an associated DOM conditions

and the execution time are reported in milliseconds (ms). Hence, the reported improvement could be perceived as trivial with the absolute values alone. However, we emphasize that the improvement here is per test execution. Given the prevalence of flaky tests in a development environment that requires intensive testing regularly, e.g., a project under continuous integration [30], we posit that the *collective reduction* of these small improvements can save a substantial amount of future expenses of testing; **Section 7**, i.e., *Discussion*, will detail the potential benefits in practice.

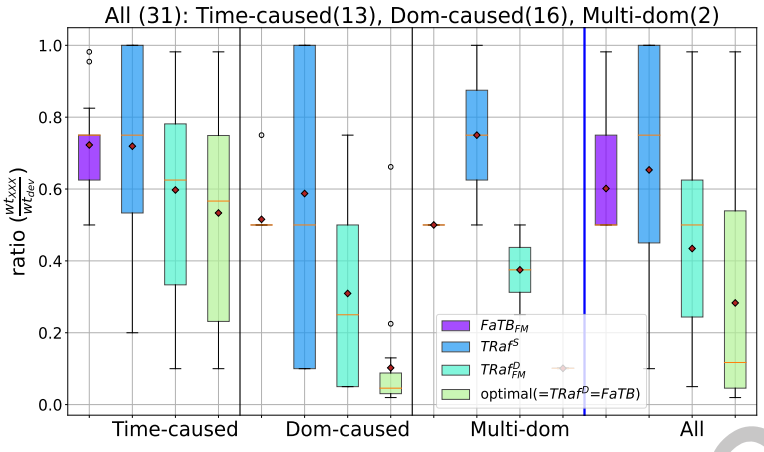
Answer to RQ1: *TRaf* can statically repair async Wait flakiness by finding more efficient wait times, reducing the test execution time by 11.1% compared to the developer-written patches. With additional dynamic tuning, it reduced the test execution time by up to 20.2% and by 16.8% at the first successful refinement, which requires two trials of test reruns (one for patch validation and one during the refinement). The comparison with a post-processing approach (FaTB_{FM}) further demonstrates that *TRaf* can achieve comparable and sometimes better performance without waiting for developers to examine the flakiness, reducing an average of 6.2% of the execution time.

6.2 RQ2. Efficiency

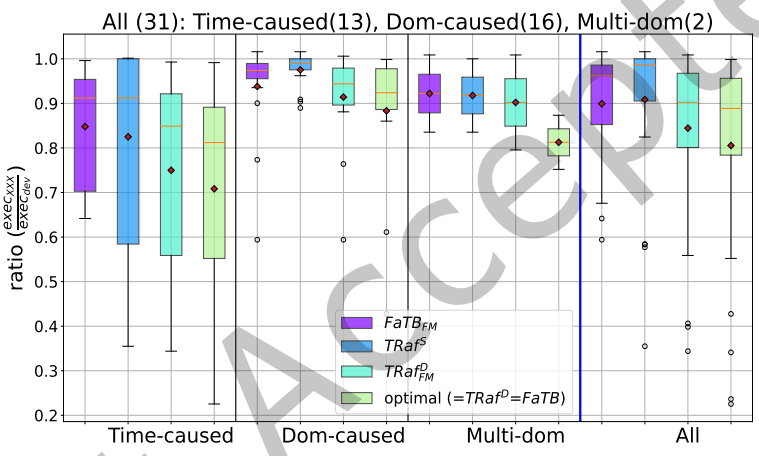
Table 3 presents the number of test reruns in *TRaf* during the entire execution, i.e., from the localization to the repair and validation (n_{total}), during the localization to obtain candidates (n_{loc}), and during the dynamic optimization (n_{tuning}). FaTB refines the wait time selected by developers to alleviate async wait flakiness. Thus, the total number of test reruns (n_{total}) of the baselines FaTB_{FM} and FaTB is calculated as the number of refinement attempts (n_{tuning}) plus one, with n_{loc} always zero; the extra test run (plus one) is to re-check the validity of the time value in a developer-written patch. Overall, *TRaf* and FaTB show consistent efficiency across various root causes of async wait flakiness (the rows *Time-caused*, *Dom-caused* and *Multi-dom*) as they did for effectiveness, implying the general usefulness of the *time-based fixing strategy* for different scenarios of async wait flakiness.

Column $\textit{TRaf}^{\mathcal{S}}$ (n_{loc}) in Table 3 shows that the localization module of *TRaf* often placed the flakiness-resolving wait time at the top of its candidate list ($n_{loc} = 1$). Specifically, for 24 out of 31 (77%) studied flaky tests, *TRaf* successfully mitigated the flakiness by using the time value at the top without the need for further inspection of remaining candidates; the average ranking of valid wait time values is 1.2. Moreover, the cost of the localization stage can vary depending on the project. The average cost time ranges from 120 ms to 530 ms under our running experiments and is associated with the size of the test files. Hence, the location stage is a relatively inexpensive procedure, as it merely computes the lexical similarity between statements. Regarding the dynamic tuning, *TRaf* found the optimal wait time with an average of 7.4 trials (i.e., test reruns), as shown in n_{total} of the column $\textit{TRaf}^{\mathcal{D}}$ in Table 3. FaTB also required a similar number of test reruns to obtain the optimal with 8.8 test reruns on average (n_{total} in the column FaTB). This observation highlights the cost efficiency of *TRaf* despite the expenses (n_{loc}) to localize a flakiness-resolving wait time value in addition to the tuning (n_{tuning}) compared to FaTB; *TRaf* even reduced the total expense (n_{total}) by finding a better wait time than the one chosen by developers efficiently, as demonstrated through the effectiveness and efficiency of $\textit{TRaf}^{\mathcal{S}}$.

