# Who's to Blame? Rethinking the Brittleness of Automated Web GUI Testing from a Pragmatic Perspective

Haonan Zhang*, Kundi Yao*, Zishuo Ding†, Lizhi Liao‡, Weiyi Shang*
*University of Waterloo, †The Hong Kong University of Science and Technology (Guangzhou), ‡Memorial University of Newfoundland
{haonan.zhang, kundi.yao}@uwaterloo.ca, zishuoding@hkust-gz.edu.cn, lizhi.liao@mun.ca, wshang@uwaterloo.ca

***Abstract*—Automated web GUI testing is important for software quality, however, its effectiveness is often undermined by test case brittleness, especially in continuously evolving real-world applications. In this experience paper, we pragmatically investigate the root causes of brittleness. We first analyze why legacy test cases, derived from the Mind2Web dataset, fail when executed on current web application versions. Our findings reveal that brittleness stems from multifaceted factors, including test script design, web application complexity, and automation framework limitations. A longitudinal study further shows that 81.7% of repaired tests break again within six months, primarily due to similar recurring issues, highlighting the persistent nature of brittleness. We further demonstrate that Large Language Models, when provided with human-like diagnostic context, can successfully repair a substantial portion of these brittle tests, though human expertise remains important for more complex scenarios. Our findings emphasize that brittleness is a multi-faceted problem requiring collaboration between different parts involved in the automation testing.**

***Index Terms*—Experience report, web GUI testing, test automation, test evolution and maintenance**

## I. Introduction

Automated web Graphic User Interface (GUI) testing is a widely adopted method to reduce the intensive labor and considerable time required by manual testing [1], [2]. However, a persistent challenge in this domain is the brittleness of automated test cases [3]. Brittleness refers to the tendency of test cases to fail when the web application under test changes. Such changes can include updates to its structure or layout, or they can arise from external conditions like network fluctuations, server performance issues, or other environmental variations. This brittleness often transforms previously reliable test cases into fragile, outdated artifacts that demand frequent and expensive maintenance [4]–[6]. Over time, these maintenance demands grow to a point where many organizations opt to abandon and rewrite their test suites entirely [7], thereby diminishing the efficiency and dependability of automated testing. The tight coupling between automated test cases and the web application, along with the dynamic and adaptive nature of modern web applications, intensifies this problem [3], [8]. Even small alterations can disrupt test execution, making brittleness a critical barrier to achieving sustainable and effective automated web GUI testing.

To tackle brittleness, it is essential to systematically identify and understand its root causes in practical, real-world settings. However, there is a gap between state-of-the-art research concerning automated web GUI testing and the realities of testing widely used, real-world, and dynamic web applications [9], [10]. Existing research often lacks a pragmatic perspective when studying challenges and solutions directly relevant to testing the diverse and dynamically evolving web applications found in real-world environments. For instance, some findings are derived from static analyses of bug reports or issue discussions, potentially overlooking issues that only manifest during live test execution on actively changing applications [11], [12]. Other studies that do involve test execution often rely on some specific open source projects [6] or only examine specific industrial systems [8], [13]–[15], which can limit the broad applicability of findings. Furthermore, many of these open source subjects used in previous studies are static and relatively simple [9], [16], or, even if extensive, are often deployed in controlled local environments for experiments [17]–[19]. Such experimental setups cannot sufficiently reflect the true complexities of real-world web applications built for large user volumes and continuous updates [9], [16]. Consequently, these approaches might not reveal critical forms of brittleness, especially those arising from dynamic content rendering, asynchronous operations, and complex user interface interactions common in large-scale, ever-changing web applications.

The limited scope of existing studies curtails their practical effect since their insights often fail to capture the complex, adaptive requirements of automated web GUI testing on modern web applications that receive frequent updates. As a result, despite various proposed solutions to address difficulties in automated web GUI testing [17]–[29], these issues continue, and some of them have even persisted for nearly two decades [4]. The central problem is that the solutions derived from root causes identified by prior studies are often partial or temporary fixes. They do not adequately identify and address the fundamental issues contributing to brittleness in actual web GUI testing environments. Uncovering these deep-seated problems and understanding their true nature requires moving beyond narrowly focused experiments or analyses of static artifacts towards investigations grounded in the dynamic complexities of real-world, evolving web applications.

To facilitate the development of effective, long-lasting strategies for improving the reliability and adaptability of automated web GUI testing in diverse web application settings, a

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00251

broader, more pragmatic study of the root causes of automated web GUI testing brittleness is needed. In this paper, we present an **experience report** of conducting automated GUI testing for real-world web applications. We intend to identify the root causes of brittleness associated with automation testing on modern web applications and understand how to address these effectively. Specifically, we use test cases derived from the Mind2Web dataset [30], which contains detailed instructions to complete tasks on real websites. To support our study, we developed a tool that automatically transforms these task instructions into executable test cases. Since these task instructions are based on older versions of websites, our first phase of analysis involves executing the generated test cases on the latest versions of the corresponding web applications. We then analyze the resulting failures to uncover the root causes of why these legacy test cases break, systematically categorizing these brittleness issues and identifying solutions. Following this, our second phase involves a deeper analysis to understand recurrent issues and effective repair strategies pertinent to a continuous evolution cycle, aiming to identify patterns of failure and solution types that can inform more resilient, long-term testing practices.

During manual repair, we observe that diagnosis relies on the current application state and its differences from the old test case. This contrasts with some automated repair tools [18], [19], [24], [25] that require access to old application versions for comparative analysis and often focus narrowly on locator issues, while actual brittleness causes are more diverse. The effort involved in manually addressing these varied issues led us to investigate whether Large Language Models (LLMs) could assist. We explore whether LLMs can emulate human-like diagnostic reasoning to suggest fixes using only the current execution context, and then evaluate their proposed solutions.

To guide our investigation, we aim to address the following research questions:

**RQ1: What causes legacy web GUI tests to break on real-world applications?** As applications evolve, understanding why these older tests initially fail is important. We therefore investigate the primary root causes of such brittleness to identify effective ways to address them.

**RQ2: How does web GUI test brittleness persist as applications continuously evolve?** Building on initial findings, we examine recurring brittleness patterns to uncover generalizable strategies for sustained test case resilience over time.

**RQ3: Can LLMs assist in repairing brittle web GUI tests by emulating human approaches?** The manual repair of brittle tests is often time-intensive. We therefore investigate whether LLMs can effectively replicate this human-like repair strategy and evaluate their fixes.

This paper makes the following contributions:

- A collection of test cases for real-world web applications, which can serve as a dataset for future research.
- An empirical characterization of root causes for legacy web GUI test brittleness on current applications.
- An analysis of recurring brittleness patterns observed in evolving web applications, shedding light on strategies for more sustainable test maintenance.
- An evaluation of LLMs for assisting web GUI test repair by emulating human-like diagnostic reasoning, assessing their effectiveness for multifaceted brittleness issues.

**Paper organization.** Section II presents the background and motivation of our study. Section III introduces the setup of our study. Section IV presents the results of our study. Section V presents the lessons learned from our study. Section VI discusses the threats to the validity of our results. Section VII presents the related work to our study. Finally, section VIII concludes this paper.

