Investigating the Adoption and Maintenance of Web GUI Testing: Insights from GitHub Repositories

Sergio Di Meglio[a], Luigi Libero Lucio Starace[a], Valeria Pontillo[b,c], Ruben Opdebeeck[c], Coen De Roover[c], Sergio Di Martino[a]

[a] *Department of Electrical Engineering and Information Technology, University of Naples Federico II, Italy*
Email: (sergio.dimeglio, luigiliberolucio.starace, sergio.dimartino)@unina.it
[b] *Gran Sasso Science Institute (GSSI) — L'Aquila, Italy*
Email: valeria.pontillo@gssi.it
[c] *Software Languages (SOFT) Lab — Vrije Universiteit Brussel, Belgium*
Email: (ruben.denzel.opdebeeck, coen.de.roover)@vub.be

Abstract

Context: Web GUI testing is a quality assessment practice aimed at evaluating the functionality of web applications from the perspective of its end users. While prior studies have explored the technical challenges of automated Web GUI testing, fewer works have explored how this practice is applied in real-world web apps.

Objective: This study aims to investigate the adoption, characteristics, and maintenance of automated web GUI testing practices in open-source web applications, focusing on identifying trends and providing actionable insights for researchers and practitioners.

Method: We conducted a large-scale empirical analysis of 472 web applications on the GitHub platform, developed in JAVA, JAVASCRIPT, PYTHON, and TYPESCRIPT. These projects use popular browser automation frameworks like SELENIUM, PLAYWRIGHT, CYPRESS, and PUPPETEER. The study involved examining project characteristics and analyzing the co-evolution and maintenance of automated web GUI tests over time.

Result: Our findings empirically document automated web GUI testing adoption patterns in open-source projects, providing insights into the practical drivers behind both initial framework adoption and migration between different testing frameworks. Projects incorporating these tests generally show higher community engagement and consistent maintenance efforts. The analysis reveals that Web GUI tests tend to co-evolve with the underlying applications, reflecting their integration into the development lifecycle.

Conclusion: The study provides valuable insights into the prevalence and maintenance of Web GUI testing, highlighting practical implications for improving testing practices. Our findings can guide further research on the matter and support practitioners in enhancing their testing strategies.

Keywords: Web GUI Testing; Web Applications; Browser Automation Frameworks; Empirical Study.

1. Introduction

Web Graphical User Interface (GUI) testing is a quality assessment practice that aims at evaluating a web application's quality by executing and observing its user-facing behavior through web browser interactions [7, 19].

By replicating realistic end-user interactions, such as navigating through pages, clicking links, filling and submitting forms, etc., this empirical approach aims at identifying deviations from expected behavior [21], ultimately providing an informed assessment of web app quality.

Web GUI testing can be conducted manually by human testers interacting with the app or through automation. While manual testing remains valuable for exploratory scenarios [18], automation is often employed to improve efficiency, scalability, and consistency [66]. Among the various automated approaches (described in Section 2.1), this study focuses specifically on *scripted automated web GUI testing*, in which test code is written to reproduce realistic user interactions leveraging browser automation frameworks such as SELENIUM, PLAYWRIGHT, or CYPRESS. This paradigm remains the most established and consolidated for building maintainable web GUI test suites and enables transparent, reproducible analysis of test code evolution [35].

Despite its importance, empirical studies about how scripted automated web GUI testing is adopted and maintained in real-world projects remain limited. Prior work has primarily focused on the technical challenges of scripted automated web GUI testing, such as fragility (the tendency of test scripts to break due to minor UI or DOM changes) and flakiness (non-deterministic failures caused by asynchronous content or timing issues) [5, 36, 49, 52]. As a result, many studies propose techniques to improve test robustness or automate test creation, while the broader question of how these tests are actually used and maintained over time is mostly neglected.

Additionally, the evolution and co-maintenance of unit tests have been extensively studied in the literature [41, 64], revealing important insights into how testing practices shape and reflect software development. However, comparable evidence for scripted automated web GUI tests remains scarce. This gap is further compounded by limitations in prior datasets. Many existing studies rely on outdated or student-developed applications, or identify web GUI test scripts through inherently imprecise heuristics without manual validation [27, 55]. These constraints have hindered large-scale, reliable observations of scripted automated web GUI testing practices in the wild.

To bridge this gap, this paper leverages the curated dataset from our recent study [23] to conduct a large-scale empirical study of 472 open-source web applications hosted on GITHUB, covering four widely-used browser automation frameworks (SELENIUM, PLAYWRIGHT, CYPRESS, and PUPPETEER). The primary objective is to provide a comprehensive analysis of how Web GUI testing is adopted in real-world applications, the characteristics of projects that integrate these tests, and the extent to which these tests co-evolve with their respective applications.

Our findings indicate that, in our dataset, scripted automated Web GUI testing is more commonly adopted in recent projects, with most cases appearing from 2018 onwards, aligning with the emergence of modern frameworks such as PLAYWRIGHT and CYPRESS. While this trend may partly reflect the timeline of these tools, we also find evidence of migration from older frameworks, often motivated by the need to reduce flakiness, address deprecated APIs, or unify their tooling. Web applications that adopt scripted GUI testing typically exhibit higher development activity, more contributors, and have more active issue trackers, often with a larger volume of reported and managed issues. Moreover, we observe that Web GUI test scripts tend to co-evolve with the application, require relatively infrequent maintenance, and are primarily maintained by a small group of core contributors.

These tests often undergo targeted updates, such as assertion or selector changes, to preserve stability over time.

Summarizing, this paper makes the following contributions:

1. A large-scale empirical analysis of scripted automated Web GUI testing adoption in 472 open-source web applications, focusing on the use of popular browser automation frameworks, the characteristics of projects that integrate these tests, and the co-evolution of test and application code;
2. A set of actionable insights and implications to improve scripted automated Web GUI testing practices and further explore the adoption patterns and co-evolutionary dynamics of these tests;
3. An online appendix [22], which provides all data used to conduct our study, enabling replication and extension of our research by the community.

The remainder of this paper is organized as follows. Section 2 presents the background and related work on Web GUI testing adoption and maintenance. Section 3 introduces the research questions that guide our study and the context of the study. Section 4 details the research method employed and the results obtained for each research question. Section 5 discusses the implications of our findings for both researchers and practitioners, while Section 6 reports the limitations and the mitigation strategies applied. Finally, Section 7 concludes the paper and outlines potential future research directions.

2. Background and Related Work

This section provides background notions on web GUI testing, with particular emphasis on automated scripted testing approaches that constitute the core of our study. Following this technical grounding, we present an overview of the related works, positioning our study within the existing body of knowledge.

2.1. Web GUI Testing

Web GUI testing is a quality assurance technique that evaluates the behavior of a web application through simulating and monitoring user interactions within a web browser [7]. This practice aims at detecting discrepancies from expected outcomes [21], offering an informed assessment of the application's functionality.

Web GUI testing techniques can be broadly categorized into manual and automated approaches. Manual testing involves human testers executing test cases step-by-step, observing the application's behavior, and verifying outcomes against expected results. While flexible for exploratory testing [18] and adaptable to UI changes, manual testing becomes inefficient and error-prone for large-scale applications [13]. Automated testing, on the other hand, employs specialized tools to simulate user interactions programmatically, enabling more scalable and repeatable testing processes. Automated approaches can be further divided into three main methodologies: (1) script-based testing; (2) low-code and capture-and-replay approaches; and (3) scriptless approaches.

Scripted approaches require testers to write code that generates synthetic user interactions, usually through browser automation frameworks like SELENIUM or PLAYWRIGHT. These frameworks provide APIs for locating elements in the web pages, performing actions, and verifying outcomes through assertions. Low-code and capture-and-replay solutions offer a middle ground, allowing testers to define re-executable scenarios without writing code directly, but respectively by recoding interactions in an instrumented web browser (capture-and-replay) or by leveraging visual editors (low-code approaches) [31]. At the most automated end of the spectrum, scriptless approaches [10, 62] eliminate the need to manually specify interaction sequences altogether, with tools dynamically identifying elements and generating test actions during execution.

The choice between these methodologies involves trade-offs. Scripted testing provides precise control and is well-suited for complex validation scenarios, but requires programming expertise and ongoing maintenance. Low-code solutions reduce the technical barrier while maintaining some flexibility, but these advantages are generally associated with less maintainable tests [17]. Scriptless approaches can adapt more easily to UI changes and can guarantee a basic level of test coverage with minimal effort, but may lack precision in validating specific business logic. While all three approaches have their merits, this study focuses specifically on scripted testing.

In the domain of scripted automated web GUI testing[1], several browser automation frameworks have been developed, like SELENIUM [53], CYPRESS [14], PLAYWRIGHT [48], and PUPPETEER [50]. Each of these frameworks is compatible with multiple programming languages and offers distinct features tailored to different testing needs.

SELENIUM, one of the earliest proposed frameworks, relies on the *WebDriver* component to interact with web browsers. This component provides a flexible API that manages the behavior of web browsers during automated testing processes [51]. However, this approach requires a dedicated driver for each browser (e.g., CHROMEDRIVER for Chrome), making SELENIUM highly versatile but also adding setup complexity and potential compatibility issues [35]. In contrast, CYPRESS and PLAYWRIGHT operate within the browser environment, eliminating the need for a separate *WebDriver*, reducing configuration overhead. PUPPETEER also avoids the use of a *WebDriver* but is specifically designed for Chromium-based browsers, leveraging the Chrome DevTools Protocol for automation.

The choice of a browser automation framework depends on the specific requirements and constraints of a project. Regardless of the framework used, automated tests rely heavily on locators to accurately identify web elements for interaction. Locators leverage the layout properties embedded in HTML pages to identify elements based on attributes such as id, class, name, href, textual content, or positioning within the Document Object Model (DOM) [36]. For example, locators may use query expressions such as XPath or CSS selectors to identify elements [57]. Listing 1 shows an example of a web GUI test developed in JAVA using SELENIUM. In this example, the test navigates to a local web page, interacts with form fields identified by locators such as id, name, and XPath, enters input data, and simulates a click on the submit button.

Despite the advances in web browser automation frameworks, web GUI testing faces significant challenges

[1]For the sake of simplicity, we use the term *web GUI testing* throughout the remainder of the paper to refer specifically to *scripted automated web GUI testing*.

```java
@Test
public void signUpTest(){
  driver driver = new ChromeDriver();
  driver.get("http://localhost:8080/");
  driver.findElement(By.id("fname")).sendKeys("John");
  driver.findElement(By.name("lname")).sendKeys("Doe");
  driver.findElement(By.xpath("//form/input[3]")).sendKeys("123");
  driver.findElement(By.linkText("Submit")).click();
}
```

Listing 1: Web GUI Test implemented in Java using SELENIUM [21]

such as slowness [11, 35] and fragility [12, 21, 34]. The former refers to the amount of time required to run a web GUI test compared to other testing activities. The latter remains a persistent issue, with scripts frequently breaking due to UI changes, dynamic content updates, or modifications in the underlying application structure [12, 21, 34]. This fragility increases maintenance overhead, making it costly to sustain automated test suites over time [35].

Beyond technical challenges, developing and maintaining effective web GUI tests using a scripted approach requires specific expertise in automation tools. As a result, many companies, and particularly smaller organizations with limited resources, struggle to justify the investment in web GUI testing activities [11, 35].

2.2. Related Work

Literature on web GUI testing primarily focuses on specific testing frameworks or techniques, rather than examining how web GUI tests are integrated and applied in real-world software projects.

Balsam and Mishra [5] performed a systematic literature review to group all the frameworks and techniques proposed. Additionally, the authors summarized the main challenges behind testing web applications. Similarly, Junior et al. [30] surveyed 222 practitioners across various companies, revealing that manual web GUI testing remains widespread despite increasing adoption of SELENIUM. Their findings also highlighted key challenges, including tool limitations and the inherent complexity of web GUI test automation, which can hinder the widespread adoption of this approach.

Other works focused on supporting web GUI testing practices by addressing specific challenges. Biagiola et al. [8] proposed TEDD, a tool that automatically detects and validates test dependencies in end-to-end (E2E) web test suites. The authors conducted an empirical evaluation on six different projects that implemented SELENIUM as a browser automation framework. Stocco et al. [58] proposed approaches to automatically repair broken GUI tests or to support the generation of maintainable test code [57]. Moran et al. [44] proposed an approach to localize the root cause of flakiness in web tests using spectrum-based localization techniques that analyze test execution under different combinations of external environmental factors. More recently, Di Meglio et al. [21] presented a preliminary empirical study to estimate the extent to which a web test is prone to fragility.

Lastly, the body of work on web GUI testing adoption and maintenance is relatively limited, with most works focusing on desktop or mobile applications or being often tied to specific tools or industrial contexts. These works are the most closely related to our study. Table 1 summarizes the articles, along with the main differences between

them and our study.

Alégroth et al. [2] examined the maintenance effort for Visual GUI Testing (VGT) in companies like Siemens and Saab, highlighting substantial maintenance costs and the benefits of frequent updates over infrequent ones. This work presents some analogies with our study, i.e., both works focus on maintenance activities. Additionally, their findings underscore the importance of web GUI testing in industrial contexts and emphasize the need for broader, large-scale studies of the adoption and maintenance of GUI testing. Nonetheless, the study of Alégroth et al. [2] was conducted on VGT, which utilizes computer vision rather than traditional DOM-based methods, such as SELENIUM or CYPRESS. In a follow-up study, Alégroth et al. [1] reported on the long-term use of VGT at SPOTIFY, highlighting its context-dependent viability and providing practical guidelines for industry.

Grechanik et al. [28] compared the manual versus automated maintenance tools for web GUI test suites, showing that automation tools reduce effort and improve consistency, albeit with upfront costs. This work shares some similarities with our study, as both address GUI test maintenance; however, it differs in scope and context. Specif-

Table 1: Comparison between our study and the most closely related papers..

Related Work	Main Focus	Differences
Alégroth et al. [2]	A mixed-method study to empirically identify the costs and factors associated with the maintenance activities of automated GUI-based testing. The study was conducted in two industries, i.e., Siemens and Saab.	• The approach is different; the authors drew their conclusion using interviews within a company and analyzing the test suite of an application in another one; • The study leverages Visual GUI Testing (VGT) instead of traditional DOM-based methods, e.g., SELENIUM.
Grechanik et al. [28]	A comparison study between manual and automated maintenance tools for web GUI tests.	• The approach is different; the authors drew their conclusion using a case study with 45 practitioners; • Their study focused on desktop applications and commercial tools.
Kresse and Kruse [33]	A comparison study of three desktop GUI testing tools (EGGPLANT, QF-TEST, and TEST-COMPLETE) for analyzing development and maintenance efficiency.	• The study focuses on a cost-driven comparison of commercial tools; • The analysis was performed on a single web application.
Shewchuk and Garousi [54]	A longitudinal study on maintaining a web GUI test suite with IBM Rational Functional Tester.	• The study focuses on the capabilities of the maintenance tool rather than web GUI tests. • The study was conducted on a single web GUI test suite.
Cristophe et al. [11]	An empirical study in which the authors analyzed 287 open-source Java web applications using SELENIUM to investigate the prevalence of automated functional tests. They also conducted a longitudinal analysis on 8 selected projects to understand how these tests evolve and which parts of the tests are most frequently maintained.	• Our study focuses on different browser automation frameworks, i.e., SELENIUM, CYPRESS, PLAYWRIGHT, and PUPPETEER; • Our work is based on a larger number of web applications; • Statistical analyses to analyze the correlation between project characteristics and the adoption of web GUI Testing.

ically, their controlled experiment focused on desktop applications and commercial tools (e.g., QUICKTEST PRO), whereas our study investigates large-scale, real-world maintenance practices of automated web GUI tests using open-source frameworks such as SELENIUM, CYPRESS, and PLAYWRIGHT.