The columns $\textit{TRaf}^{\mathcal{D}}_{FM}$ and FaTB_{FM} present the efficiency of *TRaf* and its baseline under limited resources. Overall, both approaches underwent a similar number of refinement attempts during the optimization (n_{tuning}) and throughout the entire execution (n_{total}). During the tuning, $\textit{TRaf}^{\mathcal{D}}_{FM}$ and FaTB_{FM} often successfully found the refined value in their first optimization attempt, and for $\textit{TRaf}^{\mathcal{D}}_{FM}$, n_{loc} was often one by accurately localizing a valid wait time that resolves flakiness; $\textit{TRaf}^{\mathcal{D}}_{FM}$ successfully refined the obtained wait time value within two and on average 1.3 runs (n_{tuning}). In RQ1, we observe that $\textit{TRaf}^{\mathcal{D}}_{FM}$ can achieve comparable or even better performance compared to FaTB_{FM} by identifying more effective timeout values than those selected by developers. Hence,



(a) Wait Time Comparison



(b) Execution Time Comparison

Fig. 5. Comparison between the variations of *TRaf*, developer-written patches, and FaTB by calculating the relative improvement over the effectiveness of developer-written patches (*ratio* in the y-axis). *"< 1"* refers to the improvement over the developer-written patches. *All* refers to including all flaky tests.

combined with the analysis of the localization cost and the cost to obtain the optimal, we posit that *TRaf* can *efficiently* propose an adequate wait time to mitigate async wait flakiness to developers in advance.

Answer to RQ2: *TRaf* can efficiently discover adequate wait time values to alleviate async wait flakiness issues without requiring many test reruns (i.e., 77% by the first trial), even with the additional tuning (e.g., at most two trials for first refinement). The comparison between *TRaf* and FaTB (i.e., a post-processing method) regarding the cost to obtain the optimal and the cost for the first successful optimization attempt further highlights the efficiency of *TRaf*, achieving comparable and often better performance.

Table 3. Comparison of the number of test reruns (n_{total}, n_{tuning}, n_{loc}, as described in Section 5.1) between *TRaf* and FaTB. *Average* refers to the average within each root cause, and *Total* denotes the average without differentiating root causes. $TRaf^{S}$ always has zero for n_{tuning}, as there is no dynamic tuning in static *TRaf*. FaTB and $FaTB_{FM}$ only have n_{tuning}, as there is no localization phase in them.

Root cause	project	n_{total} / (n_{tuning})		n_{total} / (n_{loc} , n_{tuning})		
		$FaTB_{FM}$	FaTB	$TRaf^{D}_{FM}$	$TRaf^{S}$	$TRaf^{D}$
Time-caused	puppeteer	2 / (1)	10 / (9)	5 / (2, 3)	2 / (2, -)	9 / (2, 7)
	puppeteer	2 / (1)	10 / (9)	5 / (2, 3)	2 / (2, -)	6 / (2, 4)
	react-uploady	2 / (1)	8 / (7)	2 / (1, 1)	1 / (1, -)	7 / (1, 6)
	react-uploady	2 / (1)	8 / (7)	2 / (1, 1)	1 / (1, -)	7 / (1, 6)
	react-uploady	3 / (2)	7 / (6)	3 / (1, 2)	1 / (1, -)	7 / (1, 6)
	react-rxjs	2 / (1)	4 / (3)	2 / (1, 1)	1 / (1, -)	4 / (1, 3)
	react-rxjs	2 / (1)	7 / (6)	2 / (1, 1)	1 / (1, -)	2 / (1, 1)
	react-rxjs	3 / (2)	3 / (2)	3 / (1, 2)	1 / (1, -)	3 / (1, 2)
	react-uploady	2 / (1)	10 / (9)	5 / (1, 4)	1 / (1, -)	8 / (1, 7)
	predictive-text	2 / (1)	8 / (7)	2 / (1, 1)	1 / (1, -)	7 / (1, 6)
	predictive-text	2 / (1)	8 / (7)	2 / (1, 1)	1 / (1, -)	7 / (1, 6)
	edelweiss-ui	2 / (1)	6 / (5)	2 / (1, 1)	1 / (1, -)	6 / (1, 5)
	cordless	2 / (1)	8 / (7)	2 / (1, 1)	1 / (1, -)	8 / (1, 7)
Average	-	2.2 / (1.2)	7.5 / (6.5)	2.9 / (1.2, 1.7)	1.2 (1.2, -)	6.3 / (1.2, 5.1)
Dom-caused	dotcom	2 / (1)	9 / (8)	2 / (1, 1)	1 / (1, -)	9 / (1, 8)
	jbrowse	2 / (1)	10 / (9)	2 / (1, 1)	1 / (1, -)	9 / (1, 8)
	jbrowse	2 / (1)	13 / (12)	3 / (2, 1)	2 / (2, -)	9 / (2, 7)
	jbrowse	2 / (1)	12 / (11)	3 / (2, 1)	2 / (2, -)	9 / (2, 7)
	jbrowse	2 / (1)	8 / (7)	2 / (1, 1)	1 / (1, -)	8 / (1, 7)
	jbrowse	2 / (1)	13 / (12)	2 / (1, 1)	1 / (1, -)	7 / (1, 6)
	jbrowse	2 / (1)	9 / (8)	2 / (1, 1)	1 / (1, -)	8 / (1, 7)
	codacollection	2 / (1)	8 / (7)	2 / (1, 1)	1 / (1, -)	8 / (1, 7)
	elm-select	2 / (1)	8 / (7)	2 / (1, 1)	1 / (1, -)	8 / (1, 7)
	flashmap	2 / (1)	13 / (12)	3 / (2, 1)	2 / (2, -)	9 / (2, 7)
	flashmap	2 / (1)	13 / (12)	3 / (2, 1)	2 / (2, -)	9 / (2, 7)
	flashmap	2 / (1)	13 / (12)	3 / (2, 1)	2 / (2, -)	9 / (2, 7)
	flashmap	2 / (1)	10 / (9)	2 / (1, 1)	1 / (1, -)	9 / (1, 8)
	coinscan-front	3 / (2)	10 / (9)	3 / (1, 2)	1 / (1, -)	10 / (1, 9)
	keptn	2 / (1)	6 / (5)	2 / (1, 1)	1 / (1, -)	6 / (1, 5)
	material-ui	2 / (1)	7 / (6)	2 / (1, 1)	1 / (1, -)	7 / (1, 6)
Average	-	2.1 / (1.1)	10.1 / (9.1)	2.4 / (1.3, 1.1)	1.3 / (1.3, -)	8.4 / (1.3, 7.1)
Multi-dom	client	2 / (1)	6 / (5)	2 / (1, 1)	1 / (1, -)	6 / (1, 5)
	preact-devtools	2 / (1)	8 / (7)	2 / (1, 1)	1 / (1, -)	7 / (1, 6)
Average	-	2.0 / (1.0)	7.0 / (6.0)	2.0 / (1.0, 1.0)	1.0 / (1.0, -)	6.5 /(1.0, 5.5)
Total	-	2.1 / (1.1)	8.8 / (7.8)	2.5 / (1.2, 1.3)	1.2 / (1.2, -)	7.4 (1.2, 6.2)