## II. Background and Motivation

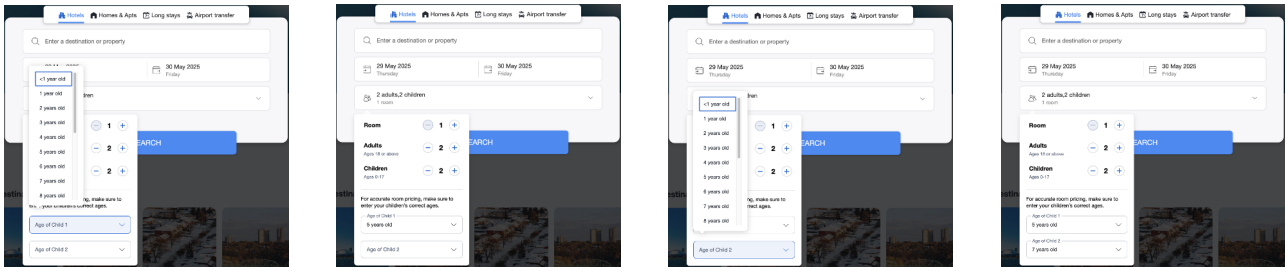
This section first provides a brief overview of automated web GUI testing and then presents a motivating example to illustrate the nuanced brittleness challenges that this paper aims to uncover, particularly those arising from the highly dynamic nature of modern real-world web applications.

```
def test_agoda():
    driver.get('https://agoda.com')
    ...
    # Click dropdown for Child 1's age
    driver.find_element(By.ID, "age-dropdown-1").click()
    # Select '5 years old' for Child 1
    driver.find_element(By.ID, "5-years-old").click()
    # Click dropdown for Child 2's age
    driver.find_element(By.ID, "age-dropdown-2").click()
    # Select '7 years old' for Child 2
    driver.find_element(By.ID, "7-years-old").click()
    ...
```

Listing 1: Test code of selecting the ages of children on Agoda

### A. Automated web GUI testing

Automated web GUI testing involves executing predefined test scripts that simulate user interactions with a web application's GUI. The goal is to verify that the application behaves as expected across different functionalities, browsers, and devices [31]. The code snippet in Listing 1 simulates the interaction process depicted in Figure 1. As shown in Listing 1, a typical automated test script, often written using frameworks like Selenium [32], consists of a sequence of operations. Each operation usually involves two main steps: **Locating a web element:** The test script first needs to find a specific GUI element on the web page (e.g., a button, input field, or link). This is commonly done using various locator strategies such as element ID, name, CSS selectors, or XPath expressions. **Performing an action:** Once the element is located, the script performs an action on it, such as clicking a button, typing text into a field, or selecting an option from a dropdown. While straightforward in principle, this process is highly susceptible to brittleness in practice. Changes to the web application, however minor, can invalidate the process of locating a web element and performing an action, causing tests to fail even if the underlying application logic remains correct. This brittleness leads to significant maintenance overhead and is a primary focus of our research.



(a) Select age of child 1 (b) Age of child 1 selected (c) Select age of child 2 (d) Age of child 2 selected

Fig. 1: Process of selecting the ages of children on Agoda

### B. Motivation: The elusive nature of brittleness in real-world applications

The challenge of test brittleness is significantly amplified in modern, dynamic web applications that extensively use JavaScript [33], asynchronous operations [34], and complex UI frameworks [35], and are deployed on complex infrastructures [36], [37]. Many existing studies on test brittleness utilize specific open-source projects that are relatively static and are deployed in controlled, local environments. Such setups often fail to capture the full spectrum of issues that arise in real-world production applications.

Consider a concrete example of a subtle brittleness issue that can occur on a real-world travel booking website when selecting ages for multiple children from dynamically generated dropdown menus, as depicted in Figure 1. A problem frequently arises in this interaction flow because web elements within dynamic components like this dropdown are often transient. They are actively added to, removed from, or updated within the Document Object Model (DOM) as the user interacts with the page. For instance, after selecting an age for *Child 1* and then activating the dropdown for *Child 2*, the options for *Child 1* might not instantaneously disappear from the DOM. In this brief interval, code attempting to select an age for *Child 2* (i.e, "7 years old") might inadvertently find and establish a reference to the "7 years old" element from the *Child 1* dropdown. However, by the time the script attempts to perform a click action on this referenced element, the *Child 1* dropdown and its options may have fully vanished from the DOM. This can lead to the stale element reference exception [38], as the referenced DOM element becomes detached due to these rapid UI updates.

This problem can persist, even when employing established practices such as explicit waits, which our generated test code incorporates. This illustrates a form of brittleness that is difficult to identify, reproduce, and diagnose outside the context of testing on complex, real-world web applications. This suggests that the underlying root causes are likely multi-faceted, potentially stemming from how the web application's framework handles UI updates, the inherent testability of the application's design, or subtle aspects of test script logic interacting with a rapidly changing DOM. Uncovering these deeper, often elusive, root causes to understand exactly "who's to blame" when tests fail in these complex scenarios requires a pragmatic approach that studies brittleness directly on diverse, evolving applications. Such an investigation, as pursued in this paper, is important for developing more robust testing and repair solutions. Our work, therefore, aims to identify these multifaceted causes of brittleness and also explore how modern techniques, like Large Language Models, might assist in repairing tests by reasoning about these complex failures that depend heavily on context.

## III. Case Study Setup

Figure 2 gives an overview of our study design, from the setup of our study to the answer of our research questions. This section describes the setup of our study.

### A. Subject data

We build our test scenarios on the Mind2Web dataset [30], which provides 2,350 multi-step tasks collected from 137 widely used real-world websites that are selected using the SimilarWeb rankings [39], which is a very commonly used top websites ranking list. The websites in the dataset include *AirbnbAmazon*, and *AccuWeather*, etc., spanning 31 domains (e.g., shopping, travel, finance, public services, media). Each task includes a natural-language description and a verified action sequence of pairs (*operation*, *target element*) over the original DOM snapshots. The action sequences were recorded from human annotators completing tasks naturally within a web browser, which ensures the recorded workflows mirror how a person would actually navigate the website, making them representative of test cases that are written manually. The operations include *Click*, *Type*, and *Select*, which are the most common operations a user would perform within a web application [17]. On average, a task in the Mind2Web dataset [30] contains 7.3 actions, and the pages include about 1,135 DOM elements per view, reflecting the realistic complexity of the pages. The dataset's extensive diversity provides a strong basis for generalizable findings. At the same time, its focus on realism, using complex live websites instead of simplified environments, creates the exact conditions where real-world test case brittleness appears. By studying these human-curated workflows, our analysis can directly reflect the practical challenges of maintaining automated web GUI tests during actual software development. Mind2Web is widely used in many prior studies [40]–[42], further validating its relevance and suitability as a benchmark for web automation research.

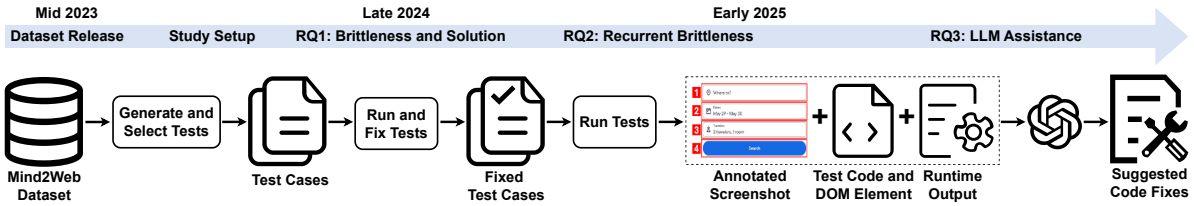



Fig. 2: An overview of our study approach

To avoid triggering the potential anti-bot services of these websites, we use the undetected-chromedriver [43] as the web driver to interact with the websites. However, after some exploration, we still have to exclude 17 websites that presented significant access barriers to automated testing. Some websites have shut down (e.g., www.redbox.com), some websites still have their anti-bot services triggered even with undetected-chromedriver, prompting the test scripts to do the CAPTCHA [44] verification or two-step verification (e.g., www.zocdoc.com). In total, our study includes 120 websites from the original Mind2Web dataset. The remaining websites still cover a wide range of functions, from e-commerce and travel booking to information retrieval and service scheduling, ensuring our analysis is grounded in a rich collection of real-world web applications.

### B. Test case generation

To transform the steps to complete the tasks in the Mind2Web dataset into executable web GUI test cases, we propose a pipeline with two major steps: data extraction with locator generation, and executable test cases generation.