Kresse and Kruse [33] compared three desktop GUI testing tools: EGGPLANT, QF-TEST, and TESTCOMPLETE, finding significant differences in development and maintenance efficiency, and stressing the importance of early tool selection for long-term costs. Both studies address the maintenance of GUI tests; however, their analysis focuses on a cost-driven comparison of commercial tools, while our study investigates large-scale, open-source web GUI testing practices and the evolution of automation frameworks over time.

Similarly, Shewchuk and Garousi [54] conducted a longitudinal study on maintaining a web GUI test suite with IBM Rational Functional Tester, highlighting the high maintenance demands of GUI automation and stressing the need for modular test design and continuous adaptation. This work presents some analogies with our study, as both address test maintenance and evolution over software versions. However, while their analysis focused on a single desktop system and a commercial testing tool, our study investigates large-scale, real-world web GUI testing practices across hundreds of open-source projects and multiple open-source frameworks.

Although previous studies offer valuable insights into test script collection and test failure analysis, they do not systematically investigate the prevalence, characteristics, or long-term maintenance of Web GUI tests in large-scale, real-world projects. While several studies have investigated the adoption and evolution of unit testing [38, 41, 64, 65], research on the use and maintenance of other forms of automated testing in web development is still emerging. For instance, a recent study by Di Meglio et al. [24] analyzed the adoption and maintenance of web performance tests in open-source projects. Although focused on a different class of tests, their findings emphasize the need for empirical research on real-world testing practices beyond unit testing. To the best of our knowledge, the only empirical study explicitly focusing on web GUI testing remains the now-outdated work by Christophe et al. [11], which analyzed 287 open-source Java web applications using SELENIUM to assess the prevalence of automated functional tests.

While we adopt the same approach proposed by Christophe et al. [11] to investigate test maintenance (see Section 4.3), our study is based on a significantly larger sample of 472 web applications that employ a variety of browser automation frameworks, including SELENIUM, CYPRESS, and PUPPETEER. In addition, compared to the works reported in Table 1, our study offers a broader and more generalizable perspective by focusing on scripted web GUI testing in open-source web applications. This allows us to uncover adoption trends, migration patterns, and maintenance behaviors across diverse ecosystems. Furthermore, while prior studies often focus on maintenance effort or tool evaluation, our work combines quantitative and qualitative analyses to investigate co-evolution dynamics, issue management, and contributor roles. In doing so, we complement existing literature with large-scale empirical evidence on how web GUI testing is adopted and sustained in practice.

3. Goal and Research Questions

The *goal* of this study is to explore web GUI testing practices in web applications that use widely adopted browser automation frameworks, specifically SELENIUM, PLAYWRIGHT, CYPRESS, and PUPPETEER. By investigating these frameworks, this study aims to deepen our understanding of web GUI test are adopted, maintained, and co-evolve with the application code over time. These frameworks were selected due to their extensive use in both research and industry [16, 43], as well as their robust support for creating and running automated tests across various platforms and browsers.

In this study, we define *adoption* as the point at which a project introduces at least one scripted web GUI test using a browser automation framework. Operationally, this is identified by the first commit that adds such a test file. *Maintenance* refers to any subsequent activity that modifies, updates, or deletes existing web GUI test files. This includes changes to assertions, selectors, or test structure, and is measured through commit-level analysis of test evolution. Finally, we define *co-evolution* as the temporal coupling between changes to web GUI test files and changes to application code. This concept captures the extent to which test and production code evolve in tandem, and is analyzed through commit history and association rule mining.

Our study revolves around three research questions. First, we analyze the adoption of web GUI testing in open-source web applications. This quantitative investigation seeks to determine how frequently these tests are integrated into software development practices. While previous research has emphasized the importance of web GUI testing, it has also highlighted several challenges that hinder its widespread adoption, such as fragility, slowness, and unpredictability. However, due to the lack of comprehensive data on real web applications, the extent to which web GUI testing is actually adopted remains unclear. By analyzing adoption trends, we provide a current snapshot of how web GUI testing is being employed in open-source web applications. So we asked:

RQ$_1$. *To what extent are web GUI tests adopted in open-source web applications?*

We continue our analysis by investigating whether the adoption of web GUI testing practices would have any effect on the nature and characteristics of the project. Multiple factors, such as project size, development activity, and community engagement, may influence the decision to adopt web GUI testing. By identifying the characteristics of projects that integrate web GUI testing, we can better understand whether factors like project maturity, team size, or repository activity correlate with testing adoption. Understanding these relationships could provide valuable insights for developers evaluating the benefits of web GUI testing and help researchers assess whether certain types of projects are inherently better suited for these practices. So we asked:

RQ$_2$. *What are the main characteristics of projects adopting web GUI testing?*

Finally, we aim to investigate the co-evolution of web GUI tests with the application by examining how test suites are maintained as the application changes over time. This includes analyzing the lifespan of web GUI test cases, their modification frequency, and whether tests change after application updates. web GUI testing is often

perceived as fragile and costly to maintain, yet empirical evidence on its long-term sustainability remains limited. Investigating how web GUI tests co-evolve with the underlying application can provide insights into their maintenance burden and whether they stay aligned with software changes. If web GUI tests tend to be abandoned or require frequent changes, this could indicate a need for better test maintenance strategies or tooling improvements. Understanding these dynamics can help practitioners refine their testing strategies and inform research on improving test robustness. So, we asked:

> **RQ$_3$.** *Do web GUI tests co-evolve with the web application, and how are these tests maintained as the application evolves?*

Context Selection. The *context* of our study is the recently published E2EGit dataset [23], a publicly available and curated collection of end-to-end web tests mined in May 2024 from non-trivial open-source web applications hosted on GitHub. All analyses presented in this paper are conducted on the entire E2EGit dataset, without any additional filtering or sampling.

As discussed in [23], the E2EGit dataset was constructed through a rigorous multi-stage mining pipeline. Initially, over 1.4 million GitHub repositories were sampled using the SEART tool [15]. To ensure relevance and non-triviality, these repositories were then filtered based on established thresholds for commits (≥ 2000), contributors (≥ 10) and stargazers (≥ 100), and forks were excluded. This yielded 14,053 repositories, which were further refined to 5,563 web applications by detecting dependencies on widely adopted web frameworks (e.g.: Spring Boot, React, Angular, etc.) across four major programming languages: Java, JavaScript, Python and TypeScript.

Within these web applications repositories, web GUI tests were identified by detecting dependencies on popular browser automation frameworks (namely Selenium, Playwright, Cypress, Puppeteer) and verifying the presence of test engines (e.g., `JUnit`, `Mocha`, `Pytest`). This process resulted in the identification of 472 repositories containing 43,670 validated web GUI tests. The full E2EGit dataset is included in our replication package [22], along with all additional data produced during our investigation.

4. Research Method and Results

This section discusses the research methods employed to address the three main research questions targeted by our work and the results we obtained.

*4.1. **RQ**$_1$: To what extent are web GUI tests adopted in open-source web applications?*

Research Method. To answer **RQ**$_1$, we extract all the information related to the browser automation frameworks used by practitioners to perform web GUI testing to understand their adoption and prevalence. In particular, our research method consists of four steps:

1. *Analyzing Adoption Trends Over Time*: We first analyze the evolution of web GUI testing adoption, comparing the creation date of each repository with the timestamp of the first introduced web GUI test. This

analysis allows us to assess how early projects incorporate web GUI testing into their development workflows. Additionally, we investigate the adoption timeline of web GUI testing in relation to the release dates of major browser automation frameworks, providing insights into how the availability of these tools influence adoption trends over time.

2. *Examining Framework Distribution*: To gain insights into the prevalence of different browser automation frameworks, we analyze their distribution across web applications. This analysis allows us to identify the most commonly used frameworks and assess their adoption patterns within the collected projects.

3. *Investigating Framework Co-Adoption and Migration*: Since some projects could leverage multiple browser automation frameworks, we conduct an additional analysis to explore possible co-adoption and migration patterns. Specifically, we analyze whether projects switched from one framework to another over time or consistently used multiple frameworks in parallel to address diverse testing needs. To achieve this, we scan the commit history of each repository, identifying cases where a framework transition or new framework adoption occurs.

4. *Analyzing the Rationale Behind Adoption and Migration*: To further understand the motivations for adopting or migrating browser automation frameworks, we conduct a qualitative analysis of commit messages. Specifically, we collect commit messages associated with two key events: (i) the *adoption* of web GUI testing, identified by the first commit introducing a test along with the ten preceding commits, and (ii) the *migration* from one framework to another, again including the ten preceding commits for contextual information. To isolate relevant messages, we apply a keyword-based filtering strategy using a dedicated set of keywords for adoption (e.g., `add e2e`, `introduce test`, `cypress`, `selenium`, `puppeteer`, `playwright`) and for migration (e.g., `migration`, `refactor`, `replace`, `reimplement`). This keyword-based filtering yielded 588 and 481 commit messages, respectively. The complete list of the keywords for both adoption and migration is reported in our online appendix [22]. We then perform a manual qualitative analysis to identify the *"Why"* information in commit messages, as suggested by Tian et al. [59]. To do so, we follow an open coding protocol [26] to identify and categorize potential reasons behind adoption and migration in the 1,069 commit messages. Specifically, the first two authors of this study independently check all 1,069 commit messages by examining both the link contents and the commit messages for *"Why"* information. In cases where *"Why"* could not be found by both, we marked the message as unknown. During the analysis, each emerging reason is compared with existing ones to determine if it is a new reason through multiple comparison sessions. Finally, the two authors exchange ideas for the categorization and reach a consensus.

Analysis of the Results. Figure 1 aims to highlight when web GUI testing was adopted in relation to the creation year of each repository. Specifically, it provides insight into whether web GUI testing was integrated at the beginning of a project or introduced at a later stage. The x-axis of the matrix represents the year in which the repository

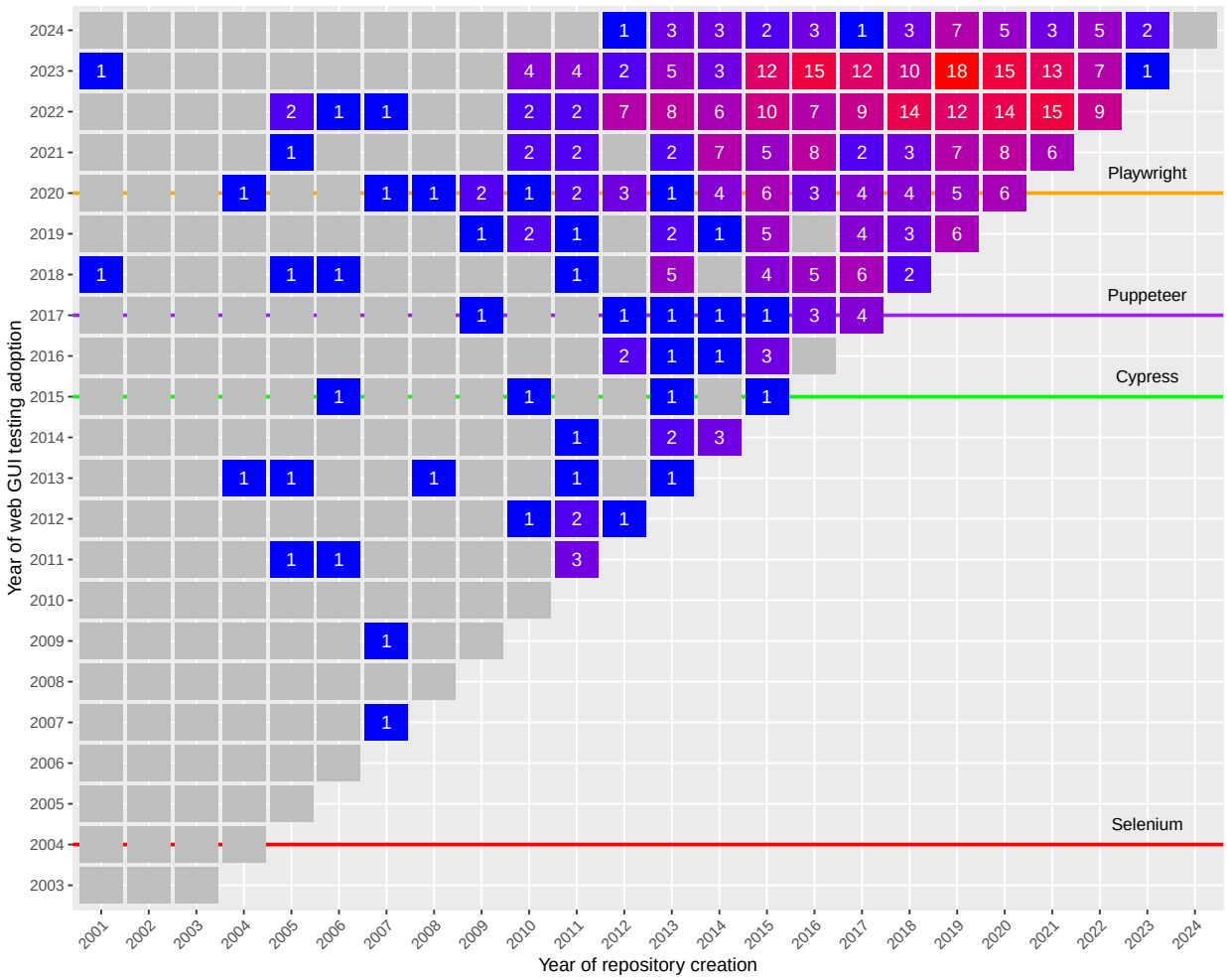

Figure 1: Repository creation and web GUI testing adoption over time. The x-axis of the matrix represents the year in which the repository was created. In contrast, the y-axis represents the year in which the repository first adopted web GUI testing. Each square shows the number of repositories created in a given year and the year they first adopted web GUI testing. Colored lines indicate the first stable releases of browser automation frameworks considered in the study.

was created. In contrast, the y-axis represents the year in which the repository first adopted web GUI testing. Each cell at position (x, y) indicates the number of repositories that were created in year x and adopted web GUI testing in year y. For example, the cell at $(2015, 2023)$ has a value of 12, meaning that 12 repositories created in 2015 adopted web GUI testing eight years later, in 2023.