6.3 RQ3. Augmentation Contribution

Table 4 shows the impact of using the history-based augmentation in *TRaf*. For 31 flaky tests that we studied, *TRaf* could not identify any valid latency value for three cases, that flaked by DOM-related issues (*DOM-caused* and Multi-dom), without the aid of the augmentation. Upon manual inspection, we found that a relatively small number of test files were located under the directory with the flaky test file, and more importantly, none of the statements (one case) or only the fix-location (two cases) contained a wait time call. For such cases, *TRaf* without the history-based augmentation ($TRaf_{woa}$) inherently cannot localize any candidate wait time values, becoming *non-applicable*. The column *RR* in Table 4 compares the repair rate of *TRaf* between with and without

the history-based augmentation: with the history-based augmentation, *TRaf* successfully obtained wait time values that resolve async wait flakiness for all 31 flaky tests including the three that it ($TRaf_{woa}$) failed before.

The third and fourth columns of Table 4 present the differences in the wait time and test execution time ($diff_{WT}$ and $diff_{ET}$) between with and without the history-based augmentation for *TRaf*. These differences were calculated only for cases (28 out of 31) where *TRaf* found a valid time value without the augmentation, to investigate whether the augmentation leads to the finding of more *efficient* time values in addition to the improvement in the repair rate (*RR*). As shown in column '$TRaf^S - TRaf^S_{woa}$', the augmentation enables *TRaf* to find smaller latency values that still resolve the flakiness issue while reducing the test execution time compared to the case without the augmentation; for the tests flaked due to the *Time-caused* reasons, $TRaf^S$ selected a timeout on average 115.4 ms (11.8%) smaller than the ones chosen without the augmentation, reducing the test execution time by 48.9 ms (4.5%). Column '$TRaf^D_{FM} - TRaf^D_{FM,woa}$' demonstrates the impact of choosing more efficient timeout values on the subsequent dynamic optimization process by comparing the first successful refinement attempt between *TRaf* with ($TRaf^D_{FM}$) and without the history-based augmentation ($TRaf^D_{FM,woa}$). Overall, the timeout value of $TRaf^D_{FM}$ is, on average, 37.6 (ms) (5.0%) smaller than that of $TRaf^D_{FM,woa}$, reducing the execution time by 12.9 (ms) (1.3%) through the augmentation for the flaky tests in *Time-caused* category.

Answer to RQ3: The augmentation step improves the effectiveness of *TRaf*, allowing it to reach a 100% repair rate under 100 reruns. It also allows *TRaf* to find a more efficient wait time value that further reduces the test execution time.

Table 4. Comparison of repair rate (RR), average wait time (WT), and execution time (ET) between *with and without* history-based augmentation for static and dynamic *TRaf*. The differences in WT ($diff_{WT}$) and ET ($diff_{ET}$) are in *ms*. Negative *diff* means that the one with the augmentation outperforms the one without it. *RR* is computed using the entire set of studied flaky tests (n_{all}), and *diff* is computed only for cases (n) that can be repaired without the augmentation by *TRaf*.

		$TRaf^S$ - $TRaf^S_{woa}$		$TRaf^D_{FM}$ - $TRaf^D_{FM,woa}$			Repair Rate (*RR*)	
Root cause	n	$diff_{WT}$	$diff_{ET}$	$diff_{WT}$	$diff_{ET}$	n_{all}	*with aug*	*without aug*
Time-caused	13	-115.4	-48.9	-37.6	-12.9	13	100%	100%
Dom-caused	14	0	0	0	0	16	100%	87.5%
Multi-dom	1	0	0	0	0	2	100%	50%
Total	28	-53.5	-22.7	-17.4	-6	31	100%	90.3%

7 DISCUSSION

The prevalence of async wait flaky test and the impact of Time-based treatment. Asynchronous wait flaky test, shortly async wait test, is one of the most common types of flaky tests found in open-source and proprietary projects [12, 17, 27, 28, 33, 39]. Luo et al. studied 201 flaky test-fixing commits from 51 open-source projects to identify the common root causes and fix strategies adopted in practice. For these 201 commits, 74 (36.8%) were related to fixing async wait flaky tests [27]; among these 74 flaky tests, 39 adopted a fixing strategy of *adjusting the wait time or timeout*, which we refer to as *Time-based* fixes in this paper. Similar statistics were observed in the study of six large-scale proprietary projects at Microsoft [17]: out of 134 fixed flaky tests studied by the authors, 87 fixes were related to asynchronous method calls, and 31% of them involved increasing the wait time, revealing developers' preference toward a simple time-based fix strategy. Recently, Romano et al. conducted

an empirical study of flaky tests in UI testing, reaffirming the previous findings about async wait flakiness in the new testing domain [39].

