Master's thesis

Master's Programme in Computer Science

Addressing End-to-End Testing Challenges with Cypress

Noora Rytilä

September 17, 2025

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki, Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Noora Rytilä

Työn nimi — Arbetets titel — Title

Addressing End-to-End Testing Challenges with Cypress

Ohjaajat — Handledare — Supervisors

Prof. M. V. Mäntylä

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	September 17, 2025	66 pages, 4 appendix pages

Tiivistelmä — Referat — Abstract

Background: End-to-end (E2E) testing is a software testing method that tests the complete application workflow. The goal is to verify that the application functions as expected from the end user's perspective. E2E testing has multiple challenges, such as fragility, maintainability, flakiness, and long execution times. **Aims:** This thesis aims to identify and categorise the challenges of web application E2E testing and develop practical solutions to address them. The focus is on addressing challenges related to the test design and implementation using the Cypress testing framework. **Method:** A Systematic Literature Review (SLR) was conducted to categorise and analyse the challenges of E2E testing. Design Science Research (DSR) was applied to develop a set of E2E test suites in iterative development iterations. Key approaches to address the challenges were using data attributes as locators, the Page Object Model, and API stubbing. **Results:** Using data attribute locators remarkably reduced fragility when compared to other locators. Using the Page Object Model (POM) improved maintainability, and API stubbing improved the test execution time. While Cypress's built-in retry mechanism reduced flakiness issues, some flakiness remained. **Conclusions:** This study identified and categorised key E2E testing challenges and demonstrated that they can be addressed through carefully designing and implementing the test suites. Techniques like robust locators, POM, and API stubbing improve test reliability, maintainability, and performance. Further automation and real-world use case validation are necessary to generalise the findings and address remaining issues.

University of Helsinki guidelines allow the usage of LLMs in theses. ChatGPT-4o has been used to refine language and writing.

ACM Computing Classification System (CCS)
Software and its engineering → Software creation and management → Software verification and validation → Software defect analysis → Software testing and debugging

Avainsanat — Nyckelord — Keywords

Software Testing, Test Automation, End-to-End Testing, Cypress Framework

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction

Software testing is a central process in software development. Software testing ensures quality, detects software defects and bugs early in the application's lifecycle and promotes cost-effectiveness [30]. Manual software testing is tedious, labour-intensive, and at risk of human error. Automated software testing improves software reliability and frees up developers' and quality assurers' time [3]. End-to-end testing is an automated software testing approach that tests an application's workflow from start to finish. In end-to-end testing, the goal is to verify that all software components, such as the user interface, backend, and database, work as expected from the end user's perspective [6].

Common challenges of E2E testing are test fragility, maintenance difficulties, flakiness, and long execution times. These challenges are well-recognised in academic literature and industry practice [22, 48]. To better understand the challenges, this thesis starts with a Systematic Literature Review (SLR). The SLR identifies and categorises the challenges of E2E testing in modern web applications. This review, reported in Section 2, answers the first research question:

- **RQ1:** What are the challenges of E2E testing?

After identifying and motivating the problem through SLR, this thesis adopts the Design Science Research (DSR) methodology to iteratively design and evaluate testing strategies to address specific E2E testing challenges. The developed artefact is a practical E2E test suite implemented using the Cypress testing framework. The testing strategy is evaluated in the context of a real-world open-source application, Vuestic Admin.

The following research questions guide the design and development of the artefact:

- **RQ2:** How can the fragility of E2E tests be mitigated?
- **RQ3:** How can the maintainability and modularity of E2E tests be improved?
- **RQ4:** How can the execution times of E2E tests be reduced?
- **RQ5:** What strategies can reduce flakiness in E2E tests?

Each challenge is addressed in a separate artefact development iteration, using techniques such as robust locator strategies, the Page Object Model, and API stubbing. The results suggest these techniques can significantly improve test robustness, maintainability, and execution efficiency.

The rest of the thesis is structured as follows: Section 2 presents background information and the findings of the Systematic Literature Review. Section 3 presents the Design Science Research method, the research context, and describes the research and artefact development process. Section 4 presents the results of implemented solutions, Section 5 discusses the findings, and Section 6 concludes the thesis.

2 Background

This chapter provides theoretical and practical background information for the rest of the thesis. Section 2.1 introduces the fundamentals of software testing, and Section 2.2 focuses specifically on end-to-end (E2E) testing of web applications, with a focus on goals, approaches, and tools. Section 2.3 presents the process and findings of a systematic literature review, which identified and categorised E2E testing challenges.

2.1 Software Testing

Software testing is a crucial software development process aiming to verify that an application works as intended. Software testing has three goals: **verification**, **validation**, and **defect detection** [7]. *Verification* confirms that the software meets its technical requirements, and the *validation* process confirms that it meets business requirements. *Defect detection* identifies errors, bugs, and deviations between the requirements and actual outcome. The importance of software testing is highlighted by the fact that finding and fixing bugs takes up about half of the total development time in a typical software project [23].