The data shows that adoption of web GUI testing frameworks in the projects remained low until 2017, with only 12 projects adopting testing frameworks that year, followed by steady growth: 26 projects in 2018, 25 projects in 2019, 44 in 2020, 53 in 2021, peaking at 122 projects in 2023. This empirical pattern aligns temporally with the release timeline of newer frameworks: CYPRESS (2015), PUPPETEER (2017), and PLAYWRIGHT (2020). Notably, although SELENIUM was first released in 2004, in our dataset only two projects adopted web GUI testing respectively in 2007 and 2009, with adoption remaining minimal ($\leq$ 12 projects per year) until 2017. Furthermore, as shown

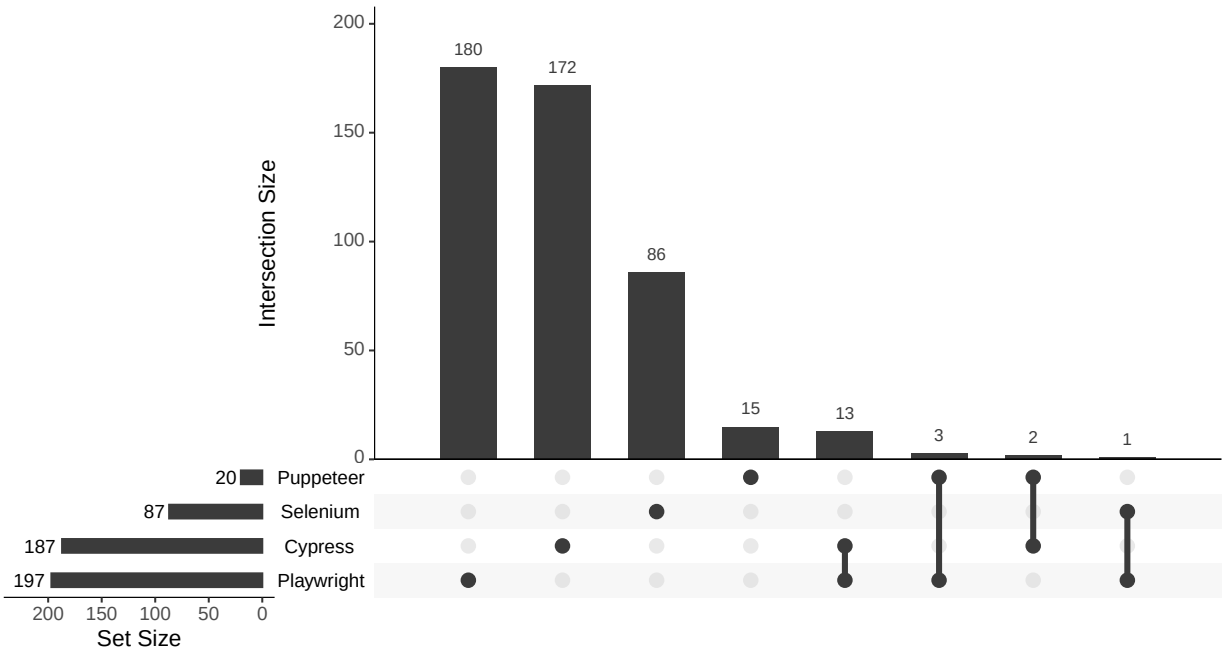

Figure 2: Distribution of the browser automation frameworks in the collected web application projects.

by the clustering of data points in recent years, the considered web application projects are increasingly adopting web GUI testing frameworks closer to their creation date, suggesting a possible trend towards incorporating automated testing earlier in the development lifecycle, which would align with the growing recognition of automated testing practices in modern software development workflows [35].

Figure 2 reports the results of the analysis of the distribution of browser automation frameworks. The analysis reveals that PLAYWRIGHT is currently the most widely adopted browser automation framework, used in 197 projects, closely followed by CYPRESS with 187 projects. In contrast, SELENIUM and PUPPETEER exhibit significantly lower adoption rates, being found in 87 and 20 projects, respectively. This trend highlights a shift towards modern JAVASCRIPT-based testing tools, likely driven not only by their developer-friendly APIs and seamless integration with contemporary development workflows, but also by the fact that a large portion of the analyzed repositories are themselves implemented in JAVASCRIPT or TYPESCRIPT. In such cases, developers may prefer to adopt testing tools that align with the same language ecosystem, facilitating better integration, reusability of utilities, and reduced context-switching during development.

Interestingly, the vast majority of projects (453 out of 472) rely on a single browser automation framework. On the other hand, a small subset of projects (19) adopt multiple frameworks, possibly to leverage complementary features or to support migration between tools.

Among the multi-framework setups, the CYPRESS + PLAYWRIGHT combination is the most prevalent, appearing in 13 projects. Other combinations are less frequent: PLAYWRIGHT + PUPPETEER in 3 projects, CYPRESS +

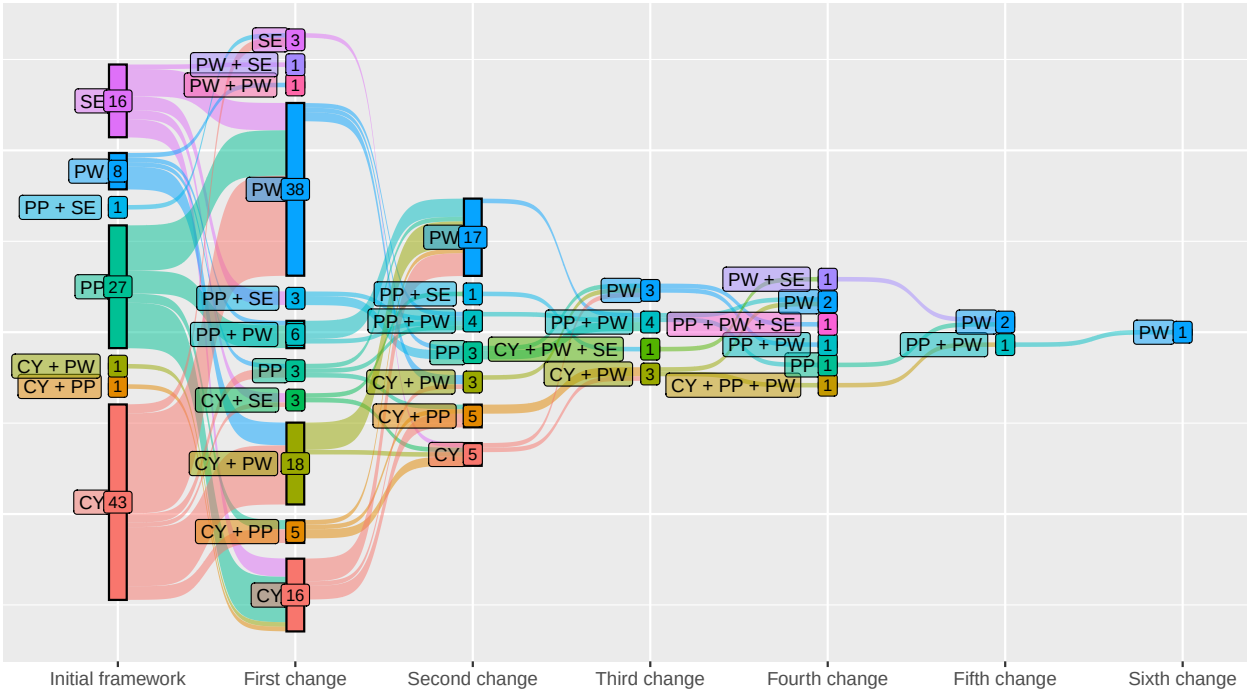

Figure 3: Analysis of the co-adoption pattern for projects with at least two different browser automation framework adoptions. Node labels represent framework combinations using acronyms: CY (Cypress), PW (Playwright), SE (Selenium), and PP (Puppeteer). Numbers represent the number of projects subjected to each transition in the different stages of framework adoption.

PUPPETEER in 2, and SELENIUM + CYPRESS in 1, indicating that such hybrid approaches are relatively rare and tailored to specific needs.

Concerning the analysis of framework co-adoption and migration frameworks, we observed that several projects consistently used the same framework throughout their history. Specifically, 78 projects started with and continued to use SELENIUM, 146 projects remained with CYPRESS, 133 projects stuck with PLAYWRIGHT, and 18 projects consistently used PUPPETEER. Additionally, 97 (21%) projects had experienced at least one framework change in their commit history—results are shown in Figure 3.

We can observe that CYPRESS and PUPPETEER represent the most common choices as initial browser automation frameworks, with a frequency of 43 and 27, respectively. Concerning SELENIUM and PLAYWRIGHT, these appear less frequently, while we also found two project that shows a framework combination (*"Cypress-Playwright"* and *"Cypress-Puppeteer"*) from the beginning. As projects evolve, a substantial number integrate additional frameworks over time. Notably, many projects initially using CYPRESS or PUPPETEER incorporate PLAYWRIGHT later in their development, as evidenced by transitions labeled as *"CY-PW"* or *"PP-PW"*. Indeed, PLAYWRIGHT turned out to be the most frequent migration path, occurring in 37 repositories from CYPRESS and 21 repositories from PUPPETEER, making PLAYWRIGHT the most popular destination overall and suggesting that developers tend to adopt the most recent browser automation frameworks. This result is partially confirmed by the evolution of the 16

projects that initially adopted SELENIUM. We can observe that these projects later added CYPRESS or PUPPETEER (as shown by the labels *"SE-CY"* and *"SE-PP"*) or completely migrated to the more recent frameworks.

In addition, while many projects have made at most two changes in the use of browser automation frameworks, there are some in our sample that have made five or more co-adoptions or migrations. In particular, this project[2] started with SELENIUM and, after five changes where the developers tried different combinations, i.e., *"CY-SE"*, *"SE-PP"*, *"SE-CY-PW"*, and *"SE-PW"*, they concluded the migration to PLAYWRIGHT, which suggests again the idea that developers use the most recent browser automation framework.

Interestingly, we also identified three outliers in our analysis, that initially adopted CYPRESS and then SELENIUM[3]. This reverse migration could be attributed to several possible reasons. First, SELENIUM offers broad cross-browser support, including compatibility with Internet Explorer, which CYPRESS does not natively support. Second, SELENIUM integrates better with more testing environments and programming languages. Finally, SELENIUM's maturity and strong community support may make it attractive for projects that require a stable and well-documented solution.

Considering the temporal dynamics of framework migrations, PLAYWRIGHT exhibits a particularly rapid adoption trajectory. Although it was released only in late 2020, the number of migrations toward PLAYWRIGHT increased sharply in the subsequent years. After a few early shifts in 2020 and 2021, we observed a notable rise in 2022 (23 projects) and 2023 (32 projects), followed by sustained activity in 2024 (14 projects). These migrations account for 62% of all observed transitions, underscoring how quickly the framework has become a preferred choice among developers. In contrast, migrations to CYPRESS began more gradually, with the first observed migration occurring in 2018, three years after the framework's introduction. In subsequent years, observed migrations have remained steady at a lower scale (typically 3–7 projects per year). PUPPETEER shows a similarly moderate but consistent trend since its 2017 release, with 2–5 migrations per year. These findings complement the migration analysis by showing not only that PLAYWRIGHT is the most frequent migration target, but also that shifts towards its adoption have been concentrated in a much shorter timeframe, suggesting a strong and more recent shift in developer preferences toward modern, cross-browser, and JavaScript-native testing frameworks.

As for the analysis of the rationale behind adoption and migration, we identified eight commit messages that provided insights into the adoption of GUI testing and five commit messages that explicitly revealed the motivation behind a migration between browser automation frameworks. Table 2 provides the link to each commit. In many cases, commit messages are either too generic or entirely unrelated to the actual content of the commit, making it difficult to infer the rationale behind test adoption or framework replacement. This result aligns with what is reported in the literature, i.e., developers believe they write high-quality commit messages, while they made the opposite [37].

Regarding adoption, we observe that web GUI testing is often employed to strengthen CI pipelines or enable

[2] https://github.com/Alfresco/alfresco-content-app

[3] The projects are: https://github.com/kiegroup/kie-tools and https://github.com/apache/incubator-kie-tools

Table 2: Overview of the commits that refer to the adoption or migration patterns.

ID	Commit	Category
1	apache/helix/commit/1c16a6f4cac3dcefa10a3beb14fabf14bf5cd547	Migration
2	automattic/sensei/commit/e9a9c0ba205d3e2f3793d8e19dd9e9a14e7323bc	Adoption
3	cockroachdb/cockroach/commit/c58279d4ce53c26987ef238e2ef268d6f6125f04	Adoption
4	expo/expo/commit/d002a1f54737cf04cfbedb289c140c3d6317adbd	Adoption
5	goharbor/harbor/commit/de6e517136d0d0120a49c1a7339addbcbc662314	Migration
6	gravitational/teleport/commit/e1e0b790961ef5da0aa0928dd70e81bac15f0d18	Adoption
7	huridocs/uwazi/commit/da9f2a5d7ae5d68b74d30eba203ef5a4c16d8d50	Migration
8	plan-player-analytics/plan/commit/b9fa29544d48cf92ec701f5568b585bb801f01b6	Adoption
9	pwa-builder/pwabuilder/commit/101758dd87dcce086b49732a8dbc71c9f189bab1	Adoption
10	serenity-js/serenity-js/commit/7b3c6c83d5caa48b4362dee0f30a154f00cb46e2	Adoption
11	serenity-js/serenity-js/commit/3c9aa4b16d223844116ffcb21d23f9cc8b96a793	Adoption
12	tryghost/koenig/commit/0511a3ebb910d35417378b6352eea2901810ce6b	Migration
13	tryghost/koenig/commit/baf9b666879ef05eb2e7fa4ef6be6204f071925a	Migration

automated health checks. Additionally, tools like CYPRESS or PLAYWRIGHT are introduced with DOCKER compatibility, GITHUB ACTIONS integration, or platform-aware CI scripts, demonstrating an intent to support scalable and maintainable test environments (Commits 3, 6, 9). Other adoption reasons are feature-driven, aiming to test UI behaviors that are difficult to verify with unit tests alone. For instance, some projects adopt web GUI tests to validate complex UI flows, such as markdown rendering or black-box systems like Fast Refresh, gaining visibility into critical behaviors (Commits 2, 4). Adoption also appears at the component level, with approaches such as the Screenplay Pattern, enabling modular and expressive tests aligned with modern front-end architectures. Finally, some commits represent an initial setup phase, introducing tools like PUPPETEER or SELENIUM alongside test harnesses and utilities, laying the foundation for broader test development over time (Commits 8, 10, 11).

Examining the migration pattern, we observe that the reasons often reflect the limitations or deprecation of existing frameworks. For instance, projects replaced SELENIUM with CYPRESS due to the deprecation of its methods (Commits 1, 5). At the same time, in other cases, persistent test flakiness led projects to adopt CYPRESS, applying refinements such as disabling test isolation and adjusting timeouts to improve stability (Commit 7). We also observed that the migration is driven by long-term alignment. One project migrated from PUPPETEER to PLAYWRIGHT to unify tooling across the codebase while the test suite was still small, minimizing disruption. Although the migration required only minimal changes, it also involved minor compatibility adjustments to ensure smooth execution (Commits 12, 13).