In this paper, we also studied flaky tests in web front-end testing but mainly focused on those flaked by asynchronous wait. The primary objective of the study is to investigate the prevalence and effectiveness of time-based treatment on this specific type of flaky test. Our initial investigation of existing fixes for async wait flakiness (Section 3) reveals a strong inclination among developers for time-based solutions for async wait flaky tests regardless of their root causes. This suggests that, in practice, developers tend to prioritize efficiency, opting for quick but partial fixes as long as they are sufficient to ensure continuous delivery of stable software.

Time-based treatment for async wait flakiness by *TRaf*. Given that the time-based strategy of adding or adjusting timeout does not require developers to identify the root cause of asynchronous flakiness, they are commonly used as hotfixes to mitigate the flakiness [12, 17, 27, 28, 33, 39]. Nevertheless, this usability often comes at the cost of increased test execution time. The main idea of this time-based strategy is to wait long enough for an asynchronous call to complete. Hence, choosing a proper timeout to resolve the flakiness is challenging without knowing what is *"enough"*, which is often the case. As a result, in their efforts to mitigate flakiness, developers tend to set a large new timeout, resulting in the increase in the test expenses. Given that regression testing has become a standard development practice, this increase in cost, however small it is per test, may easily come back as a huge burden in testing. *TRaf* tackles this issue by finding an efficient time to wait for an asynchronous call, using code similarity and past changes, assuming that the clues to the right time value exist in the same codebase of present and past near the code with a similar context to a fix-location. The experimental results with 31 async wait flaky tests show that the time values from the lines with similar context can address the flakiness more efficiently than the time values in the developer's fixes. *TRaf* achieved this performance almost without any test rerun involved, successfully ranking the proper wait time at the top of its candidate list in most cases (25 out of 31 cases); within a few trials of dynamic refinement, i.e., at most two, *TRaf* further refined the localized values. This low fixing overhead of *TRaf* demonstrates its potential for on-the-fly handling of async wait flakiness, making it particularly useful in hotfix scenarios. Considering the prevalence of async wait flakiness in practice, though its effectiveness may vary depending on codebase characteristics and environmental factors, *TRaf* can offer a valuable solution for enhancing the reliability and efficiency of web front-end testing processes.

Developer response to the fix of *TRaf*. We sent 16 pull requests to the developers, each fixing one test[8]. As part of a pull request, we provide both the proposed wait time values and the statistical and experimental results we gathered by running *TRaf* on the tests. So far, the developers have accepted three pull requests, six pending, and seven rejected.

For these seven rejected pull requests, developers preferred using relatively large latency values to reduce the risk of flakiness issues reoccurring, despite the resulting increased testing cost. For instance, regarding our pull request to *react-uploady*[9], the developer responded that simply increasing the wait time without caring about whether it results in increased testing cost has been their go-to tactic for react-uploady; this is because local tests passing without any issue locally often fails in CI unexpectedly. Similarly, for the rejected pull request to *jbrowse-components*[10], developers reasoned that due to the complexity of the projects and tests, they are compelled to increase the timeout as they are uncertain about how to optimize it and lack the necessary time to do it. From these findings, we conjecture that the preference for longer wait times is likely due to the complexity and scale of the projects, as well as the unpredictable behavior in CI. For those we obtained positive responses, developers merged our wait time values into the main branch. These positive responses demonstrate the practical value of *TRaf* in leveraging optimal wait times to reduce flakiness and test execution time.

[8]The details are provided in the pull request column of our dataset, https://doi.org/10.5281/zenodo.10728548
[9]https://github.com/rpldy/react-uploady/pull/668
[10]https://github.com/GMOD/jbrowse-components/pull/4227

Table 5. Comparison of wait time and test execution time between developers' fixes, *FaTB*, and our approach, *TRaf*, on 8GB machine. *Avg* refers to the average within each root cause, and *Total* denotes the average without differentiating root causes. *optimal* denotes the optimal wait time value to which *FaTB* and dynamic *TRaf* ($TRaf^D$) eventually converge. *dev* denotes the wait and execution times under the developer's chosen values. *p-value* is to check statistical significance; *p-value* smaller than 0.05 are written in bold, and *<0.001* denotes the p-values less than *0.001*. Those obtaining the same average execution time as the baseline are highlighted in light orange in addition to those bold by outperforming the baselines. $diff_{TRaf^S}$ and $diff_{optimal}$ denote the test execution time differences between 8GB and 32 GB machines at $TRaf^S$ and *optimal* ($TRaf^D$), respectively; the negatives mean that the 32GB machine performs faster than the 8GB machine.