TABLE I: Locators used in the test generation process.

| Locator name | Description |
|---|---|
| id | By the element's unique *id* attribute |
| link text | By a link element's visible text |
| name | By the element's *name* attribute |
| xpath link | XPath search for a link element |
| xpath attribute | XPath search using a specific attribute |
| xpath relative id | XPath search relative to a parent or sibling's *id* |
| xpath image | XPath search for an image by its attributes |
| xpath href | XPath search for an anchor tag by its *href* attribute |
| xpath inner text | XPath search using specific inner text content |
| xpath position | XPath search by the element's absolute DOM position |

*Data extraction and locator generation*: This initial stage processes the raw task information from Mind2Web, which includes MHTML snapshots of web pages corresponding to each action step. We first parse these snapshots and decode the encoded content, including the content within iframes, to HTML files. As the target element of each step is flagged with a unique identifier in the dataset, we then use these identifiers to extract the target elements from the HTML files. An important component of a robust test case is a reliable locator, the mechanism used to find a specific element on a web page. To ensure the generated tests are as robust as possible from the outset, we generate a comprehensive and prioritized suite of locator candidates for each target element. The types of generated locators and the prioritization strategy are modeled after the best practices employed by modern automation tools like Selenium IDE [45], which prioritize locators that are less likely to change during website evolution. Our generation process creates a list of candidate locators ordered by a predefined priority, from most to least stable. The specific locators and their descriptions are detailed in Table I. For example, given an HTML element like `<button id="login-btn" name="name-btn" type="button">Login</button>`, it would first generate a list of potential locators in a specific order of preference: an ID locator (`id=login-btn`), which is the top priority due to its intended uniqueness; A name locator (`name=name-btn`), used if a unique ID is unavailable; An XPath attribute locator, such as one constructed from the type attribute (e.g., `//button[type='button']`), which is selected when neither an ID nor a name provides a stable reference. Other candidates, such as those based on the inner text or the element's absolute position in the DOM, would follow with lower priority. Our tool replicates this by generating a prioritized suite of locator candidates for each target element.

*Executable test cases generation*: As depicted in Listing 1, each web GUI test is composed of a sequence of actions. The code for each individual test action is constructed using standard automation testing framework APIs (e.g., Selenium), incorporating both the element locator and the specified action type. For each action, we first create the code to find the target element. This involves selecting the first available locator from the previously generated prioritized suite of candidates for that element. This chosen locator is then used to populate the element finding API call of the framework. Subsequently, the corresponding action type (e.g., click, type text) recorded in the dataset for that step is used to complete the construction of the executable code for that particular action. To systematically assemble these individual actions into complete test suites, we employ a template engine, Jinja [46], which organizes the action code sequences into a set of executable test cases compatible with the pytest framework, automatically including necessary import statements and test structures. Following this automated process, we generated one distinct test case for each task in the Mind2Web dataset, resulting in a total of 2,052 executable test cases. Note that these test cases are not fully automatically program-generated; rather, they are guided by human-annotated action sequences from the Mind2Web dataset to ensure the workflows reflect realistic user behavior. This approach closely mirrors how testers use record-and-playback tools like Selenium IDE in practice. More details can be found in our replication package[1].

[1]https://figshare.com/s/fa27cafd8f27ae8a6389

### C. Test selection

To investigate the root causes of test brittleness, it requires iteratively executing test cases designed for those older website versions against their modern counterparts. Manually executing and repairing all 2,052 generated test cases would be prohibitively labor-intensive and time-consuming. Therefore, we need to select a representative subset of tests that is both manageable and representative. To achieve this, for each unique website in our dataset, we selected the longest and most complex test case, using the number of action steps as a direct proxy for length and complexity. This resulted in a focused cohort of 120 test cases. Our rationale is that longer, more complex test cases are more likely to cover a wider range of functionalities, traverse more diverse UI elements, and involve more intricate user interaction flows. Consequently, these test cases are inherently more susceptible to breaking when a website evolves. By focusing on these complex cases, we maximize the probability of observing a broader and more diverse spectrum of test failures within a smaller sample size. Figure 3 illustrates the distribution of lines of code for these selected test methods, excluding import statements. In total, these test methods comprise 2,346 lines of code. The number of lines of code per test method ranges from 6 to 59, with an average of 19.55 lines.

To further validate the representativeness of the test cases we selected, we manually inspected the 120 selected test scenarios. This analysis confirmed that our sample is not merely long but also functionally diverse and representative of complex, real-world web app usage. The scenarios encompass a wide range of user journeys, including end-to-end transactions involving multi-step booking processes and purchasing flows, complex searches with filtering that require users to apply multiple specific filters to find jobs, real estate, or products, and detailed form submissions through extensive forms for service quotes, applications, or scheduling. Furthermore, even within the same domain of the Mind2Web dataset, the selected scenarios test distinct user workflows to avoid redundancy. For instance, the scenarios in the automotive domain are not limited to a single action type like purchasing. They span a variety of distinct user goals such as valuing a current vehicle (https://www.kbb.com/), selling a car through a detailed form (https://www.cargurus.com/), searching for a new car with specific features (https://www.carmax.com/), and scheduling a demo drive (https://www.tesla.com/). This variety in tasks ensures our findings are not confined to a narrow subset of web applications but are reflective of the broader challenges in web GUI testing.

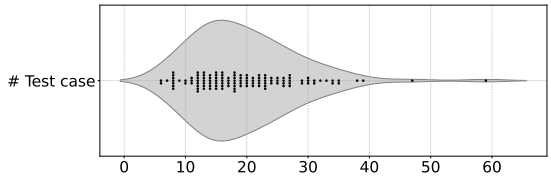


Fig. 3: Code lines distribution of the selected tests

### D. Environment setup

Our experiments are conducted on a macOS Ventura 13.1 operating system, equipped with an Apple M1 Pro CPU and 32GB of RAM. We select Google Chrome as the testing browser with a 1920×1080 resolution during test execution. To set our focus on less commonly explored aspects of brittleness and to minimize unrelated test failures, we enhance standard element interaction methods. In light of prior work [28], [29], functions such as element finding and clicking are augmented to incorporate Selenium's explicit wait mechanisms [47]. This ensures that the framework dynamically waits for target elements to be present, visible, and interactable before any interaction attempts. Furthermore, a pre-test step is implemented to automatically dismiss ubiquitous cookie consent windows [48] using pre-identified locators, ensuring a consistent starting state for each test.

### E. Test execution timeframe

The Mind2Web tasks were documented in mid-2023. By generating tests from this dataset and executing and repairing these tests on live websites in late 2024 and early 2025, our study introduces a temporal gap of 1.5 years and 0.5 years, respectively. This allows us to investigate the root causes of brittleness that accumulate over longer evolution cycles, providing findings that are representative of the substantial maintenance challenges developers and testers face in real-world development workflows, which are supported by some prior studies. Feldt et al. [49] found that test cases are long-term assets and have a long lifespan, with a typical half-life of 5 to 12 months, indicating that test cases actively evolve and fail over timeframes highly comparable to the one in our study. Also, in practice, test maintenance does not always happen continuously [50]. Alégroth et al. [51] revealed that in the industrial environment, test cases are often maintained infrequently and degrade to a point where a significant "big-bang" maintenance effort is required. This reflects a realistic scenario where developers must repair a large batch of old, broken tests after a period of deferred maintenance. Our study directly simulates this important and costly process by repairing a suite of 1.5-year-old and 0.5-year-old tests. Furthermore, our practice mirrors how automated repair tools are practically applied. State-of-the-art automated web GUI test repair tools are often applied to major releases where numerous application changes are accumulated, instead of on a per-release and per-commit basis [18], [52], [53]. For example, Lin et al. [53] applied automation tools to fix web GUI test cases for web applications released 1 to 30 versions ago. Similarly, Shao et al. [52] applied automation tools to fix web GUI test cases that are at least 2 years old. Therefore, evaluating tests that are 1.5 years and 0.5 years old is representative of the brittle and out-of-sync test cases that developers frequently encounter.

## IV. Case Study Results

In this section, we present the results of our case study. Overall, it comprises three main parts: (1) an initial analysis

of test case failures when legacy tests encounter current web applications to identify root causes and repair strategies; (2) an examination of recurring brittleness patterns to understand long-term test case maintenance challenges; and (3) an investigation into the use of LLM for assisting in test repair by emulating human-like approaches. We detail the motivation, approach, and results for each our research question.

*RQ1: What causes legacy web GUI tests to break on real-world applications?*

**Motivation** Legacy web GUI test cases are often tightly coupled with the structure and behavior of the web applications they were originally designed to test. However, as web applications evolve over time, for example, through updates to their layout, DOM structure, or functionality, these legacy test cases often fail due to their inability to adapt to such changes. For example, a test case that interacts with a specific button may break if the button's identifier changes or if the button is relocated on the page. These failures occur even when the core functionality of the application remains intact, leading to significant maintenance challenges. Understanding the root causes of these failures when legacy test cases are executed on updated web applications is essential for developing effective repair strategies and reducing the maintenance burden associated with outdated test suites.