> **RQ$_1$.** While most projects were created after 2009, our analysis reveals a delayed adoption of browser automation frameworks, with an increase in usage after 2018, aligning with the release timeline of newer frameworks. Web applications primarily use CYPRESS and PLAYWRIGHT, while SELENIUM and PUPPETEER are less common. Most projects rely on a single framework, with CYPRESS + PLAYWRIGHT being the most frequent combination among projects using multiple frameworks. We also observed that projects tend to migrate toward newer frameworks, especially PLAYWRIGHT, reflecting a shift toward more modern tools. Finally, our qualitative analysis of commit messages revealed that adoption is driven by CI integration and complex UI testing needs, while migration is motivated by flakiness, deprecated APIs, or the desire for tooling unification.

4.2. *RQ$_2$: What are the main characteristics of projects adopting web GUI testing?*

Research Method. In the context of **RQ$_2$**, we analyze whether projects that adopt web GUI testing exhibit distinct software repository characteristics. Given the added complexity and maintenance costs of GUI tests, their adoption may depend on a project's size, activity, or popularity. Identifying potential correlations helps us understand which projects are more likely to integrate web GUI tests and whether specific repository attributes encourage their use. For these reasons, we focus on three different dimensions:

Project Maturity Metrics. This dimension offers insights into the overall development of the project. Specifically, we compute the number of Lines of Code (LOC), the project's age, and the number of tests, excluding web GUI tests. LOC serves as a measure of the project's scale, potentially correlating with the need for a broader testing suite, including GUI tests, to manage complexity. The age of the project reflects its longevity and development history, which may influence testing practices. Finally, the presence of non-GUI tests helps us understand the general emphasis on testing beyond GUI aspects, indicating whether these repositories focus more heavily on other testing types or complement them with web GUI testing. We also consider the programming language of the web frameworks used, i.e., PYTHON, JAVA, TYPESCRIPT, and JAVASCRIPT, as it can impact both the testing frameworks available and the likelihood of adopting web GUI testing.

Popularity Metrics. This dimension refers to the number of stars and watchers. These metrics are commonly used in software repository mining to estimate the general visibility or perceived popularity of a project within the developer community. However, it is important to note that these indicators are only proxies for popularity and do not directly measure it. They may not reflect actual usage of the software, as they can be influenced by factors such as novelty, reputation, or social endorsement. For example, GITHUB stars may signal interest or endorsement, but they do not imply active usage or maintenance, nor do they provide direct evidence of testing activity. To better capture development and maintenance-related dynamics, we complement these metrics with development activity indicators such as issue and commit activity, which offer stronger signals of ongoing upkeep, active involvement with the project, and responsiveness to bugs or feature requests.

Development Metrics. We refer to the number of distinct contributors, commits, and total issues per project. A higher number of contributors might lead to more diverse changes in the code, which can increase the likelihood of UI-related issues and, in turn, encourage web GUI testing adoption. Similarly, repositories with frequent commits are subject to regular updates, which require web GUI tests to ensure more stable interfaces. The total number of issues offers a view into both development activity and potential quality challenges; projects with a high issue count may employ web GUI tests to address user-facing bugs and enhance software quality.

The analysis is performed on the 5,563 non-trivial repositories previously collected [23], as detailed in Section 3. Specifically, our sample includes 472 repositories with at least one web GUI test and 5,091 repositories without web GUI tests. To compute the previously described metrics, we run SEART [15] and PYDRILLER [56]. Once we have computed the metrics, we first assess whether the metrics differ in the set of projects with web GUI tests and projects without web GUI tests. We inspect boxplots depicting the distribution of the metrics. Then, we apply the Kolmogorov-Smirnov test [32] to investigate whether the distributions of each characteristic for the two groups are normal. Consequently, to assess whether statistically significant differences exist between the distributions, we select the non-parametric Mann-Whitney U test [40], which does not assume normality. More in detail, for each of the considered project characteristics, we test the null hypothesis that there is no difference in the distribution of the characteristics between projects that adopt web GUI tests and projects that do not. Consistent with established practices in software engineering research (e.g., [3, 20]), the initial confidence level α for statistical tests is set to 0.05. To address the issue of multiple comparisons when testing a family of hypotheses, the final α is adjusted using the Bonferroni correction [9].

When statistically significant differences among the distributions are detected, we also quantify the effect size of observed differences using Cliff's delta [32]. We interpret Cliff's delta according to the guidelines presented in [61], which suggest that delta values of 0.11 or greater indicate a *small* effect, 0.28 or greater indicate a *medium* effect, and 0.43 or greater indicate a *large* effect.

After studying the statistical significance of the distributions, we assess whether the statistically significant metrics identified in the previous step are still significant when combining all metrics: this analysis is required since the individual effect of a factor might be reduced (or even lost) when other factors come into play [47]. Hence, we devise a *Logistic Regression Model*, which belongs to the class of *Generalized Linear Model* (GLM) [45]. We use this statistical modeling approach because it does not assume the distribution of data to be normal. To implement the model, we rely on the `glm` function available in the `R` toolkit.[4] Moreover, to avoid multi-collinearity, we perform a `VIF` (Variance Inflation Factors), using the `vif` function implemented in `R` toolkit. More in detail, we selected a standard VIF threshold value of 5 [46], and discarded all variable that exceeded such threshold.

Analysis of Results. The distribution of the considered characteristics across the two groups of projects is reported in Figure 4. The boxplots highlight several trends, suggesting that projects with web GUI tests have some differ-

[4] https://www.r-project.org/

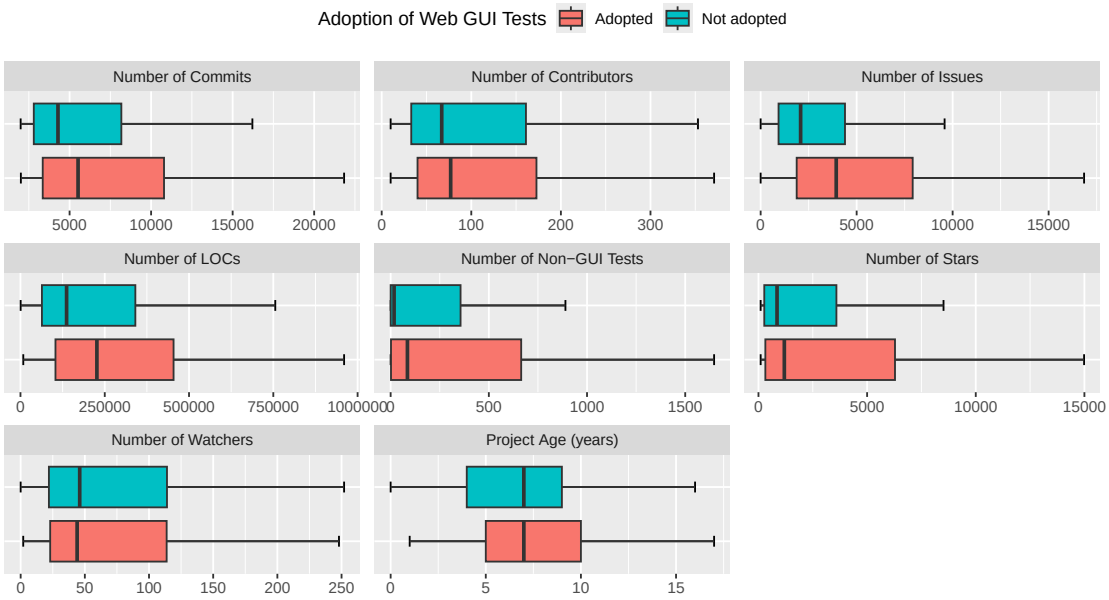

Figure 4: Distribution of Repository Characteristics for Projects with and without web GUI Tests. Outliers are hidden for ease of visualization.

ences from those without them in several characteristics. For instance, projects adopting web GUI tests showed a higher median number of commits, contributors, and lines of code, suggesting that these projects involve more collaborative and active development efforts. Similarly, projects adopting web GUI tests have more issue reports and a higher count of non-GUI tests, which could indicate a more comprehensive quality assurance process. Moreover, projects featuring web GUI tests exhibit a generally higher count of stars, which might imply that these projects may receive more community interest or are considered more significant by users. We do not observe a clear distinction between the two groups in terms of project age and number of watchers.

Table 3: Results of the Mann-Whitney U and Cliff's delta Statistical Tests. The *p-values* in **bold** refer to the metrics for which we accepted the alternative hypothesis, i.e., the distribution of the characteristics in the two groups differs in a statistically significant way.

	Mann-Whitney U Test	Effect size	
Characteristic	**p-value**	**Cliff's delta**	**Interpret.**
# of Commits	$\mathbf{1.68 \times 10^{-8}}$	0.15	Small
# of Contributors	**0.006**	0.07	Negligible
# of Issues	$\mathbf{2.03 \times 10^{-24}}$	0.28	Medium
# of LOCs	$\mathbf{6.41 \times 10^{-14}}$	0.20	Small
# of Non-GUI Tests	$\mathbf{6.7 \times 10^{-10}}$	0.16	Small
# of Stars	**0.0003**	0.09	Negligible
# of Watchers	0.8	–	–
Project Age (years)	$\mathbf{8.32 \times 10^{-8}}$	0.14	Small

Concerning the statistical tests and the computed effect sizes, the results are reported in Table 3. We highlighted in bold the *p-values* for which we accepted the alternative hypothesis that the distribution of the char-

Table 4: Overview of VIF repository characteristics analysis.

Feature	Value
TotalIssue	3.15
Commits	2.75
IsWebJs	1.74
IsWebTS	1.44
IsWebJava	1.40
Watchers	1.370
Contributors	1.35
Project Age	1.16
LOC	1.09

acteristics in the two groups differ in a statistically significant way. We can observe that statistically significant differences exist between the two groups for all attributes except the number of watchers. The Cliff's delta effect size indicated that, in most cases, the differences between the two groups are small, suggesting that their practical impact is generally limited. Specifically, the effect size is negligible when considering the number of stars and contributors, implying that the presence of web GUI testing is not strongly associated with greater popularity or community involvement.However, a medium effect size was observed for the number of issues, suggesting a more substantial difference in this dimension. This may indicate that repositories with web GUI tests tend to have more active issue tracking, possibly due to the added complexity introduced by maintaining automated tests or greater attention to quality assurance practices. Overall, while some distinctions exist, especially in issue-related activity, the limited effect sizes suggest that the adoption of web GUI testing does not strongly correlate with broader project metrics in most cases.

Before proceeding with the *Logistic Regression Model*, we performed the VIF analysis. Based on the results of this analysis, the number of stars metric was excluded because of the high multicollinearity (VIF > 5) with other variables. All the other variables can be considered for the Logistic Regression Model analysis because they exhibit acceptable multicollinearity levels (the maximum value is 3.15 for TotalIssues). The results of the VIF analysis on the selected variables are reported in Table 4.

Finally, Table 5 reports the results of the *Logistic Regression Model*. The result highlights that some characteristics could be used to predict the likelihood of a project adopting web GUI tests. Specifically, TotalIssues, IsWebJava, IsWebJs, and IsWebTs (i.e., the adoption of browser automation frameworks for JAVA, JAVASCRIPT or TYPESCRIPT) show positive associations with web GUI testing. Interestingly, other factors such as LOC, commits, and contributors do not show statistically significant associations. The result confirms that the adoption of web GUI testing is not necessarily driven by project size or general activity level. Rather, it appears to be more common in projects with high maintenance activity (as indicated by issue tracking), rather than in simply large or long-lived projects. This underscores that web GUI testing is not a default practice but could be adopted when maintenance demands

are high. Moreover, the lack of significance for certain variables does not contradict the trends observed in the earlier univariate analysis. Instead, it highlights how regression controls for the interdependence among predictors. For instance, the lack of significance for variables such as LOC and Contributors could be connected to their limited independent explanatory power when considered alongside more informative variables such as Total Issues or some technologies adopted in web applications, as observed in the `vif` analysis.

Table 5: For each variable, the tables report the value of the estimate, the standard error, and the statistical significance. The latter is explained by the number of stars, i.e., '***' indicates a p<0.001, '**' indicates a p<0.01, '*' indicates a p<0.05, and '.' indicates a p<0.1.

	Logistic Regression Coefficients			
	Estimate	**S.E.**	**Sig.**	**Rel.**
(Intercept)	-5.2	0.25	***	↘
LOC	-1.22	2.25		-
Contributors	-0.60	1.27		-
Commits	-12.45	3.65	***	↘
ProjectAge	3.05	0.33	***	↗
Watchers	-1.83	1.50		-
nOtherTest	2.02	3.05		-
TotalIssue	12.82	2.55	***	↗
IsWebJava	1.05	0.16	***	↗
IsWebJs	0.81	0.22	***	↗
IsWebTs	1.46	0.15	***	↗
IsWebPy	-0.89	0.46		-

The strong positive correlation between the Total Issues metric and the adoption of web GUI testing motivated us to qualitatively investigate these issues more closely.

1. We first collected all the open and closed issues from the repositories, for a total of approximately 3.3 million.

2. We then classified the issues based on their origin, differentiating between those automatically generated by CI/CD pipelines—specifically using GitHub Actions, Jenkins, Travis, and CircleCI—and those manually opened by developers. This classification was achieved by examining the username associated with each issue, which is typically standardized for automated processes, such as: *github-actions[bot]*.

3. To identify issues related to web GUI testing, we first compiled a list of all labels associated with the collected issues. We then manually selected 25 labels relevant to our study, such as *e2e*, *e2e-test-case*, *Scope: Test [E2E]*, *cypress*, and *run-e2e*. The full set of selected labels is made available in [22]. Using this label set, we extracted both open and closed issues related to web GUI testing, considering both automatically generated and manually reported ones.

4. Finally, we analyzed the resolution status of web GUI testing issues to determine whether they were actively addressed by developers or left unresolved due to inactivity. The goal of this classification was to distinguish

between issues that were opened due to a test failure caused by an actual bug in the application, which would likely lead to the issue being actively investigated and resolved, and those that stemmed from test fragility, flakiness, or slowness. The latter category often includes issues that are either ignored, quickly closed, or eventually closed due to inactivity, as they do not indicate a concrete defect in the application.

Table 6: Summary of open and closed issues, categorized by origin (auto-generated vs. manually created). For each category, the table reports the total number of issues and those related to web GUI Testing. Closed issues are further distinguished between those closed due to inactivity and those actively resolved by developers.

Closed				**Open**			
Auto-Generated		**Manually**		**Auto-Generated**		**Manually**	
374,647		2,771,809		4,016		186,593	
GUI Testing		**GUI Testing**		**GUI Testing**		**GUI Testing**	
600		6,647		57		357	
Inactivity	**Taken over**	**Inactivity**	**Taken over**	**Inactivity**	**Taken over**	**Inactivity**	**Taken over**
244	356	285	6,304	-	-	-	-

Table 6 presents an overview of the issues related to web GUI Testing, discerning between those automatically generated (AG) and manually created ones. Among the 3.3 million issues, 7661 were related to web GUI testing, indicating that discussions around such issues remain relatively limited. This suggests that while projects with a higher number of issues are more likely to adopt web GUI testing, its presence does not necessarily lead to a significant increase in reported issues. We attribute this result to the fact that projects adopting web GUI testing tend to be more mature, as previous analysis shows that web applications with web GUI tests generally have more established testing practices. This maturity could be translated into more organized and efficient testing workflows and, thus, more issues being opened and managed.