Software testing can be categorized into two groups: manual and automated testing [23]. In manual testing, a human tester does the testing activities by using the software as an end user. In automated testing, the testing activities are automated by using specific testing software. The Software testing V-Model (Figure 2.1) shows how software testing types and software development process steps relate. The V-Model has multiple levels of testing: **unit testing** is the lowest level of testing that focuses on testing individual software components. **Integration testing** tests the communication between individual components, and **system testing** verifies that the system meets its requirements. **Acceptance testing**, done by customers or end users, ensures that the software can be used as intended [27].

The V-Model is a structured approach to software development and testing, but one of its main limitations is its inflexibility. It is unsuitable for modern development projects and environments where applications are constantly updated [5]. Agile testing methodologies provide a more flexible software development and testing method. Agile methodologies support iterative development with small and incremental updates to the application. In

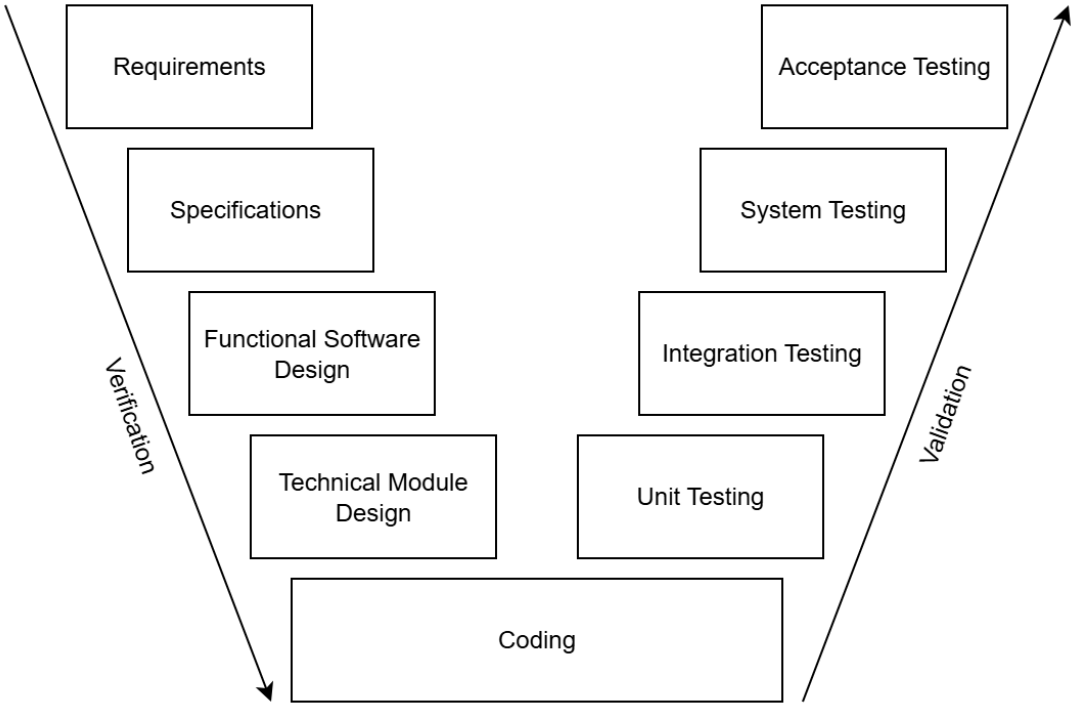

Figure 2.1: Software testing V-Model, adapted from [5] and [7]

agile development, the software is frequently delivered to production, and feedback from each iteration guides the following [5]. Testing plays a key role in agile development iterations as it ensures the application functionality after changes and detects defects and bugs before production.

2.2 End-to-End Web Testing

This subsection introduces the key concepts and techniques of end-to-end (E2E) testing of web applications. Section 2.2.1 begins with an overview of E2E testing, describing its role in software quality assurance and in relation to other types of testing. Section 2.2.2 describes the most common approach to E2E web testing: script-based testing. Section 2.2.3 introduces the concept of locators, which are used to locate elements in E2E tests. Finally, Section 2.2.4 introduces the Page Object Model (POM) design pattern, a common abstraction used in E2E tests to improve maintainability and modularity of E2E tests.