**Approach** To investigate the root causes of failures in legacy test cases, we conduct a systematic analysis through an iterative process. As depicted in Figure 2, this process begins by executing legacy test cases, originally generated from the Mind2Web dataset (mid-2023 release), against the current versions of the respective web applications (as of late 2024). When an unexpected failure occurs during execution, we collect the runtime output and a screenshot of the web page at the point of failure. Based on this information, we then identify the root cause and implement a corresponding fix to let the test continue. This repair cycle is repeated until the test case executes successfully without errors. To mitigate potential bias from flaky tests, a test case is considered fully repaired only after it passes consistently in at least two separate executions. Throughout the repair process, detailed comments are added to the codebase to document each root cause and the rationale behind its fix. The entire codebase is maintained in a software version control system to facilitate tracking of changes and subsequent labeling of the root causes.

To classify and label the root causes of brittleness identified during the iterative repair process, we employ an open coding pair review [54], [55]. Initially, two authors independently review the comments and code change history for a selected set of 20 test cases. Following this individual review, they discuss their findings to reach an agreement on an initial codebook. Using this established codebook and shared understanding, the same two authors then independently label all identified brittleness instances across the entire dataset. To assess the inter-rater reliability of this initial independent labeling, we calculate Krippendorff's alpha coefficient [56]. This statistical measure is chosen because it is suitable for situations where multiple categorical labels can be assigned to a single instance, as a test case may exhibit brittleness due to several concurrent reasons. We then quantify and interpret the level of agreement achieved [57]. Any remaining discrepancies in labeling are subsequently reviewed and resolved through discussion between the two authors until a consensus is reached. This rigorous process ensures that the identified root causes of test failures are both accurately and reliably categorized, providing a solid foundation for our analysis.

**Results** In the initial execution phase, all 120 legacy test cases failed, underscoring the significant brittleness arising from discrepancies between these older tests and the current web applications. To move beyond temporary fixes and comprehensively address this brittleness, we classify the identified root causes based on the primary software component involved in the automated testing process. These components are: the automation framework, the web application under test, the test code itself, and external factors. This classification aims to highlight which component requires improvement to achieve robust automation. Our rationale is similar to production software debugging, where a failure's root cause can stem from the production code, its dependent libraries and frameworks, or the deployment environment, instead of always from the production code.

Table II presents our classification, which groups the sources of brittleness into four primary categories: automation framework implementation, test script design, web application complexity, and external factors. For each primary category, the table further details specific sub-categories, accompanied by a description of the issue and a concrete example to illustrate how such failures manifest in practice. The inter-rater reliability for our labeling, measured using Krippendorff's alpha, is as follows: for automation framework implementation, $\alpha = 0.75$ (moderate agreement); for test script design, $\alpha = 0.81$ (strong agreement); for web application complexity, $\alpha = 0.88$ (strong agreement); and for external factors, $\alpha = 1.00$ (perfect agreement). The results suggest that our categorization of failure causes is both reliable and consistent.

*Automation framework implementation:* Our findings indicate that a substantial number of test case failures (66 out of 120 instances) are not solely due to the test code itself but can also be attributed to limitations or unsupportive behaviors within the automation framework's implementation. While prior work has acknowledged framework wait behavior as a root cause of brittle tests [13], our study additionally identifies and explores brittleness linked to the framework's element locating features and page scrolling capabilities, aspects less emphasized in previous research.

Regarding the **framework wait behavior**, we observe that failures often occur because frameworks may primarily use an element's display and enabled status as indicators of interactability. However, in dynamic real-world web applications, situations where a target element is obstructed by other elements are common. This can lead to actions being intercepted or failing, even when standard framework waiting mechanisms are employed. To truly address such brittleness,

TABLE II: Rationales of the brittleness of automated web GUI testing

| Aspects involved | Category | Description | Freq | Ref./New |
|---|---|---|---|---|
| **Automation framework implementation** (66/120) | Framework element location feature | Limitations in the framework's element location capabilities prevent the identification of certain element types, resulting in location failures. | 29/66 | New |
| | Framework wait behavior | Inadequate waiting mechanisms that often identify the element as interactable even when it is not, leading to the failure of the action. | 26/66 | [13] |
| | Framework scrolling behavior | The framework's scrolling functionality may fail to bring target elements completely into the viewport, causing subsequent interaction attempts to fail. | 17/66 | New |
| **Test script design** (111/120) | Element locator design | Selection of ambiguous or overly specific (fragile) locators, especially when more robust alternatives exist, leads to inconsistent element identification. | 99/111 | [6], [53] |
| | Test data specificity | Test input is overly sensitive to temporal or environmental conditions can render test cases invalid as these external factors change over time. | 42/111 | [58] |
| | Alignment with application logic | Test scripts no longer accurately reflect the current application workflow, such as when steps are added, removed, or modified. | 30/111 | [6], [11] |
| | Interaction readiness | Scripts failing to await stabilization of dependent UI elements before target interaction can cause inconsistent execution. | 24/111 | [5], [13] |
| | Cross-window/tab interaction | Test scripts may improperly handle interactions spanning multiple browser windows or tabs, such as by failing to wait for new tabs to fully load. | 14/111 | New |
| **Web application complexity** (79/120) | Incidental GUI update | Seemingly minor alterations to the UI, often side effects of unrelated development work rather than new features, can unexpectedly invalidate tests. | 69/79 | New |
| | Deep shadow DOM structure | Elements encapsulated within multiple layers of Shadow DOM, making interaction logic complex and prone to breakage. | 7/79 | New |
| | DOM element redundancy | Presence of duplicate elements in the DOM, often for responsive design or dynamic content, can lead to ambiguous locator matches and test failures. | 6/79 | New |
| | Complex iframe structure | The use of multiple or nested iframes within the application complicates element interaction and increases test script brittleness | 2/79 | New |
| **External factors** (10/120) | Application-native dynamic pop-ups | Pop-ups generated by the application's own business logic can appear unexpectedly, disrupting test flow. | 8/10 | New |
| | Third-party content | Dynamically injected third-party content can alter page layout or obscure elements, thereby interfering with test execution. | 2/10 | New |

The 'Freq' column indicates the number of test cases within the respective 'Aspects involved' group where this category was identified as a root cause; a single test case can be assigned multiple category labels. The 'Ref./New' column lists literature where related concepts are discussed. 'New' indicates a new category that has never been observed or discussed in prior work.

rather than testers resorting to temporary workarounds like fixed thread sleeps, automation frameworks themselves may need to incorporate more comprehensive interactability checks, ensuring an element can be properly engaged with before an action is attempted.

Brittleness related to the **framework element locating feature** manifests in several ways. For example, we found instances where the Selenium framework struggled to reliably locate `<SVG>` elements using XPath. When a locator matches multiple elements, the framework's default behavior of selecting the first element in the DOM order can lead to targeting a non-visible element if others matching the locator precede it. This problem is often compounded by DOM element redundancy (a label from the web application complexity aspect), forcing the use of more specific and potentially fragile locators, which in turn can contribute to brittleness categorized under element locator design in the test script.

As for brittleness concerning the **framework's scrolling behavior**, we encountered illustrative examples during our iterative testing. For instance, when testing *akc.org*, a test script intended to click the "Companion Events" element failed because the element was outside the current viewport. While many automation frameworks, including Selenium, are designed to automatically scroll elements into view before interaction, this did not consistently occur. Consequently, the click action could not be performed, causing the test case to fail. To proceed with the test, a manual scroll command had to be injected as a temporary workaround, highlighting a deficiency in the framework's automated scroll-to-view capability.

*Test Script Design:* A significant majority of failures, 111 out of 120 instances, were attributed to aspects of Test script design. This underscores the critical role that the construction and maintenance of test scripts themselves play in overall test suite robustness.