Manually reported problems dominate in both volume and resolution rate, suggesting low adoption of GitHub Actions and CI/CD tools. While 95% of manually created web GUI testing issues were actively resolved, only 59.3% of AG ones received attention. This suggests that web GUI test failures or discussions raised manually are taken more seriously than those flagged automatically. Furthermore, AG web GUI Testing issues are often ignored: we found that 40.7% of these issues were closed due to inactivity, indicating that CI/CD-detected web GUI test problems may not always be a priority.

Given that nearly half of the automatically generated web GUI testing issues were closed due to inactivity, it became crucial to understand the underlying reasons for this dismissal. One key hypothesis is that many of these issues stem from test flakiness or execution slowness rather than actual application defects. To investigate this further, we focused on a subset of web GUI testing problems specifically related to slowness and flakiness, identified by four labels: *flaky-e2e, slow-e2e, metric: stale flaky e2e test report, metric: flaky e2e test.*

Almost all (41 out of 57, corresponding to 72%) open AG issues connected to web GUI testing have as label flak-

iness or slowness, indicating that GUI test failures detected during CI/CD execution are predominantly classified as such. In contrast, only 9 (3%) of the open manually-created issues related to GUI testing were linked to flakiness or slowness. Concerning closed issues, out of the 244 AG issues that were closed due to inactivity, 85 (35%) were related to flakiness or slowness, while only 43 out of 356 (12%) taken over issues were related to the same topics. When considering manually created issues, we found that 151 out of the 285 (53%) issues closed due to inactivity were related to flakiness or slowness, as opposed to 693 out of 6,304 (11%) that were actively taken over.

These results reveal that a significant proportion of issues labeled as related to flakiness or slowness end up being closed due to inactivity, suggesting that developers tend to ignore them. This pattern is even more pronounced for automatically generated issues, where more than 72% of open issues are related to flakiness or slowness. Upon further investigation, we found that all 41 open AG issues originated from a single repository, i.e., WOOCOMMERCE. To provide further qualitative insight, we analyzed reports related to web GUI testing in the WOOCOMMERCE repository. Many of these issues were related to the introduction of web GUI testing to increase test coverage[5]. Additionally, many of these issues were closed due to inactivity rather than through active management[6], due to presenting flaky tests. This analysis lays the groundwork for future research that could further investigate issue reports and the testing practices adopted for web GUI testing, with a focus on how issues are handled and their impact on the development cycle.

> **RQ$_2$.** Our findings indicate that projects with web GUI tests tend to have more issues, commits, contributors, LOCs, and non-GUI tests. However, while these differences are statistically significant, the effect sizes are mostly small, except for the number of issues (medium). When metrics are combined, we observed that the number of total issues and the use of JAVA, JAVASCRIPT, and TYPESCRIPT are strong predictors of web GUI test adoption likelihood. Finally, the analysis on web GUI testing-related issues revealed that a significant portion of them is related to flakiness or slowness, and most issues in such portion are closed due to inactivity.

4.3. *RQ$_3$: Do web GUI tests co-evolve with the web application, and how are these tests maintained as the application evolves?*

Research Method. To answer **RQ$_3$**, we investigate the co-evolution of web GUI tests and production code through quantitative and qualitative analysis of commit histories, focusing on both technical (change frequency and coupling, nature of changes) and social (contributor roles) dimensions. Our research method consists of four steps.

1. *Quantitative insights on production and test code evolution dynamics.* Inspired by the previous work on the matter [11], we first provide quantitative insights into the rate at which web GUI tests change as the web application under test evolves. As such, we evaluate several metrics based on the categorization of commit activities shown in Table 7.

[5] E.g.: https://github.com/woocommerce/woocommerce/issues/52450
[6] E.g.: https://github.com/woocommerce/woocommerce/issues/52283

Table 7: Definitions of web GUI Test and Application Commit Metrics.

Metric	Description
Web GUI Test Commit (WGTC)	Refers to commits that add, modify, or delete at least one web GUI test file.
Application Commit (AC)	Refers to commits that add, modify, or delete web application code without touching web GUI test files.
Web GUI Test Span (WGTS)	Refers to the maximal sequence of successive WGTCs.
Application Span (AS)	Refers to the maximal sequence of successive ACs.
$\mathbf{WGTS_{C,\,D}}$	The length of a WGTS in days and commits.
$\mathbf{AS_{C,\,D}}$	The length of an AS in days and commits.

The commits considered in this analysis start from the introduction of the first web GUI test in each repository, allowing us to observe how these tests evolve alongside the application over time. Then, we complement the above commit-based metrics with an additional quantitative analysis aimed at investigating the maintenance of individual web GUI tests. This analysis aims to provide insights into how long web GUI tests are maintained before they are changed. We represent the lifespan of a web GUI test file as a sequence of similar blobs from successive commits, such that: (i) the first blob corresponds to a newly added file, (ii) the next blobs correspond to successive modifications of that file, i.e., they provide similar content, and (iii) the last blob corresponds to the last occurrence of the file. As for the definition of the similarity of blobs, we rely on the definition provided by Christophe et al. [11]. In that study, the authors used a similarity-based heuristic to track the evolution of test files over time, adopting a 66% line-based similarity threshold to determine whether two test artifacts across commits should be considered similar or distinct blobs. As specified in their work, the threshold was calibrated through manual inspection, as the commonly used 80% threshold, adopted in git to detect renamed files, was found to be too strict to capture real-world modifications typical of evolving web GUI tests [11]. To ensure methodological continuity with this foundational study, we adopted the same 66% threshold when identifying and linking evolving test scripts in our sample. These definitions give rise to the following maintenance metrics about a single web GUI test file, shown in Table 8.

Table 8: Definitions of web GUI Test Maintenance Metrics.

Metric	Description
$\mathbf{SV_{Day}}$	Number of days the web GUI test survived.
$\mathbf{SV_{Mod}}$	Number of modifications to the web GUI test.
$\mathbf{SV_{AC}}$	Number of application commits the test survived.
$\mathbf{SV_{WGTC}}$	Number of web GUI test commits the test survived.

2. *Analysis on the nature of changes made to web GUI tests over time.*

To better understand the nature of changes made to web GUI tests over time, we conducted a heuristic-based analysis focused on tests written using the SELENIUM framework. We selected SELENIUM due to its

consistent syntax, which enables easier parsing. For example, with SELENIUM, the use of the `By.<selector>` class makes the selector types explicit. We implemented a heuristic-based approach by defining regular expressions (regex) to detect two types of common maintenance activities: (i) the addition or deletion of assertions, and (ii) the addition or deletion of statements that leverage selectors, including `ID`, `name`, `tagName`, `className`, `cssSelector`, `xpath`, `linkText`, and `partialLinkText`. We applied the described heuristic to all changes involving a test file and reported statistics on the occurrence of these different types of modifications. Note that a single change may appear as both a deletion and an addition (e.g., when modifying a selector). This preliminary investigation covers 83 repositories, 11,751 commit messages, and 18,070 web GUI test modifications. It is worth noting that while we have identified a total of 87 repositories containing at least one SELENIUM-based test, four of them include only a test case that remained unchanged over time.

3. *Association rule analysis.* To statistically analyze the coupling between application changes and web GUI test changes, we define an association rule as also done in previous papers in this field [6, 29, 42]. Specifically, we examined whether a web GUI test is modified in the same commit as an application change or within a window of subsequent commits. While previous studies have shown that co-evolution between production and test code typically occurs within a 10-commit window [63], the dynamics of web GUI testing may differ from those of traditional test code evolution. As such, we conducted a systematic sensitivity analysis varying the commit window from 0 (i.e., co-evolution within the same commit) up to 10. This enables us to evaluate how coupling evolves over time and to ensure the robustness of the results across different temporal windows.

   We use the conventional metrics of *"Support"* (Supp), *"Confidence"* (Conf), and *"Interest"* (Lift) to measure the importance of an association rule [4]. $Supp(X)$ indicates the probability of the appearance of the event X, i.e., commits that modify the web application, while $Conf(X \rightarrow Y)$ indicates the probability that the event X will happen together ("is coupled") with the event Y, i.e., a change in the web GUI test code within the same commit or in the subsequent 10 commits. Moreover, $Lift(X \rightarrow Y)$ measures the degree to which the coupling between two file categories is different and independent from each other. A *Lift* value greater than one indicates that X and Y are more likely to occur together than by chance. In contrast, a value of one implies independence, and a value less than one suggests a negative association. The formula for *Lift* related to the *Conf* and *Supp* metrics are:

$$Supp(X \Rightarrow Y) = Supp(Y \Rightarrow X) = P(X \cap Y)$$

$$Conf(X \Rightarrow Y) = \frac{P(X \cap Y)}{P(X)} = \frac{Supp(X \Rightarrow Y)}{P(X)}$$

$$Lift(X \Rightarrow Y) = \frac{P(X \cap Y)}{P(X) \cdot P(Y)} = \frac{Conf(X \Rightarrow Y)}{P(Y)}$$

We use a χ^2 chi-squared test [4] to validate the statistical significance of the coupling between changes to X and Y. If the χ^2 statistic is greater than 3.84 ($\alpha = 0.05$) and the *Lift* value is greater than 1, we consider that X and Y have a statistically significant coupling. Otherwise, the observed relationship is attributed to chance.

$$\chi^2(X \Rightarrow Y) = n(Lift(X \Rightarrow Y) - 1)^2 \cdot \frac{Supp(X \Rightarrow Y) \cdot Conf(X \Rightarrow Y)}{(Conf(X \Rightarrow Y) - Supp(X \Rightarrow Y)) \cdot (Lift(X \Rightarrow Y) - Conf(X \Rightarrow Y))}$$

It is important to emphasize that the above-described metrics of *Support*, *Confidence*, and *Interest* aim at quantifying *how frequently and consistently* production and test code changes co-occur, not *why* they co-occur. Indeed, the fact that changes to test code frequently happen after changes to production code does not imply semantic coupling or causality, as test code might be updated for reasons (e.g., refactoring, flakiness fixes) unrelated to the prior changes in application code. Nonetheless, these metrics provide a scalable and objective proxy measure for co-evolution patterns in large-scale scenarios where finer-grained analyses are unfeasible, which may be further analyzed in future investigations.

Moreover, we note that our coupling analysis is based on changes at the application and web GUI test levels rather than a detailed class-level linkage. Unlike unit tests, which typically follow a one-to-one mapping between production and test classes, web GUI tests lack this association. As a consequence, analysis of the coupling between application-specific components and corresponding web GUI tests becomes challenging, limiting our ability to perform more granular investigations. This difficulty mirrors challenges observed in earlier work by Jiang and Adams [29] on the coupling between infrastructure files and application files, where file-level analysis posed similar challenges for detailed linkage between these types of files.

4. *Analysis of the social dynamics of web GUI test maintenance.* As last step of our analysis, we investigated who maintains web GUI tests throughout the evolution of each repository. Specifically, we leverage commit history data extracted using PyDriller [56], and we identified both contributors who made changes to web GUI test files and those who modified application code. Then, for each repository, we computed the intersection between these two sets to understand whether web GUI testing maintenance activities were typically performed by the same developers who developed the application.

Analysis of Results. Table 9 provides a summary of metrics related to the commit activity. We can observe that the average number of Application Commits (AC) is 1,182.91, while the average number of web GUI test commits (WGTC) is 88.82. The wide variation in AC counts (standard deviation of 3,263.88) suggests that while most projects have moderate commit counts, some include an exceptionally high number of application changes. This variability is further confirmed by the registered maximum value, which is 35,641 ACs. In contrast, WGTCs are more consistently distributed, though still showing a large range, with a maximum of 1,647 commits.

When examining the consecutive sequences of either web GUI tests or application commits, the average span is 280.25 for the application and only 10.60 for the web GUI tests. The high standard deviation in the application

Table 9: Summary statistics for co-evolution metrics.

Statistic	#WGTC	#AC	#WGTS	#AS	$WGTS_D$	$WGTS_C$	AS_C	AS_D
Mean	88.82	1,182.91	10.60	280.25	1.35	2.43	5.18	2.84
Std Deviation	174.42	3,263.88	27.10	480.2	6.14	0.56	3.61	5.52
Median	28.5	714.5	2.00	123.00	1.00	2.28	4.36	1.58
1st Quartile	9.00	252.25	0.00	46.00	1.00	2.00	3.17	1.17
3rd Quartile	84.5	1920.75	8.00	336.00	1.10	2.63	5.91	2.80
Min	1.00	1.00	0.00	3.00	1.00	2.00	2.00	1.00
Max	1,647	35,641	280	5,910	104	7	52.28	72

Table 10: Summary statistics for maintenance metrics.

Statistic	SV_{Day}	SV_{Mod}	SV_{AC}	SV_{WGTC}
Mean	673.11	5.81	1,182.08	133.08
Std Deviation	855.11	9.04	2,208.3	230.90
Median	372.00	3.00	446.5	39.00
1st Quartile	91	1.00	93.00	8.00
3rd Quartile	840.5	7.00	1,340.25	150.00
Min	1.00	0.00	1.00	1.00
Max	5,393	330	35,571	1,643

span counts (480.2) also points to variability across projects. In terms of duration, both spans for application and web GUI tests remain relatively short-lived, with median values of 1.58 days for application changes and 1 day for web GUI tests. This indicates that updates are typically addressed within a brief timeframe. Regarding the number of commit sequences involved, application spans average 5.18 commits, while web GUI test spans average 2.43, suggesting that although application updates may require more extensive changes, web GUI test modifications are often more concise.

Summary statistics for the maintenance metrics are reported in Table 10, and offer insights into the maintenance characteristics of web GUI tests over time. We can notice that web GUI tests demonstrate substantial longevity, surviving an average of 673.11 days with a median of 372 days. However, the high standard deviation (855.11) indicates considerable variation in test lifespan, with some tests lasting significantly longer than others. The frequency of web GUI test modifications is low, with a mean of 5.81 modifications and a median of 3, suggesting that most tests are likely to remain stable and require few updates. Additionally, web GUI tests tend to persist through a considerable number of application commits, with an average lifespan of approximately 1,182 commits, although this value varies widely as reflected by a high standard deviation (2,208). While web GUI tests experience fewer changes than application code, they persist through a moderate number of these updates, with an average of 133.08 commits and a median of 39. These results suggest that practitioners could perform an initial burst of activity in creating web GUI test files, followed by little or no long-term maintenance. This could be attributed to several factors, such as increasing maintenance costs, the erosion of developer trust due to flaky or

Table 11: Simplified overview of average GUI testing maintenance activities.

Commits updating GUI Test Code	Commits updating App Code	Assertions		Selectors	
		ADD	DEL	ADD	DEL
141.57	4339.04	272.19	194.04	41.19	63.74

Table 12: Average number of selector changes per selector type in GUI testing.