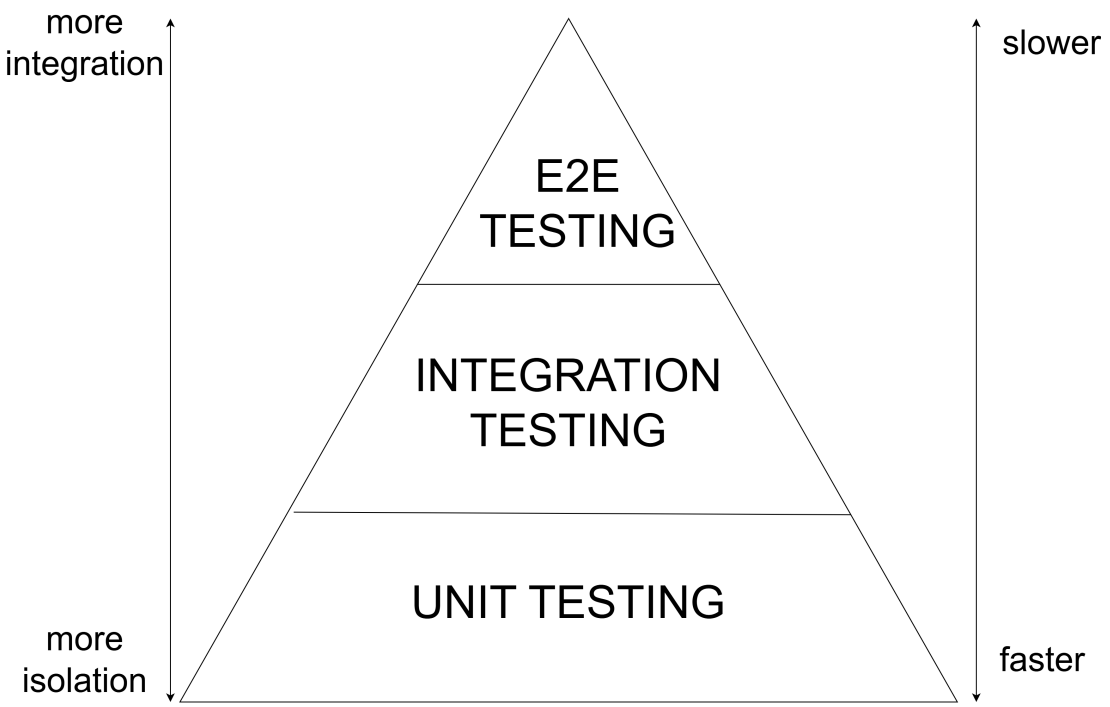

Figure 2.2: Software Testing Pyramid, adapted from [20]

2.2.1 Introduction to End-to-End Web Testing

End-to-end (E2E) web testing is a type of software testing that tests the entire web application workflow from an end user's perspective. The main goal is to ensure that the application and its different layers, including the user interface, backend, integrations, and database, function as intended [6]. E2E testing can be done manually or automatically. It can be applied to various applications, such as desktop, mobile, and web applications. In this thesis, the focus is on end-to-end testing of web applications.

E2E testing can serve the purposes of both system testing and acceptance testing in the V-Model (Figure 2.1) because E2E tests verify that the entire application functions correctly from both the technical and end-user perspectives.

The Software Testing Pyramid (Figure 2.2) presents the recommended balance between different levels of automated tests. It suggests having many unit tests, fewer integration tests, and the smallest number of E2E tests. This is based on the idea that lower-level tests are faster to run, easier to maintain, and provide quicker feedback for developers. E2E tests are slower, more complex, and more fragile and should be used more selectively [20].

In manual E2E testing, a tester follows a set of predefined test scenarios directly on the web application's graphical user interface (GUI), acting as an end-user. The key advantage of manual testing is that testers can perform many different scenarios, even those that are challenging or impractical to automate. However, manual testing is costly, time-consuming, and prone to errors [15].

In automated E2E testing, scripts or specialised software tools are used to simulate user interactions. These tests remotely control a web browser and navigate through different URLs and subpages of a web application. The tests can also extract information from the web page and simulate keyboard inputs [15]. Automated E2E testing through the GUI saves time, boosts customer satisfaction, and enhances collaboration between developers and testers [2]. The challenges of E2E testing are discussed in Section 2.3.

2.2.2 Script-based testing

The industry's most commonly used approach for E2E test automation is script-based testing. Script-based testing involves manually creating test cases and automating their execution [1]. This type of testing can be categorised into two groups: **record and replay testing** and **programmable testing**.

In **record and replay testing**, user interactions with the application's graphical user interface are recorded and converted into a test script. These scripts can be replayed during regression testing to verify that existing functionality remains unchanged [34].

In **programmable testing**, test scripts are written in code using testing frameworks. In the context of web application testing, these frameworks offer tools to locate elements on the web page and simulate user interactions [34].

Leotta et al. [33] compared these two approaches, showing that programmable tests generally require more time to develop initially but are significantly easier to maintain. The study found that, in most cases, the total cost of programmable testing becomes lower than record and replay testing as the software evolves and grows. The cost savings increase even further with each subsequent release.

2.2.3 Locators

Locators are specific constructs used in end-to-end test automation tools and scripts to locate and identify elements on a web page [37]. They identify elements such as buttons,

form fields, and navigation links. Locators enable interactions with the web page, like clicking a button or navigating to a subpage by clicking a navigation link.

Locators can be categorised into three groups based on their approach to identifying elements:

- **Coordinate-based locators** locate elements based on their screen coordinates [13]. Formerly, records and replay tools generated coordinate-based locators, but this approach is no longer widely used due to its significant fragility [34].
- **Visual locators** use image recognition techniques to locate elements. Visual locators are helpful for some use cases, like map applications, but are less robust and less efficient compared to layout-based locators [34].
- **Layout-based locators** use the properties of the DOM, an object-oriented programming interface of the web page [42]. These properties include the relative or absolute path of the element within the DOM tree, the element's text content, HTML attributes (e.g., id, class, name), or the type of the element (e.g., div, button) [15]. This approach is also called DOM-based locators.