The most dominant sub-category here was **element locator design**. Failures in this category arise when test scripts employ ambiguous, overly specific, or fragile locators, especially when more stable alternatives could have been selected. Such locator choices lead to inconsistent element identification as the application UI evolves, a well-documented challenge.

**Test data specificity** also contributed to brittleness. This occurs when test inputs are too closely tied to specific temporal or environmental conditions (e.g., hardcoded dates, user-specific data). When these conditions change, the test data becomes invalid, leading to predictable failures. Lack of Alignment with application logic was another key factor. As applications are updated, test scripts that are not correspondingly modified to reflect changes in workflows, such as new or removed steps, become obsolete and inevitably fail.

Issues with **interaction readiness** were also observed. These failures happen when scripts attempt to interact with a target element before all dependent UI components or

cascading controls have fully stabilized, leading to inconsistent behavior even if the target element itself appears interactable.

Finally, **cross-window/tab interaction** presented challenges. Test scripts sometimes failed to properly manage context when application functionalities spanned multiple browser windows or tabs, for instance, by not adequately waiting for a new tab to load or by failing to switch focus correctly. This category, to our knowledge, has not been extensively detailed as a specific root cause in prior work.

*Web Application Complexity:* The inherent Web application complexity was identified as the root cause in 79 instances. This highlights how the design and structure of the application itself can introduce significant challenges for test automation.

**Incidental GUI update** was the most frequent sub-category here. These failures occur due to seemingly minor alterations in the UI or underlying DOM, which are often side effects of unrelated development work rather than new features, yet they unexpectedly invalidate existing test cases.

**Deep shadow DOM structure** also contributed to brittleness. When target elements are encapsulated within multiple, nested layers of Shadow DOM, standard element location strategies often fail, making interaction logic complex and tests prone to breakage.

**DOM element redundancy** was another factor. The presence of duplicate or near-duplicate elements in the DOM, frequently introduced for responsive design or dynamic content rendering, can lead to ambiguous locator matches, causing tests to fail or interact with incorrect elements.

Similarly, a **complex iframe structure**, involving multiple or nested iframes, complicates element interaction significantly. Test scripts must explicitly handle context switching for each iframe, and complex structures increase the likelihood of errors and test script brittleness.

*External Factors:* While encountered less frequently in our study, external factors accounted for 10 failure instances. These factors highlight discrepancies that can arise between the application state during testing and the version or conditions experienced by end-users, often due to elements not displayed in the controlled testing environment.

Within this aspect, **application-native dynamic pop-ups** were the predominant contributor. These are overlay windows or modal dialogs originating from the application's own business logic or internal rules. Such pop-ups can disrupt the intended test flow if they appear at unexpected times from the test script's perspective or if they are inherently difficult for automation to handle.

Additionally, **third-party content** led to test failures. Content such as advertisements or analytics scripts, dynamically injected into the page from external sources, can alter the page layout, obscure target elements, or introduce unforeseen behaviors, thereby interfering with test execution.

**RQ1 Findings:** Legacy test brittleness on evolved web applications can arise from multiple factors, including test script design, web application complexity, and framework limitations, rather than a single root cause.

*RQ2: How does web GUI test brittleness persist as applications continuously evolve?*

**Motivation** After the initial repair and execution, we have constructed and applied some temporary solutions. Beyond the initial failures of legacy test cases, web applications continue to evolve, introducing new challenges for maintaining automated test suites over time. Even after fixing brittle test cases, the same or similar issues may recur as applications undergo further updates, raising questions about the long-term effectiveness of these fixes. This ongoing maintenance effort can become increasingly costly and labor-intensive, especially in dynamic, real-world environments where web applications are subject to continuous changes in their structure, behavior, and external dependencies. Investigating how brittleness persists and whether the applied fixes remain effective over time is critical to identifying generalizable strategies for sustaining the resilience of automated test cases in the face of continuous application evolution.

**Approach** To investigate how test brittleness recurs during the web application evolution process, we conduct a longitudinal study as illustrated in Figure 2. Approximately six months after the initial repair and execution phase (described under RQ1), we re-execute the previously repaired test cases. These tests are run against the then-current versions of the respective web applications (as of early 2025). Similar to our initial investigation, when a test case failed, we collected runtime output and a screenshot at the point of failure. We then systematically analyze this information to determine the root causes of these new or recurring failures and compare these findings against the root causes identified during the initial repair phase. This comparison allows us to understand patterns of persistent brittleness.

**Results** Following the re-execution of the 120 previously repaired test cases, approximately six months later (early 2025), we analyzed their outcomes to understand how brittleness persists in continuously evolving web applications. Figure 4 presents the distribution of these outcomes.

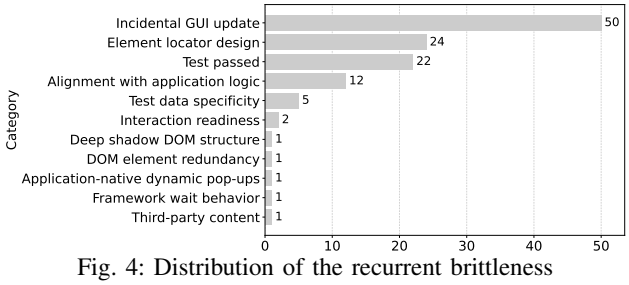


Fig. 4: Distribution of the recurrent brittleness

A straightforward observation is that a large majority of the test cases failed again. Specifically, only **22 out of the 120 (18.3%)** test cases passed without any further modification.

This indicates that the initial fixes, while effective at the time, often did not provide long-term resilience against subsequent application changes. The remaining **98 test cases (81.7%)** failed, and they all fall in the categories from RQ1.

Among the recurring failures, issues related to Incidental GUI update were the most prevalent, accounting for 50 instances. This strongly suggests that continuous, seemingly minor changes to the UI and DOM structure of web applications are a primary driver of ongoing test maintenance. The second most common cause of recurrent failure was Element locator design, with 24 instances. This implies that even if locators were fixed once, further application evolution rendered a new set of them ineffective. Other notable categories of recurring brittleness included Alignment with application logic (12 instances), indicating that changes in application workflow continued to invalidate tests, and Test data specificity (5 instances), where previously valid test data likely became outdated or inappropriate for the evolved application state. Smaller numbers of failures were attributed to Interaction readiness (2 instances), and single instances each for Deep shadow DOM structure, DOM element redundancy, Application-native dynamic pop-ups, Framework wait behavior, and Third-party content.

These results highlight several key aspects of persistent brittleness. First, the high re-failure rate underscores the dynamic nature of the studied web applications and the continuous challenge of test maintenance. Second, the prominence of Incidental GUI Update and Element Locator Design as recurring causes points to the ongoing instability of UI elements and their locators as a central problem.

Notably, "Incidental GUI updates" often involved modifications typically considered to affect reliable locators, such as altering or removing element IDs, or changing data-testid attributes. These changes sometimes occurred in ways that inadvertently degraded, rather than improved, testability, challenging the assumption that such attributes are always stable. This suggests that strategies relying solely on reactively fixing locators after each failure may prove unsustainable. Instead, more adaptive locator strategies or approaches less sensitive to minor UI modifications might be necessary.

These results also imply that the "blame" for brittleness does not solely rest with test script design. The nature of these UI changes points to the web application under test itself as a significant contributing factor to test instability. Furthermore, the recurrence of issues like Alignment with Application Logic and Test Data Specificity indicates that comprehensive test maintenance must consider not just locators but also the test logic and data to keep pace with evolving application functionalities and states. Achieving robust and sustainable automated testing, therefore, likely requires a collaborative effort, involving not only testers in designing resilient scripts but also developers in building more testable applications and considering the impact of UI changes on existing tests.

> **RQ2: Findings:** Test brittleness is highly persistent in evolving web applications, with most repaired tests re-failing within months.