Root cause	project	$TRaf^S$ Pass/Fail	$TRaf^D$ Pass/Fail	Execution Time (ms) Baseline dev	Baseline $FaTB_{FM}$	$TRaf^S$	p-value vs dev	$diff_{TRaf^S}$ (32-8)	$TRaf^D_{FM}$	p-value vs $FaTB_{FM}$	optimal	$diff_{optimal}$ (32-8)
Time	puppeteer	100/0	100/0	3123.3	2455.8	1909.9	**<0.001**	-513.4	1745.0	**<0.001**	1621.7	-302.3
	puppeteer	100/0	100/0	3380.9	2627.2	1460.9	**<0.001**	-624.7	1162.9	**<0.001**	1259.5	-455.9
	r-uploady	100/0	100/0	4226.5	3723.4	3172.6	**<0.001**	-2154.5	1689.2	**<0.001**	2451.7	-2054.0
	r-uploady	100/0	100/0	4667.2	3306.5	3453.7	**<0.001**	-2453.5	1895.3	**<0.001**	2548.9	-2143.4
	r-uploady	100/0	100/0	6172.9	4969.4	5496.2	**<0.001**	-2391.2	4849.7	**<0.001**	5079.4	-2212.5
	r-uploady	100/0	100/0	5288.4	5090.3	5250.7	**<0.001**	-3576.1	5084.6	**<0.001**	5085.5	-3477.8
	react-rxjs	100/0	100/0	150.4	144.9	145.6	0.475	-18.9	143.5	0.232	130.4	-4.4
	react-rxjs	100/0	100/0	137.3	130.9	129.5	0.196	-17.7	111.4	**0.024**	104.3	-4.9
	react-rxjs	100/0	100/0	156.9	149.6	156.0	0.502	-21.9	149.5	0.522	141.8	-14.1
	predictive	100/0	100/0	6325.7	5543.4	5520.3	**<0.001**	-4328.6	5369.8	**<0.001**	3884.0	-2795.4
	predictive	100/0	100/0	5967.1	5733.2	5157.5	**<0.001**	-4083.3	5383.6	**<0.001**	3857.1	-2883.0
	edelweiss	100/0	100/0	1251.3	1196.4	1208.7	0.483	-222.1	1086.2	0.454	1057.8	-225.4
	cordless	100/0	100/0	1225.7	1130.5	1192.3	0.389	-254.9	1129.8	0.411	866.1	-105.9
Avg.	-	-	-	3236.4	2784.7	2634.9	-	-1589.3	2292.3	-	2160.6	-1283.0
Dom	dotcom	100/0	100/0	6568.9	6502.8	6552.5	0.501	-4151.3	5914.9	**<0.001**	5782.6	-3648.6
	jbrowse	100/0	100/0	6094.5	5897.5	5807.2	**<0.001**	-3564.2	5800.3	0.104	5676.1	-3471.8
	jbrowse	100/0	100/0	3226.9	3290.0	3202.9	**<0.001**	-2146.1	3162.7	**0.034**	3087.3	-2045.7
	jbrowse	100/0	100/0	3561.3	3471.6	3214.6	**<0.001**	-2374.9	3140.9	**<0.001**	3157.1	-2340.1
	jbrowse	100/0	100/0	5309.5	5162.3	5233.2	**<0.001**	-3556.0	5162.0	0.437	5016.9	-3377.1
	jbrowse	100/0	100/0	3412.8	3385.1	3177.9	**<0.001**	-1695.7	3144.5	**<0.001**	3071.4	-1610.7
	jbrowse	100/0	100/0	3099.6	2994.7	2976.6	**<0.001**	-1988.1	2901.6	**<0.001**	2894.9	-1978.0
	coda	100/0	100/0	13903.2	13694.5	13855.6	**<0.001**	-6034.9	13700.2	**<0.001**	9616.3	-1953.5
	elm-select	100/0	100/0	505.6	391.0	486.0	**0.041**	-257.2	386.5	0.498	421.1	-278.4
	flashmap	100/0	100/0	3862.9	3729.8	3537.2	**<0.001**	-2626.7	3414.3	**<0.001**	3381.6	-2501.4
	flashmap	100/0	100/0	3299.1	3304.6	3241.3	**<0.001**	-2370.7	2976.4	**<0.001**	3087.8	-2227.2
	flashmap	100/0	100/0	2870.7	2785.1	2831.5	0.224	-2185.5	2811.0	**<0.001**	2712.5	-2072.0
	flashmap	100/0	100/0	6302.5	6233.6	6216.1	**<0.001**	-3860.3	5995.3	**<0.001**	5957.4	-3760.8
	coinscan	100/0	100/0	4550.2	4384.0	4539.5	0.489	-1115.5	4391.2	0.439	4407.9	-1165.9
	keptn	100/0	100/0	1809.4	1174.6	1801.0	0.500	-18.0	1070.8	**<0.001**	933.3	-170.9
	material	100/0	97/3	844.6	789.0	842.5	0.471	-429.5	789.8	0.436	775.0	-395.0
Avg.	-	-	-	4326.4	4199.45	4219.4	-	-2398.4	4047.5	-	3748.7	-2062.3
Multi	client	100/0	100/0	433.9	403.6	433.8	0.231	-165.3	403.2	0.481	314.5	-97.5
	preact	100/0	100/0	4970.8	4292.5	4285.6	**<0.001**	-1113.6	3955.6	**<0.001**	2929.6	-74.6
Avg.	-	-	-	2702.4	2365.1	2349.7	-	-639.5	2179.4	-	1622.1	-86.0
Total	-	-	-	3764.5	3487.8	3434.4	-	-1945.6	3191.0	-	2945.5	-1608.0

The effect of the machine environment on the proposed patches of *TRaf*. The effect of the test environment on test execution behavior is a crucial aspect of software testing, with various factors contributing to the overall efficiency and reliability of the testing process. However, in certain scenarios, the machine environment may not have a significant impact on test behavior, particularly when the tests are designed to be lightweight or executed in a controlled environment. For instance, unit tests, which typically focus on isolated components of the codebase and mock the dependencies, may exhibit consistent behavior across different machine environments. Similarly, certain types of automated tests, such as integration tests that run under certain test frameworks, may experience

minimal variability in execution behavior. Hence, to further investigate the impact of the test environment on the patches from our approach, we utilized the waiting time obtained through *TRaf* on an additional machine to execute the test and verify the effectiveness of our approach in diverse machine environments. The configuration of the machine is an Apple processor 2.3 GHz Dual-core Intel Core i5 and 8 GB of RAM.