A Robust locator is an expression that can accurately and uniquely identify an element in different versions of the same web application, even when the application evolves. Robust locators are less likely to break when minor GUI changes are made, such as modifications to an element's properties or changes to its parent and child elements [37].

In a study by Leotta et al. [34], the authors compare visual and DOM-based locators. Based on the study, DOM-based locators are more robust, faster to develop and execute. The number of broken locators was lower in experiments that used DOM-based locators, and in most cases, repair times were shorter.

2.2.4 Page Object Model (POM) Design Pattern

The Page Object Model (POM) is a typical design pattern used in end-to-end (E2E) testing. Each web page is represented as an object. Page Object Models hide the details of a web page and offer methods to interact with its features. The models are written in the same programming language as the test cases, allowing developers to operate on a higher level of abstraction, making the test scripts easier to maintain and reuse [32].

The POM design pattern reduces coupling between test cases and implementation code. Decoupling reduces fragility and ensures that changes in the web application implementation code do not require the test code to be significantly updated [19, 31]. Adopting the POM increases the time needed to develop test scripts, but the overhead decreases as the test suite and application size grow. Research suggests that for large test suites, typical in industrial settings, the return on investment becomes significant because adopting POM makes writing new tests faster [31]. The time and effort spent repairing test suites are significantly reduced when using POM [32]. The con of increased development time has been attempted to mitigate with the automatic generation of Page Objects [59, 63].

2.3 Challenges of E2E Testing

End-to-end (E2E) testing of web applications is critical for ensuring their quality, but it has multiple significant challenges. The complexity of modern web applications, the need to simulate real user interactions and the effort needed to maintain test suites as applications grow cause these challenges.

2.3.1 Systematic Literature Review

A systematic literature review (SLR) was conducted to identify and analyse the key challenges of E2E testing. The process was roughly based on the guidelines proposed by Kitchenham et al. for systematic reviews in software engineering [29] and the "Mini Systematic Review Guide for Master's Thesis" by Mäntylä [45]. The research process followed these phases:

1. Construct a search string
2. Define and apply inclusion and exclusion criteria
3. Analyse the included studies
4. Report the findings

Table 2.1: SLR inclusion and exclusion criteria

I1	The study discusses or addresses one or more challenges associated with automated script-based E2E testing of modern web applications
E1	The study focuses on some other form of testing than end-to-end testing (e.g., unit testing, acceptance testing)
E2	The study focuses on some other than web application testing (e.g., mobile, IoT testing, or desktop application testing)
E3	The study focuses on manual testing or automated visual GUI testing

The literature search was done using the Scopus database. The search string was iterated multiple times to ensure good coverage of relevant studies. The final version of the search string was:

> ("end-to-end test" OR "end to end test" OR "end to end testing" OR "end-to-end testing" OR "e2e test" OR "e2e testing" OR "e2e testing" OR "GUI testing" OR "GUI based automation testing" OR "UI testing" OR "user interface testing" OR "Web testing" AND "challenges" OR "risks" OR "issues" OR "problems" OR "difficulty" OR "difficulties" OR "disadvantage" AND NOT "mobile" AND NOT "android" AND NOT "iot" AND NOT "quantum")

Search results were limited to studies written in English and limited to the subject area of Computer Science. Other subject areas were excluded. The search string resulted in 200 papers published between 2003 and 2025. An additional 39 studies were included through backwards snowballing and prior knowledge of relevant literature, resulting in 239 studies. After applying the inclusion and exclusion criteria, 83 studies were left. Table 2.1 presents the inclusion and exclusion criteria. After skimming and reading the included papers, 53 studies were left for the final analysis. Appendix A provides a complete list of included studies. The process of searching and reviewing studies is illustrated in Figure 2.3.

The 53 selected studies were analysed, and based on this analysis, 15 challenges were identified. These challenges are summarised in Table 2.2. These challenges are categorised into three categories based on the type and cause. The categories help to structure the discussion and analysis of the challenges. Figure 2.4 shows the categories and associated challenges. Some challenges are grouped into multiple categories because they overlap.