*RQ3: Can LLMs assist in repairing brittle web GUI tests by emulating human approaches?*

**Motivation** The significant re-failure rate of previously repaired test cases, as highlighted by our RQ2 findings (where 81.7% failed again), underscores the persistent and costly nature of web GUI test maintenance due to continuous application evolution. This reality reveals the limitations of purely manual, reactive fixes and motivates exploring more efficient, sustainable strategies. While LLMs offer a promising possibility for reducing this maintenance burden, existing research [59], [60] into LLM-assisted web test repair has often been limited. These studies are typically conducted in controlled environments rather than on complex, evolving real-world systems like those in our study, and they frequently focus on specific failure types, such as only locator breakages, rather than the multifaceted root causes we observed. Thus, there is a need to investigate how LLMs perform when tasked with repairing tests that repeatedly fail in dynamic contexts, particularly when using the rich, contextual information similar to what human engineers rely upon.

**Approach** To answer RQ3, we evaluate the capability of an LLM, specifically, GPT-o3, to repair the 98 test cases that failed during our RQ2 re-execution phase. As depicted in Figure 2, for each of these re-broken test cases, we provide the LLM with information that mirrors the information a human engineer would typically use for manual debugging: (1) the failed test method's source code (incorporating its prior RQ1 fix), (2) runtime outputs with errors and stack traces, (3) the specific failing code line, (4) an annotated screenshot of the webpage at failure (using numeric and box marks to leverage LLM visual grounding capabilities [61], [62]), and (5) a textual list of these annotated elements. Our method uses only the current failure context, unlike traditional automated repair techniques that often rely on older application versions for comparative analysis [18]. The LLM is prompted to generate up to five distinct, actionable code fixes, and a repair is considered successful if any single candidate fix enables the test case to pass the previously failed step on the current application version.

**Results** Our evaluation of GPT-o3 on the 98 re-broken test cases revealed its considerable potential for assisting in ongoing test maintenance. Provided with inputs mirroring human debugging practices, the LLM successfully proposed a correct fix for 81 (82.7%) of these challenging failures. Its performance was particularly strong for the most prevalent recurring issues: it resolved 46 out of 50 (92.0%) failures due to Incidental GUI update and 19 out of 24 (79.2%) related to Element locator design. The LLM also effectively addressed most failures concerning Alignment with application logic (10 out of 12 fixed), and all those due to Test data specificity (5 out of 5 fixed). This high success rate in these categories suggests significant utility for automating frequent repair tasks.

However, the LLM did not provide successful repairs for any instances of failures attributed to DOM element redundancy, Interaction readiness, Deep shadow DOM structure, Application-native dynamic pop-ups, Framework wait behavior, or Third-party content. These categories often represent more intricate failure scenarios, potentially requiring a deeper understanding of complex state transitions, nuanced timing beyond standard waits, or specific architectural workarounds. The LLM's inability to resolve these types of issues, despite receiving rich contextual information, logically indicates that current models may still lack the profound situational awareness or extensive experiential knowledge that human engineers apply to such problems. Therefore, while LLMs demonstrate a strong capability to serve as powerful assistants by addressing a substantial volume of brittleness issues through emulated human-like pragmatic approaches, human expertise and deeper system knowledge remain indispensable for diagnosing and resolving the elusive or architecturally complex test failures.

**RQ3 Findings:** Given human-like diagnostic information, LLMs can effectively repair a large majority of recurring web test failures, human expertise remains vital for more complex issues, though.

## V. Lessons Learned

In this section, we discuss the lessons that we learned.

**L1: Brittleness is a multifaceted problem with shared responsibility.** Our RQ1 findings indicate that web GUI automation testing brittleness rarely stems from a single isolated cause. While test script design is a major factor, web application complexity and testability, and limitations within the automation framework implementation also contribute significantly to the brittleness. This distributed nature of brittleness highlights the need for cooperative efforts involving developers, testers, and the automation framework maintainers to enhance the robustness of the overall process rather than blaming a single part for the brittleness.

**Actionable guidance.** Developers and testers should establish cross-functional review processes where UI changes are evaluated for test impact before deployment. Automation framework developers should design frameworks that inherently reduce the potential for brittleness by prioritizing features that handle the real-world complexities of dynamic content and element obstruction and testing the framework on various real-world websites. Researchers can create a brittleness analytics platform to form a feedback loop among developers, testers, and framework maintainers.

**L2: Web GUI testing brittleness is prevalent in real-life systems and a recurring issue.** The complete failure of all legacy tests in RQ1, and 81.7% of the test cases that were repaired in RQ1 failed again within about six months, highlighting that web GUI test maintenance is a continuous and substantial challenge. The consistent recurrence of problems like incidental GUI updates and element locator design issues indicates that purely reactive fixes are often insufficient, and proactive and adaptive strategies are needed.

**Actionable guidance.** The persistence of brittleness indicates that reactive, one-off fixes are technically and financially unsustainable. Testers should proactively classify brittleness patterns and work with developers and framework maintainers to address recurring issues at the source. They should allocate a dedicated maintenance budget (e.g., 15-20% of testing effort) specifically for ongoing maintenance rather than treating repairs as unexpected overhead. Researchers can explore automatic solutions to classify failures using a taxonomy like ours and pinpoint the related modules in test cases, web applications, and automation frameworks that need to be fixed.

**L3: Web applications lacking stable identifiers pose challenges to their suitability for automation testing.** The prevalence of element locator design brittleness sometimes originates from web applications or their underlying frameworks generating dynamic element identifiers while omitting other stable, test-friendly attributes like `data-testid`. This absence of reliable hooks forces the test script designers to use inherently fragile locators (e.g., absolute XPath), significantly increasing the brittleness and maintenance effort, thereby posing challenges to the suitability of such web applications for effective, long-term automation testing.

**Actionable guidance.** Web app developers should adopt a mandatory coding standard that requires stable, test-specific attributes like `data-testid` for all interactive elements. Testers can perform a formal testability audit before committing to large-scale automation to evaluate whether a web application is suitable for automation testing. Researchers can focus on creating static analysis tools and IDE plugins that enforce testability standards in real-time.

**L4: Current automation frameworks can not fully address real-world dynamic GUI complexities.** While our study already incorporates robust strategies like explicit waits [47] from the automation framework, there are still 66 instances where the limitations of the automation framework contributed to brittleness. The fact that framework wait behavior issues, though fewer in number, persisted into RQ2 brittleness even after initial fixes suggests that the default mechanisms provided by popular frameworks like Selenium may not be inherently robust enough to handle all complexities of modern dynamic GUIs without custom enhancements or workarounds by testers. This indicates an opportunity for framework evolution to better support testing in highly dynamic environments, reducing the burden on individual test script writers to implement complex custom synchronization.

**Actionable guidance.** Testers and developers can develop an organization-specific wrapper library on top of the base automation framework to handle complex or domain-specific situations. Automation framework providers can use our findings to guide improvements, such as implementing state-based waiting and more resilient locator strategies, to better support testing in dynamic and evolving web UIs. Researchers can create a benchmark for evaluating the capability of current automation frameworks on testing real-world web applications with various dynamic GUI complexities.

**L5: Developers may change the web application without considering the need for automation testing.** The prevalence of failures due to Incidental GUI updates indicates that web application modifications are frequently made without adequate consideration for their impact on automated testing. Our findings show that even seemingly minor UI or DOM alterations, which do not necessarily introduce new features and sometimes affect attributes generally considered stable, can readily break existing tests. This highlights how development practices that do not prioritize or integrate testability considerations during application evolution contribute significantly to test brittleness and the subsequent maintenance effort.

**Actionable guidance.** Developers should integrate test impact assessment into code reviews to flag UI changes that may affect existing tests. Researchers should propose automated solutions to predict when UI changes will break existing tests.

## VI. Threats to validity

**Internal validity.** We use *Selenium*, a widely adopted automated testing framework, to conduct our experiments and analyze test brittleness. However, the specific failure rationales we identify, especially those related to framework behavior, may not generalize to other automation tools. Additionally, the process of manually repairing and labeling test failures involves human judgment, which may introduce some subjectivity despite our efforts to ensure consistency through independent review and consensus. Our experiments are conducted only in Google Chrome at a fixed 1920×1080 resolution, which limits environmental diversity. Brittleness may manifest differently across browsers, devices, resolutions, or network conditions, and future work should expand testing environments to improve generalizability.