Table 5 demonstrates the comparison of test execution time between developers' fixes, FaTB, and our approach, *TRaf*, using the same wait time values on 8GB and 32GB machines. With the 8GB machine, by default $TRaf^S$ outperformed developer-written patches (*dev*), as shown in the final row *Total*. Specifically, *TRaf* reduces test execution time by an average of 8.8% (330.1 ms) compared to the times obtained with the wait times later chosen by developers; with dynamic tuning, *TRaf* achieves further reductions in test execution time. When comparing FaTB_{FM} and $TRaf^D_{FM}$, *TRaf* reduces execution time by 8.5% (296.8 ms) on average. By setting the dynamic tunning to tune the wait time to the maximum (*optimal*), test execution time was reduced by 21.8% (819 ms) on average compared to the patches written by developers.

The columns $Pass/Fail$ in the Table 2 and Table 5 compare the flaky-test rate between 32 GB and 8 GB machines. The results show that under the 8 GB environment, for default static mode, $TRaf^S$, all 100 runs are passed; for the $TRaf^D$ dynamic mode, one of the cases got a result of 97 passes and 3 fails (highlighted in red). The reason for this outcome – three failures – might be due to a specific mechanism employed in a test framework and the complexity of the project, resulting in insufficient wait times to invoke the function. During our investigation, however, we were unable to find unique factors or patterns that clearly explain the failures. Nevertheless, we suspect these failures to be related to the test framework. The test framework used by the *material* project is Mocha. Unlike Jest and Cypress, which run as independent frameworks, Mocha requires more configuration items and depends on more libraries. From this, we conjecture that different test frameworks may operate differently on different device environments, causing these failures on the 8GB device.

diff columns in Table 5 show the differences in the execution time with the same wait time values between 32GB machine and 8GB machine environments. Often, the execution time at the 8GB machine was longer than the time at the 32GB machine, with an average increase of 56.7% at default static mode ($diff_{TRaf^S}$) and 54.6% when using dynamic tuning to the maximum ($diff_{optimal}$. Nevertheless, regardless of these increases, in most cases, *TRaf* was able to find efficient wait times that work with both 8GB and 32GB machine environments across repeated 100 runs. Based on these results, we posit that although the extent of reduction in test execution time may vary across different machines, *TRaf* is capable of identifying wait times that perform consistently well across diverse environments by statically finding wait times that are more effective than those later selected by developers to address test flakiness.

8 THREATS TO VALIDITY

Internal validity relates to the validity of our evaluation, i.e., whether the evaluation can support what we claim. To avoid introducing biases into the studied flaky tests in web testing, we systematically collected 49 flaky tests from 26 open-source projects containing the most employed web application testing frameworks. We evaluate the generated repair by monitoring the pass-and-fail state of a given test across its repeated runs. Previous studies have shown that 100 reruns are typically enough to determine whether a test is flaky or not [4, 5, 16, 17, 34]. Hence, to ensure the validity of *TRaf* 's fixes for Async Wait flaky tests, we rerun tests 100 times.

External validity refers to the factors that impact the generalizability of the conclusion. While our target is restricted to projects that can run locally, we improved generalizability by including diverse projects with different testing frameworks. This study focuses on examining projects written in JavaScript, a dominant language in web development. *TRaf* currently supports only JavaScript and its testing framework (i.e., Cypress, Jest, and others). Nevertheless, the main idea of *TRaf* is independent of the language. We plan to extend *TRaf* to address async wait flakiness in projects written in other popular programming languages, such as Java and Python. Currently,

our dataset is limited by the commits containing flakiness-related keywords; to ensure comprehensive coverage of the flaky test collection, we conducted a thorough search for diverse and widely studied keywords related to flakiness from previous studies [9, 12, 27, 39]. Our study focuses on DOM and time factors contributing to asynchronous wait flakiness in front-end web testing. While network or API requests can also cause flakiness, these issues are generally related to back-end processes, such as server data requests. Since our primary interest is the front end, we have limited our study to examining DOM and time factors. Future research will expand to address flakiness in the back-end, including issues caused by network or APIs, particularly focusing on those that can be mitigated by adjusting wait-time values.

Threats to construct validity may exist in our data collection of ground-truth. To minimize the risk of incorrect labeling, we used keywords frequently adopted to detect async wait flakiness and followed the collection procedure of the previous work [9]. We further manually inspected the commits to verify their relevance to the async wait flaky test and reran both flaky tests and their patches 100 times for validity, similar to previous studies of flaky tests [12, 39]. In addition, the two authors first independently performed three rounds of inspections for categorizing async wait flaky tests, examining commit messages, source and test code, error messages, and fixes, and derived a classification from the discussion. The machine we use in our study poses another potential threat to our findings and results. We have only used two different configurations of machines when we rerun the projects and execute the tests. Although it is impossible to have the same systems, we strictly control the libraries and dependencies versions to reduce external differences and compatibility issues. Meanwhile, a comparison of the fixed time values set by the developer and the fixed values obtained by *TRaf* indicates a significant difference, far beyond the scope of the influence of the external environment. Based on the common trend of the results from different projects and flaky test failures, we indicated that *TRaf* can still improve wait time in different test environments. Another potential limitation of our approach is that our experiments are mainly conducted in the local environment, considering the complexity of running different projects and reproducing flakiness. Nevertheless, it is possible that our *TRaf* tool can be executed on any other machine or utilized in the Continuous Integration environment. First, our approach is not tied to our local environment, and it can be run in other environments as well. Second, similar to conducting tests on our experimental machine, the execution process is the same. Therefore, based on how the method is designed and how it works, we believe that our approach would also be helpful in CI or other machine environments.