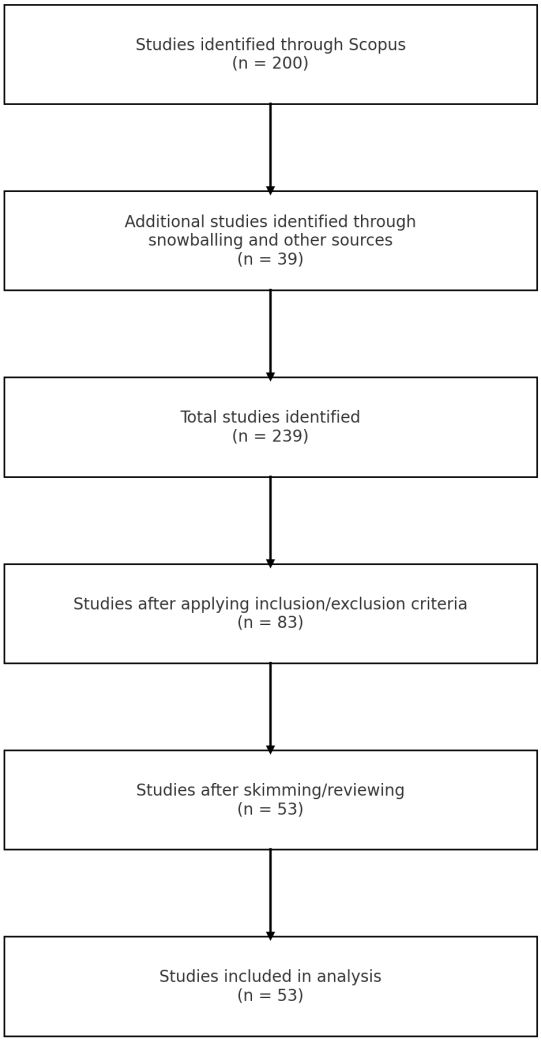

Figure 2.3: SLR study selection process

E2E Testing Challenge Categories:

1. Challenges related to how the tests are designed and developed
2. Challenges related to the characteristics of web applications
3. Challenges associated with tooling, required resources, and expertise

1. Test Design and Implementation Challenges

This category includes challenges related to how E2E tests are designed, developed, and maintained. These challenges include poor maintainability, strong coupling between the test and application code, test fragility and long execution times. These challenges are technical and can be mitigated by focusing on good design and development of the E2E test suites.

2. Web Application Complexity Challenges

Modern web applications have dynamic and interactive interfaces, and complex user behaviour is encouraged. This complexity makes E2E testing challenging [61]. The challenges in this group include dynamic content updates, complex input scenarios, flakiness and high resource usage. These challenges are related to the nature of modern web applications.

3. Tooling & Expertise Requirements Challenges

This category combines challenges related to the tools, environments, and skills required for E2E testing. Challenges in this group include challenges with tools, a high level of required expertise, and assertability. These challenges highlight the practical challenges practitioners face when implementing E2E testing.

The following subchapters discuss each of the 15 identified challenges in more detail, following the same order as in Table 2.2. Each challenge is described, and its causes and mitigation strategies are discussed.

2.3.2 Maintainability Challenges

1. Test Design and Implementation Challenge

Maintainability of the developed test suites is one of the most significant challenges in E2E testing [1, 15, 22]. More than half (27) of the analysed 58 studies pointed the maintainability challenge. As applications evolve, test suites must be updated to reflect the changes in functionality [57, 9]. Keeping test code working takes ongoing effort from

Table 2.2: E2E testing challenges

Challenge	**#**	**Mentioned in**
Maintainability	27	S2, S3, S8, S9, S10, S12, S15, S17, S18, S24, S27, S31, S34, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S51, S52
Fragility	27	S2, S3, S5, S8, S9, S12, S13, S15, S17, S18, S20, S25, S31, S34, S35, S36, S38, S39, S40, S41, S42, S43, S44, S45, S47, S49, S52
Difficulty in Test Creation	17	S7, S8, S13, S15, S17, S20, S24, S25, S28, S32, S34, S35, S36, S38, S42, S45, S48
Long Execution Times	15	S1, S6, S8, S12, S16, S18, S19, S22, S23, S25, S26, S32, S33, S36, S49
High Costs	13	S3, S8, S12, S18, S27, S32, S38, S41, S45, S46, S47
Strong Coupling	11	S3, S8, S12, S18, S27, S32, S38, S41, S45, S46, S47
Complex User Input Scenarios	10	S3, S14, S27, S28, S29, S36, S40, S42, S50, S53
Flakiness	7	S4, S8, S11, S16, S25, S30, S32
High Resources	7	S8, S12, S22, S23, S32, S37, S49
Required Skills	7	S8, S15, S17, S45, S48, S49, S50
Code Duplication	6	S5, S8, S12, S21, S34, S38
Test Dependencies	6	S8, S12, S19, S23, S32, S49
Assertability	5	S8, S25, S29, S31, S36
Dynamic Content Update	3	S8, S14, S40
Tool Challenges	3	S8, S15, S48