ID		Name		XPath		CSS		Class Name		Tag Name		Link Text		Partial Link Text	
ADD	DEL	ADD	DEL	ADD	DEL	ADD	DEL	ADD	DEL	ADD	DEL	ADD	DEL	ADD	DEL
6.42	15.32	0.55	0.72	10.67	13.30	9.87	15.10	6.60	11.38	4.92	5.31	2.57	2.06	0.55	0.53

brittle tests [44], or a strategic shift in validation efforts toward other testing levels (e.g., API or unit testing) [25]. The relatively low number of modifications, despite long test lifespans, may indicate that once established, web GUI tests are either relatively stable or neglected, highlighting the need for further investigation into test sustainability and developer engagement over time.

Concerning the second step of our analysis, Table 11 summarizes the average number of web GUI test maintenance activities observed across repositories. On average, each repository contains approximately 142 commits involving web GUI test files, compared to 4,339 commits related to the application code, highlighting the relatively low number of commits related to modifying web GUI tests. Among the different types of maintenance actions, assertion updates are the most frequent, with repositories showing an average of 194 deleted and 272 added assertions. Selector modifications are less common, with an average of 63 deletions and 41 additions per repository, indicating that locating and interacting with elements remains a maintenance concern in GUI testing.

Table 12 presents a detailed breakdown of the average number of changes across different selector types in GUI testing. By examining these results, we observe distinct patterns in how selectors are maintained.

The most frequently changed selectors are ID, CSSSELECTOR, and CLASSNAME, both in terms of deletions and additions, indicating that these selectors play a crucial role in web GUI test maintenance activity. Specifically, ID selectors exhibit an average of about 15 deletions and 6 additions, CSSSELECTOR changes occur around 15 deletions and 10 additions, and CLASSNAME sees roughly 11 deletions and 7 additions. Other selector types, such as XPATH, also show notable activity, with 13 deletions and 11 additions on average, reflecting their importance in test adaptation. Less frequently changed selectors include NAME, TAGNAME, LINKTEXT, and PARTIALLINKTEXT.

Interestingly, despite being traditionally considered one of the most robust and stable selector types, the ID selector is the most frequently deleted. This could be explained by the increasing complexity of modern HTML structures and the dynamic nature of many web applications, where elements with the ID attribute often change between page loads or iterations. As a result, testers might replace ID selectors with more flexible or stable alternatives, such as XPATH or CSSSELECTOR, which show relatively high rates of addition in our data. This swapping behavior suggests an adaptation strategy to maintain test resilience in the face of dynamic DOMs. Overall, these

findings highlight that web GUI test maintenance is an ongoing process that involves frequent assertion updates and strategic modifications of selectors to adapt to evolving UI architectures. Understanding these patterns can inform the development of more robust test design and maintenance tools. Additionally, this exploration offers early insights into the most frequent changes and lays the foundation for further studies.

As for the association rule investigation, Table 13 reports the number of repositories that exhibit statistically significant coupling across the different commit window sizes. We observe that no coupling was found at window size 0, confirming that web GUI test changes rarely occur in the same commit as application changes. As the window size increases, the proportion of repositories showing coupling grows rapidly, especially within the first commit, where we found that 51% of the projects co-evolve. A higher increase can be observed in the 2-commit window, where 63% of the projects show coupling between web GUI test and application code. The growth continues steadily up to a 3- to 4-commit window, where the percentage of coupled projects reaches 66%. Beyond this point, the increase becomes marginal, with the rate remaining stable around 68–69% starting from the 6th commit window onward. To further confirm this stability of the percentage, we also verified the number of projects captured with 15- and 20-commit windows. We observed that in both configurations, the rate is always 69%, confirming the saturation of meaningful co-evolution signals beyond the 5-commit range. These results suggest that most of the co-evolution activity between application code and web GUI tests occurs within a very short temporal span after the application code is changed. As such, in the following we analyzed three more relevant commit windows. Specifically, we focused on the commit window of size one, where we observed the higher coupling,then we moved to the 5-commit window, where we captured the vast majority of co-evolution cases, and finally we included the 10-commit window for completeness and to remain consistent with prior literature.

Figure 5 presents violin plots for the three association rule metrics, *Support*, *Confidence*, and *Lift*, measured at commit window sizes of 1, 5, and 10, chosen based on our sensitivity analysis that showed co-evolution signals stabilize and most meaningful interactions occur within this range. This comparative view illustrates how the temporal distance between application and GUI test changes influences co-evolution behavior. Detailed results for each project are available in the appendix [22].

Starting from the *Support*, The average support is 0.086, meaning that only 8.6% of app-code changes are followed by a GUI test change in the same commit. The confidence is similarly low (8.9%), and the average lift is 1.27, suggesting only a slight positive association between the two events. The median values (*Support*: 0.053, *Confidence*: 0.053, *Lift*: 1.30) and the shape of the violin plots indicate that in most projects, the likelihood of co-evolution within the same commit is limited but not negligible.

As the window expands to five, the average support and confidence nearly double (*Support*: 0.162, *Confidence*: 0.168), while the *Lift* increases significantly to 2.67 on average. This indicates a stronger association between app-code and test changes within the five subsequent commits. The median values follow the same trend, showing that in at least half of the projects, 10.7% of app changes are followed by GUI test changes within five commits.

With the 10-commit window, the pattern becomes even more pronounced. The average *Support* reaches 0.227,

Table 13: Number of repositories showing statistically significant (Lift >1 and 1χ^2 >3.84) coupling between application code changes and GUI test modifications, valuated across different commit window sizes.

Commit Window Size	#Repositories (%) with statistically significant coupling
0	0 (0%)
1	244 (51%)
2	297 (63%)
3	314 (66%)
4	314 (66%)
5	319 (67%)
6	321 (68%)
7	321 (68%)
8	323 (68%)
9	324 (69%)
10	324 (69%)
15	325 (69%)
20	325 (69%)

Table 14: Summary statistics of the number of contributors involved in maintaining production code and web GUI test code across the studied repositories.

Statistic	Contributors maintaining Production Code	Contributors maintaining Web GUI Test Code
Mean	211.37	7.56
Median	104	4
Q1	55.50	2
Q3	233.50	8
Min	11	1
Max	1255	153

and *Confidence* 0.235, indicating that nearly 1 in 4 application code changes are eventually followed by a GUI test modification within the next 10 commits. The average *Lift* increases to 4.05, suggesting a strong positive association. The violin plots reveal a concentration of values around the median *Lift* of 4.05, confirming that this association is consistently observed across many projects rather than being skewed by a few outliers. While these results cannot rule out incidental co-changes entirely, they suggest that the observed patterns are unlikely to be observed by chance alone. However, further validation would be required to confirm intentional co-evolution.

Finally, Table 14 presents the results of our analysis of contributor roles across the studied repositories. We observe that the median number of overall contributors per project is 104, indicating generally large and active development teams. Interestingly, the number of developers who actively maintain web GUI tests is particularly low (the median is 4). While further investigations are necessary, this result suggests that the maintenance ac-

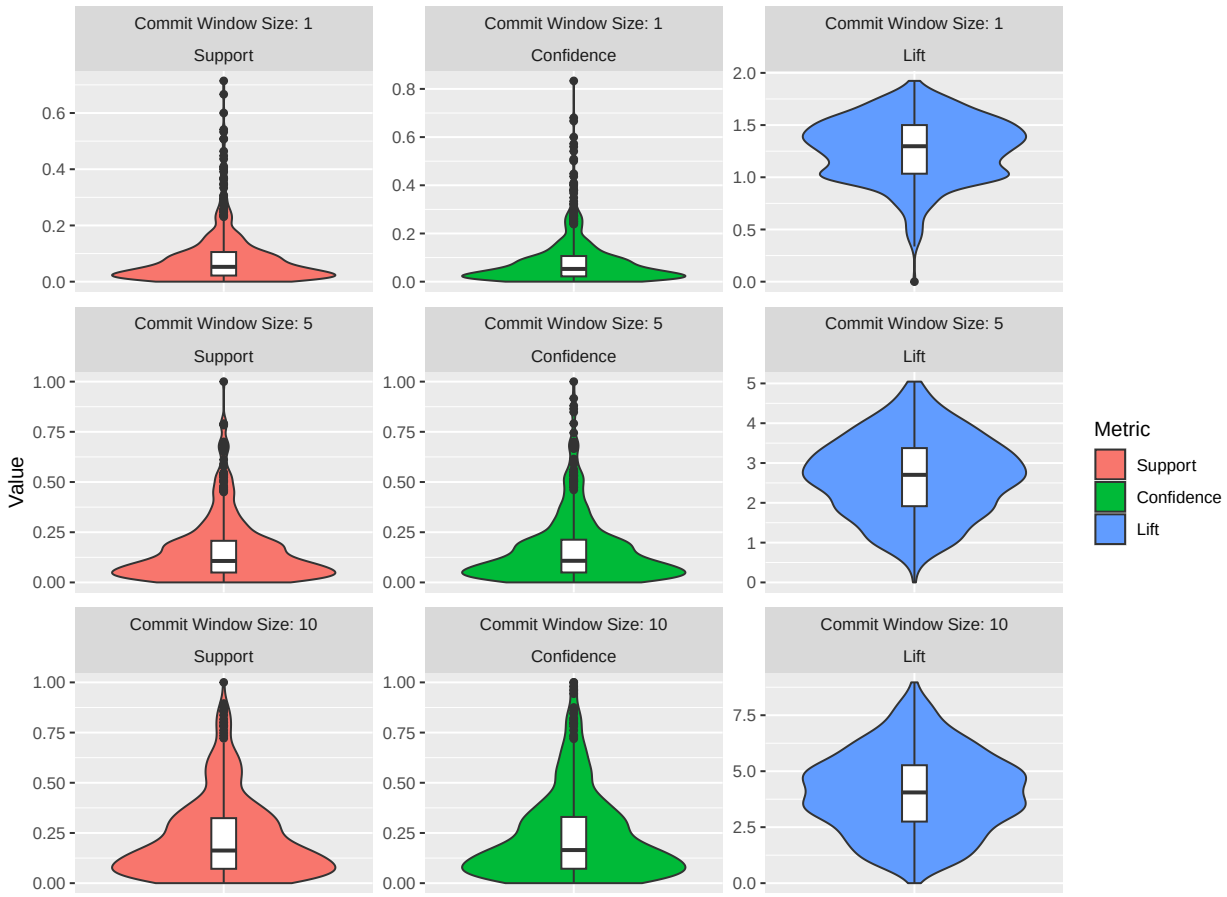

Figure 5: Violin plots showing the distribution of the *Support*, *Confidence*, and *Lift* metrics across different commit window sizes.

tivities of web GUI tests are under the responsibility of a small subset of developers, likely due to the specialized knowledge required to work with web automation frameworks and test selectors. Additionally, the intersection analysis reveals that all developers contributing to web GUI testing activities are also involved in developing production code.

> **RQ$_3$.** Our analysis highlights a variation in application and web GUI test commit activity across projects, with web GUI tests showing more consistency. Application changes and web GUI test spans are typically short, reflecting quick handling. Web GUI tests show durability, requiring few modifications over an average of 673 days. Looking at the association rule, the *Support*, *Confidence*, and *Lift* metrics suggest that application and web GUI test changes are statistically associated. Additionally, we also observe that assertion changes are the most frequent type of test maintenance in SELENIUM, while changes to selectors, especially `ID`, `cssSelector`, and `Class Name`, highlight adaptation strategies to maintain test stability. Moreover, we observe that test maintenance is handled by a small group of developers who also contribute to production code, suggesting specialized but integrated responsibilities.

5. Discussion and Practical Implications

The results provide several observations, reflections, and implications for research and practice. In this section, we elaborate on the main insights coming from our study.

5.1. Implications for Practitioners

Our findings provide several actionable insights for practitioners aiming to enhance their web GUI testing practices and address common challenges in real-world projects.

RQ$_1$ confirms a preference for modern browser automation frameworks, such as CYPRESS and PLAYWRIGHT. We have also observed a possible trend towards incorporating automated testing earlier in the development lifecycle. These findings align better with contemporary CI/CD pipelines and modern development practices. However, tools like SELENIUM remain relevant due to their cross-browser compatibility and extensive community support. Practitioners should evaluate their project needs, balancing modern framework capabilities (e.g., ease of integration, faster feedback cycles) with the broader compatibility and stability offered by mature frameworks.

In addition to the observed trends in tool preference, the qualitative analysis performed to understand the reasons behind adoption and migration led us to identify specific reasons behind the web GUI testing adoption and migration, such as strengthening CI pipelines and flakiness resolution. Practical recommendations for practitioners can be drawn from these insights, such as prioritizing the choice of web GUI testing tools that offer strong CI/CD integration (e.g., compatibility with DOCKER or GITHUB ACTIONS) and are robust against timing-related instability. In addition, web GUI tests are often adopted to validate complex UI behaviors that cannot be easily tested at the unit level. This suggests that developers should strategically introduce web GUI testing when implementing non-trivial UI flows to ensure end-user reliability.

Implication for Practitioners. Practitioners should choose GUI testing tools with strong CI/CD integration capabilities and flakiness resilience. Adoption and migration to other testing tools is often driven by the need to support automated pipelines and reduce instability in test execution.

RQ$_2$ shows that repositories with greater community engagement, as indicated by metrics like contributors, are more likely to adopt web GUI testing. While the observed effect sizes are generally small, this trend may indicate that collaborative and actively maintained projects are more inclined to invest in testing practices, possibly to manage increased complexity or to meet higher quality standards. Although these findings do not imply strong causality, they can inform practitioners and maintainers by highlighting conditions under which web GUI testing is more frequently adopted. Furthermore, in **RQ**$_2$, we found that a portion of automatically generated issues related to web GUI tests were closed due to inactivity, potentially indicating flaky tests. Practitioners should establish robust strategies for managing automated issue reports, such as prioritizing the resolution of flaky tests and implementing automated retries for transient failures. This minimizes disruptions in CI/CD pipelines and increases developer trust in automated testing.

Implication for Practitioners. Practitioners should fix flaky tests proactively. Automatically flagged flaky tests are often ignored, hurting pipeline trust over time and potentially resulting in test suites abandonment.

RQ$_3$ reveals that web GUI tests generally co-evolve with applications, requiring relatively few modifications over time. This challenges the common belief that GUI tests are inherently fragile, showing instead that they can be highly durable when properly designed and maintained. The historical analysis of test file evolution suggests that frequent, incremental updates lead to lower long-term costs compared to infrequent, large-scale overhauls. To maximize the benefits of web GUI testing, practitioners should align maintenance practices with application development workflows, ensuring that tests evolve alongside code changes. Leveraging version control data to detect unstable test files and aligning test updates with small, continuous application changes may help reduce long-term maintenance overhead and mitigate the risks associated with flaky or brittle tests.

Implication for Practitioners. Practitioners should update GUI tests incrementally. GUI tests co-evolve with application code and benefit from small, frequent updates.

Our findings also show that several web applications exhibit an initial burst of web GUI test activity followed by a decline. This pattern could be explained by the costs associated with maintaining fragile selectors, the erosion of trust caused by test flakiness, or a strategic decision to prioritize other levels of testing, such as unit or API testing, which is perceived as less costly to maintain. Recognizing these conditions could enable practitioners to anticipate when maintenance may decay and proactively adopt mitigation strategies, such as resilient locator design and a balanced allocation of testing effort across different levels.