9 RELATED WORK

Flaky tests exhibit non-deterministic and unpredictable behaviors that require to be addressed differently. Many prior studies have attempted to categorize flaky tests based on their prevalence in software testing [9, 12, 16, 18, 27, 33, 37, 39]. Luo et al. conducted an extensive study on flaky tests, studying 201 commits from 51 open-source projects, and investigated what triggers flaky tests, how they manifest, and how they can be fixed and grouped based on their findings [27]. Lam et al. studied the logs of flaky tests in an industrial setting and found that while the number of builds failed by flaky tests might be substantial, the number of distinct tests is rather limited [16]. Thus, they proposed RootFinder, a framework to identify the root cause of a given flaky test by analyzing the logs of failing and passing tests. A recent study with proprietary Microsoft projects confirmed that existing findings on flaky tests from open-source projects, including common causes and fix strategies, generalize to projects with a different level of accessibility [17]; in this work, the authors observed that developers often adapt the timeout to alleviate flakiness due to asynchronous calls.

Web testing involves various factors that are inherently unpredictable, such as user interactions and network delays, and are thereby susceptible to flakiness [12, 39]. Romano et al. performed an empirical study on flaky tests in UI-testing in web and Android environments [39]; the study revealed that flaky tests in UI-testing also have similar categories of root causes and fix strategies in traditional unit testing. All these prior studies, regardless

of the different subjects and environments, agree that flaky tests with different causes require different fixing strategies. In this study, we focus on the flaky tests caused by asynchronous waits, exploring their more detailed root causes and common fix strategies of developers in web front-end testing.

Previous empirical studies on flaky tests confirmed that developers addressed flaky tests differently depending on their causes. Hence, existing works on automated flaky test repair focused on repairing a certain type of flaky tests [6, 17, 25, 32, 41, 42, 45]. Shi et al. proposed iFixFlakies, a tool to automatically fix Test-Order Dependency flaky tests that non-deterministically fail and pass depending on the test execution order [42]. iFixFlakies resolves the order-dependent flaky tests by calling helper functions that recover shared states before pollution. Recently, Li et al. introduced ODRepair, which addresses the weakness in iFixFlakies that assumes the existence of helper functions [25]. ODRepair generates the code to clean up polluted states of tests rather than looking for helpers.

Async wait flaky tests are one of the most common flaky tests in practice [11, 33]. While relatively small in number compared to repair works on other types of flaky tests, recent studies demonstrated the potential of automated repair of async wait test flakiness. Lam et al. proposed the FaTB that dynamically improves the wait time selected by developers to reduce the test execution time [17]. However, FaTB is to refine the time values in developers' fixes, whereas *TRaf* is to automatically address flakiness before developers resolve them manually. TRaf is effective and lightweight, especially *TRaf*^{S}. Hence, it can be used as hotfixes to ensure continuous delivery of software. More importantly, *TRaf* identifies better time value candidates without any interpretation by developers. While the saving over FaTB might be small, the value is that *TRaf* does not wait for developers to address the flakiness to obtain the initial time value: it can be used as soon as the flakiness occurs at timeout or waitFor statements, which can be detected by simple pattern matching.

In the area of web testing, Olianas et al. introduced an approach that automatically repairs async wait flaky tests by replacing thread sleeps with explicit waits in an E2E Selenium WebDriver test suite [32]. Our approach *TRaf* shares similarities with previous studies on async wait flaky test repair. However, *TRaf* differs from the existing methods in that it does not need to know the developers' wait time in advance and can efficiently suggest a new time for waiting by adapting the plastic surgery hypothesis. By simply adapting the wait time for repair instead of replacing it with an explicit wait for a certain element, *TRaf* requires far less time to address the flakiness.

10 CONCLUSION

In this paper, we conducted an empirical study on asynchronous wait flakiness in web front-end testing. We investigated 49 reproducible asynchronous wait flaky tests collected from 26 JavaScript web projects. We divided these tests according to the way developers fixed them, confirming that adding or increasing the wait time is the most common repair practice in web front-end applications for this type of flaky test. Hence, we proposed *TRaf*, a novel time-based repair technique that automatically addresses asynchronous wait flakiness in web front-end testing. *TRaf* leverages code similarity to flaky lines and the version history of code under test to obtain efficient time values to alleviate test flakiness while minimizing the test execution time. Our empirical analysis demonstrates that *TRaf* consistently outperforms developer-written fixes by identifying more efficient wait time values, resulting in an average reduction of 11.1% in test execution time; notably, 77% of these time values were ranked at the top among the candidate wait times obtained by *TRaf*. Through the additional dynamic tuning, *TRaf* achieved an even more substantial reduction in test execution time, reaching an average of 16.8% decrease within two trials and up to 20.2% compared to developer-written patches. By comparing with an existing approach of refining developer-written fixes, we further demonstrate that *TRaf* can obtain comparable and often better performance to this post-processing approach right upon the detection of asynchronous wait flakiness, highlighting its usability.

REFERENCES

[1] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The Plastic Surgery Hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 306–317. https://doi.org/10.1145/2635868.2635898

[2] Scott Chacon and Ben Straub. 2014. *Pro git.* Apress.

[3] Yang Chen, Alperen Yildiz, Darko Marinov, and Reyhaneh Jabbarvand. 2023. Transforming test suites into croissants. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1080–1092.

[4] Zhen Dong, Abhishek Tiwari, Xiao Liang Yu, and Abhik Roychoudhury. 2021. Flaky test detection in Android via event order exploration. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 367–378.

[5] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting flaky tests in probabilistic and machine learning applications. *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2020), 211–224. https://doi.org/10.1145/3395363.3397366

[6] Saikat Dutta, August Shi, and Sasa Misailovic. 2021. FLEX: Fixing Flaky Tests in Machine Learning Projects by Updating Assertion Bounds. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 603–614. https://doi.org/10.1145/3468264.3468615

[7] Dennis F Galletta, Raymond Henry, Scott McCoy, and Peter Polak. 2004. Web site delays: How tolerant are users? *Journal of the Association for Information Systems* 5, 1 (2004), 1–28.