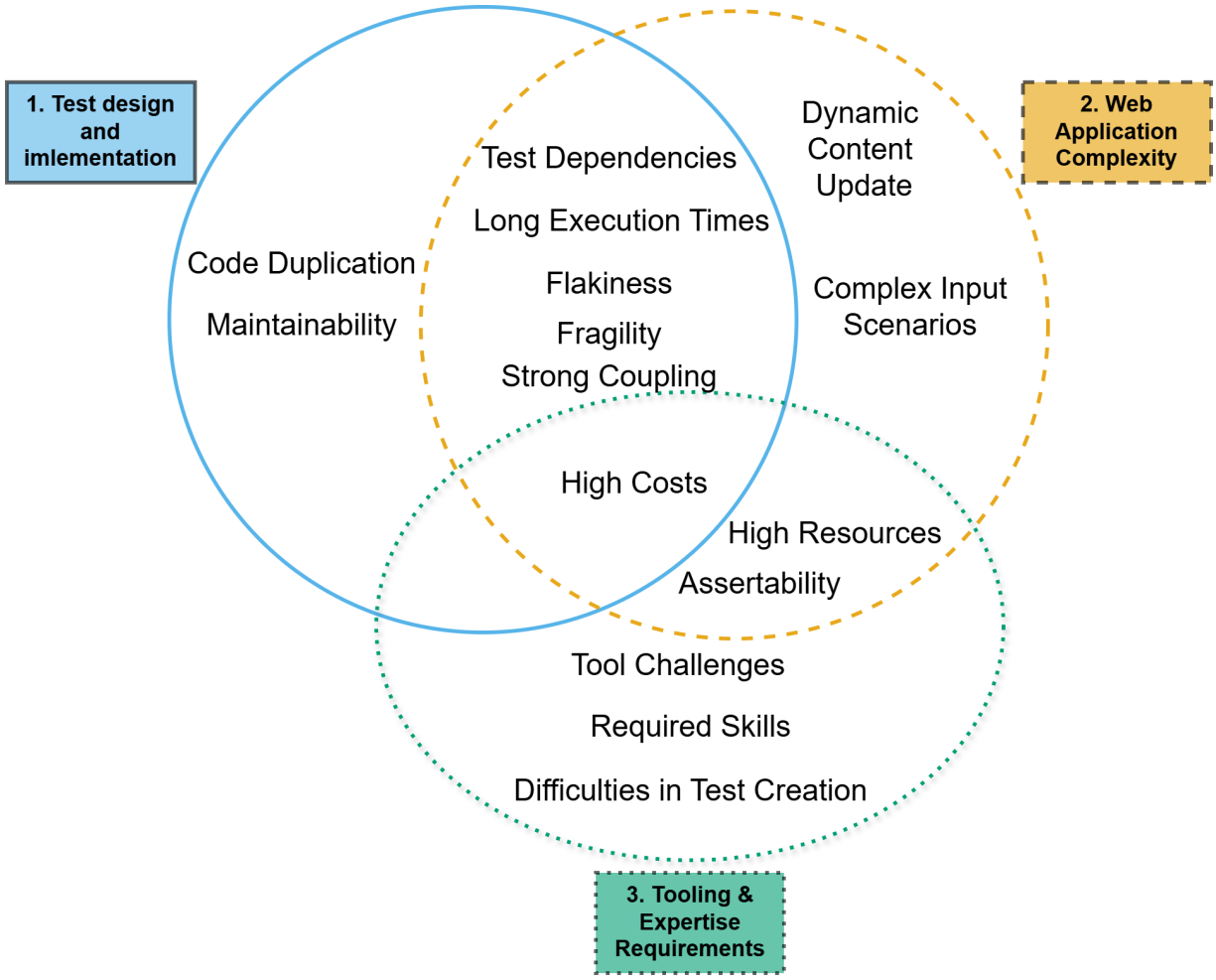

Figure 2.4: E2E testing challenge categories

skilled developers, which increases costs over time. Since E2E tests are often fragile, this becomes even more of a problem. For example, a test that locates a list item by its index may fail if a new item is added [31]. Maintainability can be improved by designing more reliable and robust test cases and reducing coupling between the application and test code [13, 56].

2.3.3 Fragility

1. Test Design and Implementation Challenge

2. Web Application Complexity Challenge

E2E tests break easily when the application changes, even if the feature under test is not changed [57]. This tendency is called fragility. Fragility often causes false positives - test failures that happen even though nothing is broken in the application [9]. When these false alarms occur, it becomes harder to trust the tests, which lowers their overall usefulness.

Like maintainability, the challenge of fragility was also discussed in more than half of the analysed studies, 27/53. Fragility is a core challenge in E2E testing and a key factor contributing to the maintenance difficulties discussed in the previous subchapter. Addressing fragility is essential to ensure that automated E2E tests remain robust and maintainable as the application evolves. The problem is prevalent in web application testing, where even minor UI changes can cause tests to fail [25]. For example, adding a new button between two existing ones usually does not prevent the user from using a functionality. However, a test that finds the button based on its position in a grid could break [62]. In such cases, the failure comes not from a functional bug but from the test's reliance on implementation details.

Fragility is commonly caused by how elements are located. Too specific locators may break when minor changes to the application layout are made. On the other hand, too generic locators can match multiple or the wrong elements. This challenge worsens when the web application's Document Object Model (DOM) is complex [15]. Fragile locators cause 73.62% of test breakages, so improving locator robustness is the most effective way to address fragility issues [25].

2.3.4 Difficulties in Test Creation

3. Tooling & Expertise Requirements Challenge

Developing E2E tests for web applications is widely recognised as challenging and resource-intensive. 17 of the 53 analysed studies discussed the challenge of creating E2E tests. E2E tests are typically created manually, which requires testers to write code that simulates user actions, such as filling out a form and clicking a submit button through the user interface [39]. This process is time-consuming and costly, requiring the tester to have excellent domain knowledge. The complexity of modern web applications also makes test creation difficult. The applications have complex business logic and a complex set of possible user input combinations [35, 22]. Also, synchronisation between the application under test and the test code makes test creation difficult. In dynamic web environments where content loads asynchronously, choosing appropriate waiting conditions or timing between steps can be challenging and prone to errors [36, 1].