Implication for Practitioners. Web GUI tests are often abandoned after initial adoption. Practitioners should be cautious of maintenance decay and plan for long-term upkeep to avoid test suite erosion.

Finally, in the analysis of developer contributions to web GUI test maintenance activities, we have observed that, while the median number of contributors to production code is 104, only a small subset (median of 4) actively maintain web GUI tests. Moreover, all test contributors were also involved in application code development, suggesting that dedicated testing specialists do not handle web GUI testing but rather core developers. This analysis indicates potential bottlenecks: test maintenance activities could be concentrated in the hands of a few developers with tailored knowledge of automation frameworks and test selectors. To overcome this potential limitation, teams should promote shared ownership of test assets through cross-training, improved documentation, and tighter integration of web GUI testing into the CI/CD workflow.

Implication for Practitioners. Web GUI test upkeep is often handled by small subsets of developers; Practitioners should promote shared ownership and cross-training to avoid potential maintenance bottlenecks.

5.2. Implications for Researchers

Our results offer several insights for further investigation. First, while this study focused on GitHub repositories using popular browser automation frameworks, extending research to additional frameworks and languages

could provide more comprehensive insights into web GUI testing practices. Such collections would be valuable for analyzing broader adoption trends and could contribute to establishing best practices that could be relevant to diverse industry contexts.

The qualitative analysis, performed in $\mathbf{RQ}_1$ on commit messages to understand the reasons behind the adoption and migration of web GUI testing, showed the difficulty in extracting a clear rationale from commit messages (only 13 out of 1,069 messages provided clear motivation). While this result aligns with previous studies [37, 59], it suggests that traditional mining of commit history is not enough to understand the reasoning behind testing practices. This highlights the need for complementary qualitative methods (e.g., interviews or surveys), as well as research into improved traceability between testing activities and their underlying motivations.

Implication for Researchers. Commit messages and issue reports rarely explain GUI test adoption/migration between frameworks. Alternative approaches, such as qualitative methods with interviews or surveys, might be needed to better understand developer intent.

The findings of $\mathbf{RQ}_2$ suggest a potential link between project maturity and the adoption of web GUI testing, with more mature projects potentially favoring this practice. This observation aligns with prior research indicating that larger, more mature projects may implement more comprehensive testing strategies [65]. Moreover, our qualitative analysis suggests that while projects with a higher total issue count are more likely to adopt web GUI testing, the nature of the issues varies significantly. Manually reported issues are actively resolved by developers, whereas automatically generated issues, often referring to test flakiness or execution slowness, are more frequently dismissed or closed due to inactivity. This suggests that projects not only invest in advanced testing practices but also develop structured approaches to triage and manage test-related issues, prioritizing those that reflect actual application defects over transient test inconsistencies. Future studies could explore how these aspects of issue management influence both the adoption and long-term maintenance of web GUI tests.

Implication for Researchers. Manually reported GUI test issues are generally actively resolved, while automatically generated ones are often ignored. Researchers should investigate how issue origin affects developer response and what this implies for the design of effective test reporting and triage systems.

The findings of $\mathbf{RQ}_3$ reveal that web GUI tests themselves undergo frequent changes, indicating an evolution process similar to that of the application code. A more granular inspection of these changes in projects using SELENIUM, showed that updates often involve modifications to element selectors, especially CSS and XPATH selectors based on attributes such as ID and CLASS NAME. These types of selectors are inherently brittle, as they are tightly coupled with the underlying HTML structure and class naming conventions. This suggests that at least part of the observed test evolution is driven by shifts in the front-end implementation rather than changes to the test logic itself. The observed nature of web GUI tests' evolution motivates future research into the specific types of changes, as well as the factors driving these modifications between structural fragility and legitimate test evolution. This also motivates future research into the specific types of changes, the root causes of selector insta-

bility, and strategies for selector robustness. Additionally, prior work has used *"work item"* aggregation, such as grouping commits within pull requests, to capture a more cohesive view of development activities and their interdependencies [6]. Incorporating this aggregation approach could further enhance our understanding of coupling by grouping related changes into single units, aligning with how tasks are managed in collaborative environments.

At the same time, the initial burst of web GUI testing activity followed by a rapid decline observed in $\mathbf{RQ}_3$ raises important questions about the underlying causes of such decay. Investigating the factors behind this trend, such as unresolved flaky tests or a shift to other testing levels, would provide a deeper understanding of why web GUI testing practices decline over time. Such studies could inform the design of tools and processes aimed at automatically detecting and mitigating flaky tests, as well as clarifying the trade-offs between web GUI testing and alternative testing strategies.

Implication for Researchers. Decay in maintenance activities for web GUI tests warrants deeper study. The observed drop-off in GUI test activity calls for an investigation into possible causes like flakiness and testing strategy shifts.

As for the association rule performed in $\mathbf{RQ}_3$, while confidence served as the primary metric for analyzing coupling, alternative metrics such as shared dependencies, semantic similarities in change logs, or time-lagged correlations could provide additional layers of insight. Researchers applying these metrics may be able to construct predictive models to identify when web GUI test updates are likely necessary. Finally, the low number of test maintainers of web GUI tests raises questions about the scalability and sustainability of current maintenance practices. This reveals several directions for future work, such as investigating socio-technical patterns in test ownership, barriers to broader contribution to testing, and designing tools and techniques to lower the barrier to entry for web GUI test maintenance. In this regard, our results provide an empirical basis for further qualitative studies, such as developer interviews or surveys, to explore how maintenance responsibility is distributed and how tool support can evolve to facilitate more inclusive participation.

6. Threats to Validity

Multiple factors might have biased the conclusions drawn in our study. This section overviews the main limitations faced and how we mitigated them, discussing them based on their impact on our research.

Construct Validity. The main threat related to the relationship between theory and observation concerns possible imprecision in the data used in the study. Our study focused on GitHub web applications that use popular browser automation frameworks for web GUI testing activities at the time of collection. Although less common frameworks may have been excluded, we believe that our selection accurately represents the current state of web GUI testing practices. We also recognize that our context selection may have excluded projects that previously employed web GUI testing but no longer do so. However, our goal was to analyze projects that actively integrate these techniques, thereby offering a precise snapshot of contemporary practices (the data reported in [23] were

collected in May 2024). A comprehensive historical analysis of all commits in non-trivial web applications is beyond the scope of this study.

As for the metrics computed in **RQ**$_2$, we relied on automatic tools to extract project characteristics related to maturity, popularity, and development activity. We acknowledge the potential noise that may arise from automated extraction, such as inaccuracies in LOC calculations, incomplete metadata, or limitations in detecting development contributions. To partially mitigate this threat, we employed well-established tools that have been previously evaluated, demonstrating good accuracy in repository mining and software evolution analysis [15, 56].

Moreover, we acknowledge that the project-level metrics used in our study, i.e., GITHUB stars, watchers, issue counts, and pull request activity, are proxies for broader constructs like adoption, popularity, and maintenance effort. While these metrics are commonly used in software repository mining, they do not directly measure the constructs they are intended to represent. For instance, GITHUB stars may reflect general interest or visibility rather than actual adoption of a software. Similarly, issue and PR activity may capture aspects of development and maintenance, but not all maintenance work is formally tracked through these mechanisms, and some issues may be unrelated to testing. These limitations represent threats to construct validity, as they concern the degree to which our metrics accurately reflect the theoretical concepts under investigation. To mitigate these threats, we interpreted our findings with caution, explicitly acknowledging the proxy nature of these metrics.

Related to the choice of the commit window, we performed a sensitivity analysis to evaluate the impact of varying this window size on the observed co-evolution patterns. This analysis covered a range of different commit sizes, from 0 to 10, and showed that the majority of co-evolution signals are captured within a 5-commit window, with results stabilizing beyond that point. Additionally, to address the potential limitation of delayed updates, where test changes are committed long after the corresponding application modifications, we extended the analysis using larger commit windows (15 and 20 commits). This allowed us to capture co-evolution practices that might not be visible in shorter time spans. In both configurations, the percentage of projects remained stable at 69%, confirming that meaningful co-evolution signals tend to saturate beyond the 5-commit range.

As for the analysis on the types of changes affecting web GUI tests, we introduced a static, heuristic-based analysis focusing on selector and assertion modifications in SELENIUM test scripts. While this approach provides a first step toward understanding the nature of test maintenance, it is inherently limited in scope and precision. First, it only captures changes that match specific syntactic patterns, such as changes involving `By.<selector>`. This could potentially ignore semantic or structural modifications that do not fit the defined regex criteria. Second, the heuristic is specific to SELENIUM, which limits its applicability to other frameworks such as CYPRESS or PLAYWRIGHT, which use different and more flexible syntactic constructs. Although our preliminary results provide valuable insights into common maintenance actions, a more comprehensive classification of change types would require a finer-grained analysis, which could involve AST differentiation or manual inspection. In addition, a dynamic analysis of changes could further strengthen the results by revealing how changes in the application

code may affect test behavior at runtime. However, as noted in prior studies [39, 60], performing dynamic analysis across historical versions of real-world projects is extremely challenging due to the low success rate in compiling past snapshots and the high cost of environment reconstruction.

External Validity. The main threat concerning the generalizability of the results is the project selection. Our study focused on open-source projects selected from GITHUB, which represent only a fraction of the broader open-source ecosystem. While GITHUB is widely used and includes projects from individual developers, academic groups, and some industry teams, it may not fully represent the practices of proprietary or enterprise software development. In particular, closed-source projects may follow different workflows, use alternative tooling, or have distinct team structures and testing strategies that are not observable in our dataset. As such, the findings presented in this study primarily reflect open-source development contexts and may not generalize to all industrial settings. Future work could complement this analysis with data from enterprise environments or industry collaborations to validate and extend the insights presented here. In this regard, we released all materials publicly available to stimulate further research that may corroborate our findings in different contexts [22].

In our study, we focused on scripted testing based on popular browser automation frameworks, such as SELENIUM, CYPRESS, PLAYWRIGHT, and PUPPETEER. This criterion provides a concrete and reproducible definition, aligned with practices commonly adopted in both industry and academia. However, we acknowledge that this definition may exclude alternative forms of web GUI testing implemented through different approaches, e.g., low-code solution. As such, our results may not generalize to all forms of web GUI testing, particularly those relying on low-code or no-code solutions, or custom testing utilities that do not use dedicated browser automation frameworks. These approaches, while potentially widespread in certain contexts, are harder to identify systematically through automated mining due to the lack of standardized APIs or naming conventions. Nevertheless, we see this as a necessary trade-off to enable large-scale analysis while maintaining clarity and reproducibility. Future work could expand on this foundation by investigating the evolution and maintenance of different testing paradigms.

The findings in $\mathbf{RQ}_2$ have shown that development metrics are statistically correlated to the presence of web GUI testing, even with a small effect size. However, projects following test-driven development (TDD) practices may exhibit different test evolution patterns compared to those that perform testing only in later development stages. Since our study does not explicitly classify projects based on development methodologies, our findings may not generalize to all software engineering contexts. Future research could investigate testing adoption within specific development paradigms to understand better how different workflows influence web GUI testing evolution.

Conclusion Validity. Threats to conclusion validity are related to the relationship between treatment and outcome. As for the statistical methods employed in $\mathbf{RQ}_2$, while the statistical results revealed differences between projects with and without web GUI tests across several characteristics, we acknowledge that correlation does not imply causation. The observed associations indicate patterns of co-occurrence, but they do not confirm that any

specific project characteristic directly causes or influences the adoption of web GUI testing. For example, we have observed that projects with a greater number of issues may tend to adopt web GUI testing. Still, the presence of a greater number of issues may be a consequence of the greater project complexity or community engagement rather than a causal factor. To partially mitigate possible overinterpretation, we complemented the analysis on the single factors with a multivariate Linear Regression Model, allowing us to assess the independent contribution of each variable while controlling for confounding factors. We selected the Linear Regression Model after verifying its suitability for our purpose. In addition, to ensure that the model did not suffer from multi-collinearity, we applied the `vif` function, aimed at discarding non-relevant independent variables. These procedures followed established guidelines [46], making us confident of the validity of the process employed. However, establishing causality between the characteristics analyzed and the adoption of web GUI testing would require further qualitative research, such as surveys or interviews with practitioners. Nonetheless, conducting such an analysis is beyond the scope of this paper, as designing a valid and unbiased survey would require considerable methodological work. As part of our future agenda, we plan to design and carry out such a study with developers and maintainers to understand better the motivations and factors influencing the adoption of web GUI testing in practice.

Concerning $\mathbf{RQ}_3$, we used the confidence of association rules to measure the coupling between application and web GUI test changes. Association rules, however, can identify coordinated change patterns but cannot establish causation links between production and test code modifications. This approach, which was also adopted in prior work investigating the co-evolution of software artifacts [6, 29, 42], may not capture all relevant co-evolution patterns (e.g., it may miss those occurring outside the considered time window), and may falsely interpret as related coincidental code changes that happen in the same window despite not being strongly related. We acknowledge that commit-based association rules are a scalable proxy for co-evolution, able to detect potential candidate co-evolution patterns for further investigation, and that deeper validation (e.g., developer studies, coverage analysis with static or dynamic traces, or manual code inspection) would be required to assess proper causal relations.

7. Conclusion

The ultimate goal of our work was to analyze the adoption and maintenance of web GUI testing in open-source projects, focusing on 472 web applications that employed popular browser automation frameworks such as SELENIUM, CYPRESS, PLAYWRIGHT, and PUPPETEER. Specifically, we analyzed the prevalence of web GUI testing, the characteristics of projects adopting it, and the co-evolution of tests with the application files they support.

Our findings show an increasing adoption of web GUI testing, particularly driven by the rise of modern frameworks like CYPRESS and PLAYWRIGHT, which have gained significant popularity in recent years. We also observed that repositories adopting web GUI testing often exhibit higher levels of community engagement, indicating that projects with automated GUI testing may attract more contributors, issue discussions, and long-term maintenance efforts. While statistical tests confirmed significant differences across several dimensions, the effect sizes

were generally small, except for the number of issues. Regarding the co-evolution of web GUI tests and application code, our findings suggest that test modifications are generally well-aligned with application changes, requiring relatively few updates over time. This contradicts the common belief that web GUI tests are inherently fragile.

Our study provides valuable insights for both researchers and practitioners. The increasing adoption of web GUI testing highlights its growing importance in modern software quality assurance, particularly as newer frameworks lower the barriers to adoption. At the same time, the observed co-evolution trends emphasize the need for better test maintenance strategies, supporting the idea that well-structured web GUI tests can remain stable over time. These results suggest that improving test design, automated test maintenance, and framework migration support could further enhance the effectiveness of web GUI testing.