**External validity.** The results of our study are based on the Mind2Web dataset. Although this dataset covers a diverse set of web applications, it may not capture every possible scenario or type of web system found in practice. As a result, there may be brittleness patterns or failure causes present in other domains that are not reflected in our findings. Additionally, for each application, we selected only one test case (the longest and most complex) to evaluate brittleness. While this approach helps ensure that we capture a wide range of potential issues, it may not fully represent the brittleness experienced across all test cases within an application. However, our selection strategy is designed to maximize coverage of diverse functionalities and UI interactions within a manageable scope, as these complex cases are most likely to expose a wide spectrum of brittleness issues. We believe that our methodology provides meaningful insights into the general brittleness of web UI tests.

**Construct validity.** Our evaluation of LLM-based repair is based on the availability of specific information (e.g., error logs, annotated screenshots) that can be leveraged to identify the root cause of the failure, aiming to emulate the information needed in the human manual repair process. However, the type of information and how much of it is available when a test fails can be different from one case to another. Because of this, how well the LLM performed in our study, using the particular information we provided, might not be the same for all repair tasks in practice. This limitation may impact the generalizability of our findings regarding LLM-based repair effectiveness. Furthermore, the non-deterministic nature of LLMs may limit the generalizability and stability of our findings. Future work should include multiple LLMs and repeated trials to assess the consistency and reliability of automated repairing across different models and scenarios.

## VII. Related Work

**Empirical studies of web GUI testing.** Numerous studies have investigated the challenges associated with automated web GUI testing. Romano et al. [11] find that a large share of flaky tests are related to mismatches between page-rendering time and hard-coded waits. While Leotta et al. [7] show that Record-and-Replay (R&R) tools reduce the upfront cost of creating test cases, yet incur higher long-term maintenance costs than programmable test suites; they also report flakiness and fragility as key obstacles with GUI tests [5]. Marshall et al. [13] report that Selenium configuration choices, particularly those related to waiting strategies, directly impact flakiness. Finally, both Hammoudi et al. [6] and Nass et al. [4] identify inadequate waiting as a primary root cause of R&R failures. Unlike these studies that often focus on specific aspects like wait times or are performed in controlled environments, we systematically analyze brittleness from multiple practical perspectives to provide a comprehensive understanding of root causes as they manifest in real-world environments.

**Automated test repair and maintenance.** There are also many studies about fixing and maintaining the web GUI test cases. Current web GUI test repair work spans behavioral difference detection [18], incremental repair of web GUI test cases [19], vision-based fixes [24], and model-based approaches for R&R test suites [63], [64]. Techniques involved in automated web GUI test repair include tree matching [65], iterative matching [53], semantic analysis of executions [25], and, most recently, LLM-driven repair [59]. Different from these studies, we focus on revisiting the challenges of achieving robust automated web GUI testing from a pragmatic perspective. Our findings may inspire researchers to focus not only on improving the test cases but also on other modules (e.g., automation framework) involved in automation testing.

## VIII. Conclusion

This paper presented a pragmatic analysis of web GUI test brittleness. Our study confirms that failures in evolving applications originate not just from test script design, but also significantly from web application complexity and automation framework limitations, making "blame" a distributed issue. The high recurrence rate of failures (81.7%) highlights brittleness as a persistent challenge, not easily addressed by one-off fixes. While LLMs show promise in resolving common issues (82.7% fix rate), human expertise remains essential for complex cases. Future research should focus on real-world failure scenarios and explore strategies for improving test resilience, framework robustness, and AI-assisted maintenance.

## REFERENCES


[1] L. Christophe, R. Stevens, C. De Roover, and W. De Meuter, "Prevalence and maintenance of automated functional tests for web applications," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 141–150.

[2] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving gui-directed test scripts," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 408–418.

[3] F. Ricca, M. Leotta, and A. Stocco, "Chapter three - three open problems in the context of E2E web testing and a vision: NEONATE," *Adv. Comput.*, vol. 113, pp. 89–133, 2019.

[4] M. Nass, E. Alégroth, and R. Feldt, "Why many challenges with gui test automation (will) remain," *Information and Software Technology*, vol. 138, p. 106625, 2021.

[5] M. Leotta, B. García, F. Ricca, and J. Whitehead, "Challenges of end-to-end testing with selenium webdriver and how to face them: A survey," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2023, pp. 339–350.

[6] M. Hammoudi, G. Rothermel, and P. Tonella, "Why do record/replay tests of web applications break?" in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 180–190.

[7] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 272–281.

[8] E. Alégroth and R. Feldt, "On the long-term use of visual gui testing in industrial practice: A case study," *Empirical Softw. Engg.*, vol. 22, no. 6, p. 2937–2971, dec 2017.

[9] S. Balsam and D. Mishra, "Web application testing—challenges and opportunities," *Journal of Systems and Software*, vol. 219, p. 112186, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121224002309

[10] M. Hu and A. Trofimov, "Course design of introducing selenium webdriver," in *2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2024, pp. 340–348.

[11] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An empirical analysis of ui-based flaky tests," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21. IEEE Press, 2021, p. 1585–1597.

[12] F. Macklon, M. Viggiato, N. Romanova, C. Buzon, D. Paas, and C. Bezemer, "A taxonomy of testable HTML5 canvas issues," *IEEE Trans. Software Eng.*, vol. 49, no. 6, pp. 3647–3659, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2023.3270740

[13] K. Presler-Marshall, E. Horton, S. Heckman, and K. Stolee, "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)*, 2019, pp. 7–13.

[14] S. Fischer, "Insights from building a GUI testing tool," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023*, T. Zhang, X. Xia, and N. Novielli, Eds. IEEE, 2023, pp. 921–924.

[15] D. Olianas, M. Leotta, F. Ricca, and L. Villa, "Reducing flakiness in end-to-end test suites: An experience report," in *Quality of Information and Communications Technology - 14th International Conference, QUATIC 2021, Algarve, Portugal, September 8-11, 2021, Proceedings*, ser. Communications in Computer and Information Science, A. C. R. Paiva, A. R. Cavalli, P. V. Martins, and R. Pérez-Castillo, Eds., vol. 1439. Springer, 2021, pp. 3–17.

[16] T. Li, R. Huang, C. Cui, D. Towey, L. Ma, Y. Li, and W. Xia, "A survey on web application testing: A decade of evolution," *CoRR*, vol. abs/2412.10476, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2412.10476

[17] R. K. Yandrapally and A. Mesbah, "Fragment-based test generation for web apps," *IEEE Trans. Softw. Eng.*, vol. 49, no. 3, p. 1086–1101, mar 2023.

[18] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "Water: Web application test repair," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ser. ETSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 24–29.

[19] M. Hammoudi, G. Rothermel, and A. Stocco, "Waterfall: an incremental approach for repairing record-replay tests of web applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 751–762.

[20] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Using multi-locators to increase the robustness of web test cases," in *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–10.

[21] M. Leotta, F. Ricca, and P. Tonella, "Sidereal: Statistical adaptive generation of robust locators for web testing," *Softw. Test. Verification Reliab.*, vol. 31, no. 3, 2021.

[22] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Robula+: an algorithm for generating robust xpath locators for web testing," *J. Softw. Evol. Process.*, vol. 28, no. 3, pp. 177–204, 2016.

[23] M. Nass, E. Alégroth, R. Feldt, M. Leotta, and F. Ricca, "Similarity-based web element localization for robust test automation," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 3, apr 2023.

[24] A. Stocco, R. Yandrapally, and A. Mesbah, "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*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 503–514.

[25] X. Qi, X. Qian, and Y. Li, "Semantic test repair for web applications," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1190–1202.

[26] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic web testing using curiosity-driven reinforcement learning," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21. IEEE Press, 2021, p. 423–435.

[27] X. Chang, Z. Liang, Y. Zhang, L. Cui, Z. Long, G. Wu, Y. Gao, W. Chen, J. Wei, and T. Huang, "A reinforcement learning approach to generating test cases for web applications," in *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*, 2023, pp. 13–23.

[28] D. Olianas, M. Leotta, and F. Ricca, "Sleepreplacer: a novel tool-based approach for replacing thread sleeps in selenium webdriver test code," *Softw. Qual. J.*, vol. 30, no. 4, pp. 1089–1121, 2022.