2.3.5 Long Execution Times

1. Test Design and Implementation Challenge

2. Web Application Complexity Challenge

E2E tests are slower to execute compared to lower-level tests, such as unit or integration tests. E2E tests simulate user interactions in a browser, render a complete application stack and validate all layers of functionality, including backend, database and frontend. 15/53 of the analysed articles discussed long execution times. In many cases, tests have fixed time delays waiting for elements to be rendered or an API call to be responded to [40]. As a result, running a full suite of E2E tests can easily take hours or even require overnight execution [1, 35]. This reduces how often tests can be executed and slows down feedback for developers. The slow execution of E2E test suites is recognised as a drawback by 29.17% of respondents in a survey about the Selenium ecosystem, a common industry standard E2E testing tool [22]. The challenge of long execution times can be mitigated by using explicit waits [40], optimising the tests [64] and running tests in parallel [35].

2.3.6 High Costs

1. Test Design and Implementation Challenge

2. Web Application Complexity Challenge

3. Tooling & Expertise Requirements Challenge

13 of the 53 analysed studies discussed the challenge of high E2E testing costs. As shown in Figure 2.4, this challenge is related to all three categories of challenges. Firstly, test design and implementation challenges and web application complexity challenges increase costs due to time spent on maintaining the tests, debugging failures, and fixing broken locators [22, 36, 57]. Secondly, the challenge of acquiring the required expertise and tools is directly linked to high costs. The required tools generate costs, and additional costs are associated with the expertise and high resources needed to develop and execute the tests [22]. The high costs can be most effectively mitigated by addressing other challenges that contribute to the costs.

2.3.7 Strong Coupling

1. Test Design and Implementation Challenge

2. Web Application Complexity Challenge

Strong coupling between the test code and the implementation code of the application under test is a common challenge in E2E testing, as discussed in 11 of the 53 analysed studies. Strong coupling is caused by the tests relying on the DOM structure of the web page when identifying and interacting with elements. E2E tests should not be closely tied to how the web application is built but instead focus on testing functionality [21].

Strong coupling makes test suites fragile when even small changes in the application can break the tests [22]. Strong coupling is a primary challenge that contributes to fragility and maintenance burden. Using the Page Object Design pattern is the most common proposal to reduce coupling [22, 35, 36].

2.3.8 Complex Input Scenarios

2. Web Application Complexity Challenge

Ten of the 53 analysed studies discussed the challenge of complex input scenarios. The business logic of modern web applications is highly complex, and some states and func-

tionalities are accessible only through lengthy and complex user interactions. Bugs and defects hide under these complex interaction sequences. The challenge of complex user input scenarios requires domain knowledge from the test developer and is one of the primary causes of the high costs associated with E2E testing [22, 36, 61]. The challenge can be addressed using the Page Object design pattern, which adds a layer of abstraction between the tests and user action on the application under test [22]. Artificial Intelligence and Machine Learning can generate complex tests from natural language inputs [22, 39].

2.3.9 Flakiness

1. Test Design and Implementation Challenge

2. Web Application Complexity Challenge

The challenge of flakiness was discussed in 7/53 of the studies analysed. Flakiness refers to the phenomenon where tests fail unpredictably, even without any changes made to the application under test [21]. Flakiness makes it challenging to trust the test results [47]. Asynchronous interactions and non-deterministic execution orders between the client and server code in web applications cause flakiness. For example, if a test continues execution before an API request response is received, it might fail in one run but succeed in the next run if the response is received more quickly [21]. Environmental factors such as memory and CPU also cause flakiness [47].

Flaky test suites waste developers' time when they spend time on debugging test failures that do not flag actual defects and bugs in the application [49]. Flaky tests increase the cost of end-to-end testing through the needed developer expertise and time, reducing confidence in the testing process [52].

Flakiness can be mitigated using explicit waits instead of fixed time delays, such as thread sleeps [21, 49]. Careful design of tests with flakiness in mind, and keeping test cases short and atomic, reduces the flakiness of the test suites [35].

2.3.10 High Resources

2. Web Application Complexity Challenge

3. Tooling & Expertise Requirements Challenge

7 of the 53 studies analysed discussed the requirement of high resources as a key challenge of E2E testing. High resources are needed to execute the entire system that runs the tests

and to execute the test suites in the cloud as part of the CI/CD pipeline [4]. This challenge contributes to the challenge of high costs discussed earlier. Focusing on resource and test execution time optimisations is the most effective way to address this challenge [4].

2.3.11 Required Skills

3. Tooling & Expertise Requirements Challenge

The challenge of required skills and expertise was discussed in 7 of the analysed studies. E2E testing frameworks often require proficiency in programming languages (such as Java, Python, or JavaScript) to write and maintain the test scripts [41], as well as automation skills [48]. Maintenance and troubleshooting require technical skills, including understanding traces and error reports. To mitigate one of the most significant challenges, fragility, it is essential that developers writing test scripts have the necessary skills and knowledge about robust locators [38]. Translating natural language requirements into automated tests is challenging [58] and requires developers working on test automation to have domain knowledge [14].

2.3.12 Code Duplication

1. Test Design and Implementation Challenge

Six of the 53 analysed studies discussed the challenge of duplicated test code in E2E tests. If following best practices for atomic testing, each test in an E2E test suite needs to perform the setup, such as navigating to the correct page. Code duplication grows the size of test suites and makes creation and maintenance difficult [58].