[8] Kening Gao, Bin Zhang, Yin Zhang, Hongru Wei, and Anxiang Ma. 2008. Study on Semantic Representation of Web Information Based on Repeating Patterns. In *2008 Fifth International Conference on Fuzzy Systems and Knowledge Discovery*, Vol. 4. IEEE, 482–486.

[9] Martin Gruber, Stephan Lukasczyk, Florian Krois, and Gordon Fraser. 2021. An Empirical Study of Flaky Tests in Python. *Proceedings - 2021 IEEE 14th International Conference on Software Testing, Verification and Validation, ICST 2021* (2021), 148–158. https://doi.org/10.1109/ICST49551.2021.00026

[10] 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.

[11] Sarra Habchi, Guillaume Haben, Jeongju Sohn, Adriano Franci, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2022. What made this test flake? pinpointing classes responsible for test flakiness. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 352–363.

[12] Negar Hashemi, Amjed Tahir, and Shawn Rasheed. 2022. An Empirical Study of Flaky Tests in JavaScript. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 24–34. https://doi.org/10.1109/ICSME55016.2022.00011

[13] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 426–437.

[14] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. 2007. Predicting Faults from Cached History. In *29th International Conference on Software Engineering (ICSE'07)*. 489–498. https://doi.org/10.1109/ICSE.2007.66

[15] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the Cost of Regression Testing in Practice: A Study of Java Projects Using Continuous Integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 821–830. https://doi.org/10.1145/3106237.3106288

[16] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 101–111. https://doi.org/10.1145/3293882.3330570

[17] Wing Lam, Kivanc Muslu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. *Proceedings - International Conference on Software Engineering* (2020), 1471–1482. https://doi.org/10.1145/3377811.3381749

[18] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. 2020. Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 403–413.

[19] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29.

[20] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. 2015. Information Retrieval and Spectrum Based Bug Localization: Better Together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 579–590. https://doi.org/10.1145/2786805.2786880

[21] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[22] Claire Leong, Abhayendra Singh, John Micco, Mike Papadakis, and Yves le traon. 2019. Assessing Transition-based Test Selection Algorithms at Google. In *International Conference on Software Engineering*.

[23] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2014. Visual vs. DOM-based web locators: An empirical study. In *Web Engineering: 14th International Conference, ICWE 2014, Toulouse, France, July 1-4, 2014. Proceedings 14*. Springer, 322–340.

[24] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2015. Automated generation of visual web tests from DOM-based web tests. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. 775–782.

[25] Chengpeng Li, Chenguang Zhu, Wenxi Wang, and August Shi. 2022. Repairing Order-Dependent Flaky Tests via Test Generation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1881–1892. https://doi.org/10.1145/3510003.3510173

[26] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/3293882.3330577

[27] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Vol. 16-21-November-2014. 643–653. https://doi.org/10.1145/2635868.2635920

[28] Jean Malm, Adnan Causevic, Björn Lisper, and Sigrid Eldh. 2020. Automated analysis of flakiness-mitigating delays. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*. 81–84.

[29] Matias Martinez, Westley Weimer, and Martin Monperrus. 2014. Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 492–495. https://doi.org/10.1145/2591062.2591114

[30] Maximiliano Agustín Mascheroni and Emanuel Irrazábal. 2018. Identifying key success factors in stopping flaky tests in automated REST service testing. *Journal of Computer Science & Technology* 18 (2018).

[31] John Micco. 2017. The State of Continuous Integration Testing Google.

[32] 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* (2022), 1–33.

[33] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 17 (oct 2021), 74 pages. https://doi.org/10.1145/3476105

[34] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the vocabulary of flaky tests?. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 492–502.

[35] 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.

[36] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. 2011. BugCache for Inspections: Hit or Miss?. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 322–331. https://doi.org/10.1145/2025113.2025157

[37] M. Tajmilur Rahman and Peter C. Rigby. 2018. The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds. *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), 857–862. https://doi.org/10.1145/3236024.3275529

[38] Julian Risch and Ralf Krestel. 2019. Measuring and facilitating data repeatability in web science. *Datenbank-Spektrum* 19 (2019), 117–126.

[39] 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)*. 1585–1597. https://doi.org/10.1109/ICSE43902.2021.00141

[40] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 345–355. https://doi.org/10.1109/ASE.2013.6693093

[41] August Shi, Jonathan Bell, and Darko Marinov. 2019. Mitigating the effects of flaky tests on mutation testing. *ISSTA 2019 - Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2019), 296–306. https://doi.org/10.1145/3293882.3330568

[42] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 545–555.

[43] Jeongju Sohn, Yasutaka Kamei, Shane McIntosh, and Shin Yoo. 2021. Leveraging Fault Localisation to Enhance Defect Prediction. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 284–294. https://doi.org/10.1109/SANER50967.2021.00034

[44] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual web test repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 503–514.

[45] Anjiang Wei, Pu Yi, Zhengxi Li, Tao Xie, Darko Marinov, and Wing Lam. 2022. Preempting flaky tests via non-idempotent-outcome tests. In *Proceedings of the 44th International Conference on Software Engineering*. 1730–1742.

[46] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

[47] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating Bugs from Software Changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 262–273. https://doi.org/10.1145/2970276.2970359

[48] Rahulkrishna Yandrapally and Ali Mesbah. 2021. Mutation analysis for assessing end-to-end web tests. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 183–194.

[49] Shin Yoo and Mark Harman. 2012. Regression Testing Minimisation, Selection and Prioritisation: A Survey. *Software Testing, Verification, and Reliability* 22, 2 (March 2012), 67–120.