[29] X. Liu, Z. Song, W. Fang, W. Yang, and W. Wang, "Wefix: Intelligent automatic generation of explicit waits for efficient web end-to-end flaky tests," in *Proceedings of the ACM on Web Conference 2024*, ser. WWW '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 3043–3052.

[30] X. Deng, Y. Gu, B. Zheng, S. Chen, S. Stevens, B. Wang, H. Sun, and Y. Su, "Mind2web: towards a generalist agent for the web," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2024.

[31] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 561–570. [Online]. Available: https://doi.org/10.1145/1985793.1985870

[32] Selenium, "Selenium automation framework," Nov. 2024. [Online]. Available: https://www.selenium.dev/

[33] P. Gao, Y. Xu, F. Song, and T. Chen, "Model-based automated testing of javascript web applications via longer test sequences," *Frontiers Comput. Sci.*, vol. 16, no. 3, p. 163204, 2022. [Online]. Available: https://doi.org/10.1007/s11704-020-0356-7

[34] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow, "jäk: Using dynamic analysis to crawl and test modern web applications," in *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*, ser. Lecture Notes in Computer Science, H. Bos, F. Monrose, and G. Blanc, Eds., vol. 9404. Springer, 2015, pp. 295–316. [Online]. Available: https://doi.org/10.1007/978-3-319-26362-5_14

[35] G. Zhang and J. Zhao, "Scenario testing of angularjs-based single page web applications," in *Current Trends in Web Engineering - ICWE 2019 International Workshops, DSKG, KDWEB, MATWEP, Daejeon, South Korea, June 11, 2019, Proceedings*, ser. Lecture Notes in Computer Science, M. Brambilla, C. Cappiello, and S. H. Ow, Eds., vol. 11609. Springer, 2019, pp. 91–103. [Online]. Available: https://doi.org/10.1007/978-3-030-51253-8_10

[36] S. Sampath and S. Sprenkle, "Advances in web application testing, 2010-2014," *Adv. Comput.*, vol. 101, pp. 155–191, 2016. [Online]. Available: https://doi.org/10.1016/bs.adcom.2015.11.006

[37] T. Shi, H. Ma, G. Chen, and S. Hartmann, "Cost-effective web application replication and deployment in multi-cloud environment," *IEEE Trans. Parallel Distributed Syst.*, vol. 33, no. 8, pp. 1982–1995, 2022. [Online]. Available: https://doi.org/10.1109/TPDS.2021.3133884

[38] S. Development, "Stale element reference exception," Aug. 2024. [Online]. Available: https://www.selenium.dev/documentation/webdriver/troubleshooting/errors/#stale-element-reference-exception

[39] Similar Web, "The world's leading ai-powered digital data company," Mar. 2025. [Online]. Available: https://www.similarweb.com/

[40] B. Zheng, B. Gou, J. Kil, H. Sun, and Y. Su, "Gpt-4v(ision) is a generalist web agent, if grounded," in *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=piecKJ2DlB

[41] J. Y. Koh, R. Lo, L. Jang, V. Duvvur, M. C. Lim, P. Huang, G. Neubig, S. Zhou, R. Salakhutdinov, and D. Fried, "Visualwebarena: Evaluating multimodal agents on realistic visual web tasks," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 881–905. [Online]. Available: https://doi.org/10.18653/v1/2024.acl-long.50

[42] W. Hong, W. Wang, Q. Lv, J. Xu, W. Yu, J. Ji, Y. Wang, Z. Wang, Y. Dong, M. Ding, and J. Tang, "Cogagent: A visual language model for GUI agents," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2024, Seattle, WA, USA, June 16-22, 2024*. IEEE, 2024, pp. 14 281–14 290. [Online]. Available: https://doi.org/10.1109/CVPR52733.2024.01354

[43] B. Kondracki and N. Nikiforakis, "Smudged fingerprints: Characterizing and improving the performance of web application fingerprinting," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/kondracki

[44] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford, "CAPTCHA: using hard AI problems for security," in *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, ser. Lecture Notes in Computer Science, E. Biham, Ed., vol. 2656. Springer, 2003, pp. 294–311. [Online]. Available: https://doi.org/10.1007/3-540-39200-9_18

[45] S. IDE, "Open source record and playback test automation for the web," Mar. 2025. [Online]. Available: https://www.selenium.dev/selenium-ide/

[46] Jinja, "Jinja is a fast, expressive, extensible templating engine. special placeholders in the template allow writing code similar to python syntax," Mar. 2025. [Online]. Available: https://jinja.palletsprojects.com/en/stable/

[47] E. Waits, "Selenium explicit waits," Nov. 2024. [Online]. Available: https://www.selenium.dev/documentation/webdriver/waits/#explicit-waits

[48] C. Utz, M. Degeling, S. Fahl, F. Schaub, and T. Holz, "(un)informed consent: Studying GDPR consent notices in the field," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 973–990. [Online]. Available: https://doi.org/10.1145/3319535.3354212

[49] R. Feldt, "Do system test cases grow old?" in *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*. IEEE Computer Society, 2014, pp. 343–352. [Online]. Available: https://doi.org/10.1109/ICST.2014.47

[50] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015, pp. 179–190. [Online]. Available: https://doi.org/10.1145/2786805.2786843

[51] E. Alégroth, R. Feldt, and P. Kolström, "Maintenance of automated test suites in industry: An empirical study on visual gui testing," *Information and Software Technology*, vol. 73, pp. 66–80, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584916300118

[52] F. Shao, R. Xu, W. A. Haque, J. Xu, Y. Zhang, W. Yang, Y. Ye, and X. Xiao, "Webevo: taming web application evolution via detecting semantic structure changes," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 16–28. [Online]. Available: https://doi.org/10.1145/3460319.3464800

[53] Y. Lin, G. Wen, and X. Gao, "Automated fixing of web UI tests via iterative element matching," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 1188–1199.

[54] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 712–721. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606617

[55] H. Zhang, Y. Tang, M. Lamothe, H. Li, and W. Shang, "Studying logging practice in test code," *Empir. Softw. Eng.*, vol. 27, no. 4, p. 83, 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10139-0

[56] K. Krippendorff and J. L. Fleiss, "Reliability of binary attribute data," pp. 142–144, 1978.

[57] G. Marzi, M. Balzano, and D. Marchiori, "K-alpha calculator–krippendorff's alpha calculator: A user-friendly tool for computing krippendorff's alpha inter-rater reliability coefficient," *MethodsX*, vol. 12, p. 102545, 2024.

[58] D. Clerissi, G. Denaro, M. Mobilio, and L. Mariani, "Dbinputs: Exploiting persistent data to improve automated gui testing," *IEEE Transactions on Software Engineering*, vol. 50, no. 9, pp. 2412–2436, 2024.

[59] Z. Xu, Y. Lin, Q. Li, and S. H. Tan, "Guiding chatgpt to fix web UI tests via explanation-consistency checking," *CoRR*, vol. abs/2312.05778, 2023.

[60] M. Nass, E. Alégroth, and R. Feldt, "Improving web element localization by using a large language model," *Softw. Test. Verification Reliab.*, vol. 34, no. 7, 2024. [Online]. Available: https://doi.org/10.1002/stvr.1893

[61] B. Zheng, B. Gou, J. Kil, H. Sun, and Y. Su, "Gpt-4v(ision) is a generalist web agent, if grounded," in *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024.

[62] J. Yang, H. Zhang, F. Li, X. Zou, C. Li, and J. Gao, "Set-of-mark prompting unleashes extraordinary visual grounding in GPT-4V," *CoRR*, vol. abs/2310.11441, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.11441

[63] J. Imtiaz, M. Z. Iqbal, and M. U. Khan, "An automated model-based approach to repair test suites of evolving web applications," *J. Syst. Softw.*, vol. 171, p. 110841, 2021.

[64] Z. Long, G. Wu, X. Chen, W. Chen, and J. Wei, "Webrr: Self-replay enhanced robust record/replay for web application testing," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1498–1508.

[65] S. Brisset, R. Rouvoy, L. Seinturier, and R. Pawlak, "Erratum: Leveraging flexible tree matching to repair broken locators in web automation scripts," *Inf. Softw. Technol.*, vol. 144, p. 106754, 2022.