As part of our future agenda, we plan to further investigate co-evolution patterns through developer studies and hybrid trace-based analyses, combining our large-scale commit-based association rule analysis with finer-grained techniques like dynamic execution traces and coverage analysis. This multi-method approach would help validate whether the temporal couplings we observed reflect intentional test maintenance practices rather than coincidental changes, and additionally could provide important insights into the actual reasons that drive web GUI test evolution. Additionally, we plan to conduct qualitative studies with practitioners, such as interviews or surveys, to understand the rationale behind the adoption and migration of web GUI testing frameworks. This will enable us to contextualize our findings and identify socio-technical factors that could influence test practices. Moreover, we recognize that further research is needed to understand why web GUI testing practices may decline over time. Such studies could explore whether developers shift validation efforts toward alternative testing strategies (e.g., API or unit testing), or whether challenges such as test flakiness and high maintenance costs contribute to test abandonment. These insights could inform the design of tools and processes aimed at automatically detecting and mitigating flaky tests, improving test resilience, and supporting more sustainable GUI testing practices in both open-source and industrial contexts.

Our findings also highlight the need for tool support to facilitate web GUI test maintenance activities, especially given the small number of developers involved and the frequent updates required for selectors and assertions. Developing such tools could reduce maintenance overhead and improve test suite resilience. Finally, we advocate for the development of test suite health indicators that go beyond change frequency and survival rates to assess the long-term relevance, reliability, and coverage of web GUI tests, enabling teams to make more informed decisions about test evolution and refactoring.

Declaration of Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data Availability Statement

The manuscript includes data as electronic supplementary material. In particular, datasets generated and analyzed during the study, detailed results, scripts, and additional resources useful for reproducing the study are available as part of our online appendix on Zenodo: https://zenodo.org/records/15882798.

Acknowledgements

This work has been partially supported by the Italian PNRR MUR project PE0000013-FAIR and the *HORIZON-KDT-JU-2023-2-RIA research project MATISSE* "Model-based engineering of Digital Twins for early verification and validation of Industrial Systems" (grant 101140216-2, KDT232RIA_00017). The authors would like to thank the handling editor and the anonymous reviewers for their insightful and constructive feedback throughout the review process. Their comments have been instrumental in enhancing the quality and clarity of our manuscript.

Credits

Sergio Di Meglio: Formal analysis, Investigation, Data Curation, Validation, Writing - Original Draft, Visualization. **Luigi Lucio Libero Starace:** Conceptualization, Methodology, Formal analysis, Visualization, Resources, Supervision, Writing - Review & Editing. **Valeria Pontillo:** Conceptualization, Methodology, Validation, Writing - Review & Editing. **Ruben Opdebeeck:** Conceptualization, Methodology, Validation, Writing - Review & Editing. **Coen De Roover:** Supervision, Resources, Writing - Review & Editing. **Sergio Di Martino:** Supervision, Resources, Writing - Review & Editing.

Bibliography

[1] Alégroth, E., Feldt, R., 2017. On the long-term use of visual gui testing in industrial practice: a case study. Empirical Software Engineering 22, 2937–2971.

[2] Alégroth, E., Feldt, R., Kolström, P., 2016. Maintenance of automated test suites in industry: An empirical study on visual gui testing. Information and Software Technology 73, 66–80.

[3] Altiero, F., Corazza, A., Di Martino, S., Peron, A., Libero Lucio Starace, L., 2024. Regression test prioritization leveraging source code similarity with tree kernels. Journal of Software: Evolution and Process , e2653.

[4] Alvarez, S.A., 2003. Chi-squared computation for association rules: preliminary results. Boston, MA: Boston College 13.

[5] Balsam, S., Mishra, D., 2025. Web application testing—challenges and opportunities. Journal of Systems and Software 219, 112186. doi:10.1016/j.jss.2024.112186.

[6] Barrak, A., Eghan, E.E., Adams, B., 2021. On the co-evolution of ml pipelines and source code-empirical study of dvc projects, in: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE. pp. 422–433.

[7] Battista, E., Di Martino, S., Di Meglio, S., Scippacercola, F., Lucio Starace, L.L., 2023. E2e-loader: A framework to support performance testing of web applications, in: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 351–361. doi:10.1109/ICST57152.2023.00040.

[8] Biagiola, M., Stocco, A., Mesbah, A., Ricca, F., Tonella, P., 2019. Web test dependency detection, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 154–164.

[9] Bland, J.M., Altman, D.G., 1995. Multiple significance tests: the bonferroni method. Bmj 310, 170.

[10] Bons, A., Marín, B., Aho, P., Vos, T.E., 2023. Scripted and scriptless gui testing for web applications: An industrial case. Information and Software Technology 158, 107172.

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

[12] Coppola, R., Morisio, M., Torchiano, M., 2017. Scripted gui testing of android apps: A study on diffusion, evolution and fragility, in: Proceedings of the 13th International Conference on predictive models and data analytics in software engineering, pp. 22–32.

[13] Corazza, A., Di Martino, S., Peron, A., Starace, L.L.L., 2021. Web application testing: Using tree kernels to detect near-duplicate states in automated model inference, in: Proceedings of the 15th ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–6.
[14] Cypress, . Testing Frameworks for Javascript | Write, Run, Debug | Cypress — cypress.io. https://www.cypress.io/. [Accessed 04-11-2024].
[15] Dabic, O., Aghajani, E., Bavota, G., 2021. Sampling projects in github for msr studies, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE. pp. 560–564.
[16] Devographics, . State of JavaScript 2023: Testing — 2023.stateofjs.com. https://2023.stateofjs.com/en-US/libraries/testing/. [Accessed 19-09-2024].
[17] Di Martino, S., Fasolino, A.R., Starace, L.L.L., Tramontana, P., 2021. Comparing the effectiveness of capture and replay against automatic input generation for android graphical user interface testing. Software Testing, Verification and Reliability 31, e1754.
[18] Di Martino, S., Fasolino, A.R., Starace, L.L.L., Tramontana, P., 2024. Gui testing of android applications: Investigating the impact of the number of testers on different exploratory testing strategies. Journal of Software: Evolution and Process 36, e2640.
[19] Di Meglio, S., Libero Lucio Starace, L., Di Martino, S., 2025a. E2e-loader: A tool to generate performance tests from end-to-end gui-level tests, in: 2025 IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 747–751. doi:10.1109/ICST62969.2025.10989035.
[20] Di Meglio, S., Starace, L.L.L., 2024a. Evaluating performance and resource consumption of rest frameworks and execution environments: Insights and guidelines for developers and companies. IEEE Access doi:10.1109/ACCESS.2024.3489892.
[21] Di Meglio, S., Starace, L.L.L., 2024b. Towards predicting fragility in end-to-end web tests, in: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 387–392. doi:10.1145/3661167.3661179.
[22] Di Meglio, S., Starace, L.L.L., Pontillo, V., Opdebeeck, R., De Roover, C., Di Martino, S., . Investigating the adoption and maintenance of web gui testing: Insights from github repositories — online appendix. URL: https://zenodo.org/records/15882798. [Accessed 14-07-2025].
[23] Di Meglio, S., Starace, L.L.L., Pontillo, V., Opdebeeck, R., De Roover, C., Di Martino, S., 2025b. E2EGit: A dataset of end-to-end web tests in open source projects, in: 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), pp. 836–840. doi:10.1109/MSR66628.2025.00121.
[24] Di Meglio, S., Starace, L.L.L., Pontillo, V., Opdebeeck, R., De Roover, C., Di Martino, S., 2025c. Performance testing in open-source web projects: Adoption, maintenance, and a change taxonomy, in: 41st IEEE International Conference on Software Maintenance and Evolution (ICSME 2025), IEEE.
[25] Fowler, M., 2012. The test pyramid. MartinFowler.com.
[26] Glaser, B.G., 2016. Open coding descriptions. Grounded theory review 15.
[27] Gortázar, F., Maes-Bermejo, M., Gallego, M., Contreras Padilla, J., 2021. Looking for the needle in the haystack: End-to-end tests in open source projects, in: Paiva, A.C.R., Cavalli, A.R., Ventura Martins, P., Pérez-Castillo, R. (Eds.), Quality of Information and Communications Technology, Springer International Publishing, Cham. pp. 40–48.
[28] Grechanik, M., Xie, Q., Fu, C., 2009. Experimental assessment of manual versus tool-based maintenance of gui-directed test scripts, in: 2009 ieee international conference on software maintenance, IEEE. pp. 9–18.
[29] Jiang, Y., Adams, B., 2015. Co-evolution of infrastructure and source code-an empirical study, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, IEEE. pp. 45–55.
[30] Junior, N., Costa, H., Karita, L., Machado, I., Soares, L., 2021. Experiences and practices in gui functional testing: A software practitioners' view, in: Proceedings of the XXXV Brazilian Symposium on Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 195–204. doi:10.1145/3474624.3474640.
[31] Katalon, . Low-code Test Automation in One Place | Katalon Platform — katalon.com. https://katalon.com/low-code-test-automation. [Accessed 10-07-2025].
[32] Kitchenham, B., Madeyski, L., Budgen, D., Keung, J., Brereton, P., Charters, S., Gibbs, S., Pohthong, A., 2017. Robust statistical methods for empirical software engineering. Empirical Software Engineering 22, 579–630.
[33] Kresse, A., Kruse, P.M., 2016. Development and maintenance efforts testing graphical user interfaces: a comparison, in: Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation, pp. 52–58.
[34] Leotta, M., Clerissi, D., Ricca, F., Tonella, P., 2016. Approaches and tools for automated end-to-end web testing, in: Advances in Computers. Elsevier. volume 101, pp. 193–237.
[35] Leotta, M., García, B., Ricca, F., Whitehead, J., 2023. Challenges of end-to-end testing with selenium webdriver and how to face them: A survey, in: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 339–350. doi:10.1109/ICST57152.2023.00039.
[36] Leotta, M., Ricca, F., Tonella, P., 2021. Sidereal: Statistical adaptive generation of robust locators for web testing. Software Testing, Verification and Reliability 31, e1767.
[37] Li, J., Ahmed, I., 2023. Commit message matters: Investigating impact and evolution of commit message quality, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE. pp. 806–817.
[38] Lubsen, Z., Zaidman, A., Pinzger, M., 2009. Using association rules to study the co-evolution of production & test code, in: 2009 6th IEEE International Working Conference on Mining Software Repositories, IEEE. pp. 151–154.
[39] Maes-Bermejo, M., Gallego, M., Gortázar, F., Robles, G., Gonzalez-Barahona, J.M., 2022. Revisiting the building of past snapshots—a replication and reproduction study. Empirical Software Engineering 27, 65.
[40] Mann, H.B., Whitney, D.R., 1947. On a test of whether one of two random variables is stochastically larger than the other. The annals of mathematical statistics , 50–60.
[41] Marsavina, C., Romano, D., Zaidman, A., 2014. Studying fine-grained co-evolution patterns of production and test code, in: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, IEEE. pp. 195–204.

[42] McIntosh, S., Adams, B., Nguyen, T.H., Kamei, Y., Hassan, A.E., 2011. An empirical study of build maintenance effort, in: Proceedings of the 33rd International Conference on software engineering, pp. 141–150.
[43] Melyawati, N.L.P., Asana, I.M.D.P., Putri, N.W.S., Atmaja, K.J., Sudipa, I.G.I., 2024. Comparison of automation testing on card printer project using playwright and selenium tools. Journal of Computer Networks, Architecture and High Performance Computing 6, 1309–1320.
[44] Morán, J., Augusto, C., Bertolino, A., De La Riva, C., Tuya, J., 2020. Flakyloc: flakiness localization for reliable test suites in web applications. Journal of Web Engineering 19, 267–296.
[45] Nelder, J., Wedderburn, R., 1972. Generalized linear models. Journal of the Royal Statistical Society: Series A (General) 135, 370–384.
[46] O'brien, R., 2007. A caution regarding rules of thumb for variance inflation factors. Quality & quantity 41, 673–690.
[47] Pecorelli, F., Palomba, F., De Lucia, A., 2021. The relation of test-related factors to software quality: A case study on apache systems. Empirical Software Engineering 26.
[48] Playwright, . Fast and reliable end-to-end testing for modern web apps | Playwright — playwright.dev. https://playwright.dev/. [Accessed 04-11-2024].
[49] Pontillo, V., Palomba, F., Ferrucci, F., 2024. Test code flakiness in mobile apps: The developer's perspective. Information and Software Technology 168, 107394.
[50] Puppeteer, . Puppeteer | Puppeteer — pptr.dev. https://pptr.dev/. [Accessed 04-11-2024].
[51] Ramya, P., Sindhura, V., Sagar, P.V., 2017. Testing using selenium web driver, in: 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT), IEEE. pp. 1–7.
[52] Rodríguez-Valdés, O., Vos, T.E., Aho, P., Marín, B., 2021. 30 years of automated gui testing: a bibliometric analysis, in: Quality of Information and Communications Technology: 14th International Conference, QUATIC 2021, Algarve, Portugal, September 8–11, 2021, Proceedings 14, Springer. pp. 473–488.
[53] Selenium, . Selenium — selenium.dev. https://www.selenium.dev/. [Accessed 04-11-2024].
[54] Shewchuk, Y., Garousi, V., 2010. Experience with maintenance of a functional gui test suite using ibm rational functional tester., in: SEKE, pp. 489–494.
[55] Soto-Sánchez, Ó., Maes-Bermejo, M., Gallego, M., Gortázar, F., 2022. A dataset of regressions in web applications detected by end-to-end tests. Software Quality Journal , 1–30.
[56] Spadini, D., Aniche, M., Bacchelli, A., 2018. Pydriller: Python framework for mining software repositories, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, New York, NY, USA. p. 908–911.
[57] Stocco, A., Leotta, M., Ricca, F., Tonella, P., 2017. Apogen: automatic page object generator for web testing. Software Quality Journal 25, 1007–1039.
[58] Stocco, A., Yandrapally, R., Mesbah, A., 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, pp. 503–514.
[59] Tian, Y., Zhang, Y., Stol, K.J., Jiang, L., Liu, H., 2022. What makes a good commit message?, in: Proceedings of the 44th International Conference on Software Engineering, pp. 2389–2401.
[60] Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., 2017. There and back again: Can you compile that snapshot? Journal of Software: Evolution and Process 29, e1838.
[61] Vargha, A., Delaney, H.D., 2000. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics 25, 101–132.
[62] Vos, T.E., Aho, P., Pastor Ricos, F., Rodriguez-Valdes, O., Mulders, A., 2021. testar–scriptless testing through graphical user interface. Software Testing, Verification and Reliability 31, e1771.
[63] Wang, S., Wen, M., Liu, Y., Wang, Y., Wu, R., 2021. Understanding and facilitating the co-evolution of production and test code, in: 2021 IEEE International Conference on software analysis, evolution and reengineering (SANER), IEEE. pp. 272–283.
[64] Zaidman, A., Van Rompaey, B., Demeyer, S., Van Deursen, A., 2008. Mining software repositories to study co-evolution of production & test code, in: 2008 1st International Conference on software testing, verification, and validation, IEEE. pp. 220–229.
[65] Zaidman, A., Van Rompaey, B., Van Deursen, A., Demeyer, S., 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empirical Software Engineering 16, 325–364.
[66] Zhang, H., Liao, L., Ding, Z., Shang, W., Narula, N., Sporea, C., Toma, A., Sajedi, S., 2024. Towards a robust waiting strategy for web gui testing for an industrial software system, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, pp. 2065–2076.