To reduce code duplication, the Page Object design pattern is proposed by studies [25, 35, 58]. In the Page Object Model, interaction with web elements is centralised in the page object code, and thus, test cases do not have to repeat the same interaction code multiple times.

2.3.13 Test Dependencies

1. Test Design and Implementation Challenge

2. Web Application Complexity Challenge

Six of the analysed studies discussed the test dependencies as a key challenge in E2E testing. Test dependencies mean that the execution of one test case influences the outcome of another [61]. Ideally, all tests in a suite should be independent so that changing the execution order of the tests does not affect the results [8, 51]. In practice, developers often create tests that depend on each other due to the difficulty in creating isolated program states for each test case or reusing states to reduce execution times [8]. Keeping test cases atomic and independent mitigates the challenge; however, independent test cases often lengthen execution times and pose challenges related to test code duplication.

2.3.14 Assertability

2. Web Application Complexity Challenge

3. Tooling & Expertise Requirements Challenge

Five of the analysed studies discussed the challenges of assertability and the lack of test oracles. Assertions are a fundamental part of tests that compare the expected results (the test oracle) and the actual outcome from the application under test [22]. In web applications, determining whether a state or execution result is correct can be challenging. Many failures cannot be captured as explicit run-time exceptions and hangs [14]. Asserting the presence of a particular bug, on the other hand, is often fragile, as texts in the application are frequently changed [28].

2.3.15 Dynamic Content Update

2. Web Application Complexity Challenge

Three of the analysed studies discussed the dynamic content update of web applications as a challenge of E2E testing. Modern web applications are complex, making testing inherently challenging, especially when testing applications where content changes dynamically [63, 65]. Dynamic content can hide bugs within the application, making exploring the functionality thoroughly using automated E2E testing tools challenging. Dynamic content update is one of the major contributors to the flakiness and fragility issues discussed

earlier in this section [35]. To address these challenges, approaches such as explicit waits [35] and Page Objects [63] should be used.

2.3.16 Tool Challenges

3. Tooling & Expertise Requirements Challenge

Three of the analysed studies mentioned tool challenges as a key challenge of E2E testing. The tools used in E2E testing are complex and have a steep learning curve [41]. Writing, reading and editing test scripts is challenging. Thus, tool challenges are also related to the challenge of required skills discussed earlier in this section.

3 Research Method and Process

This thesis followed the Design Science Research (DSR) methodology [26]. DSR is a problem-solving research approach that aims to create artefacts to address identified issues. DSR consists of a cyclical process of building and evaluating an artefact. The process is illustrated in Figure 3.1.

This Chapter first introduces the research questions of the study in Section 3.1, then explains the research context in Section 3.2, and finally describes the research and artefact development process in Section 3.3.

3.1 Research Questions

The objective of this study was to gain a deeper understanding of the challenges of end-to-end testing, to identify potential solutions, implement them in an artefact, and to evaluate their effectiveness within an E2E testing framework.

- **RQ1:** What are the challenges of E2E testing?
- **RQ2:** How can the fragility of E2E tests be mitigated?
- **RQ3:** How can the maintainability and modularity of E2E tests be improved?
- **RQ4:** How can the execution times of E2E tests be reduced?
- **RQ5:** What strategies can reduce flakiness in E2E tests?

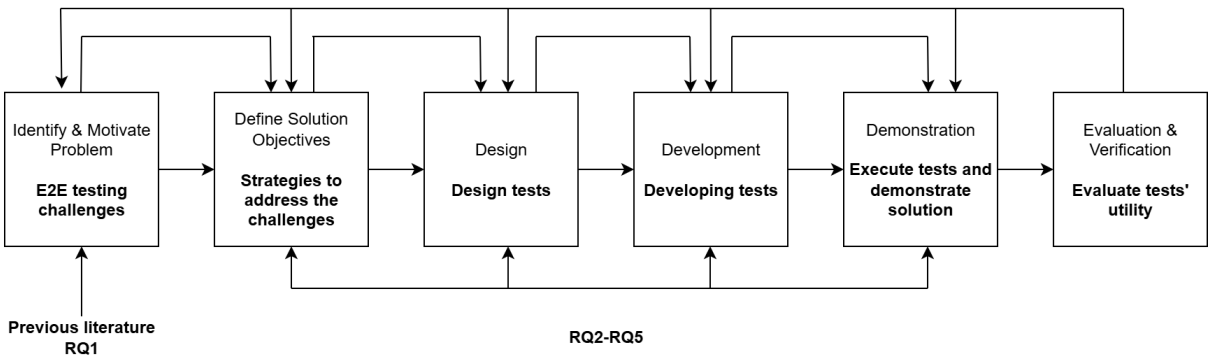

Figure 3.1: Design Science Research process in this thesis, based on [53] and [55].

RQ1 investigated the challenges of E2E testing through a systematic literature review, which identified and categorised challenges reported in academic research. RQ2-RQ5 focused on how specific challenges of E2E testing can be addressed.

The scope of this thesis is limited to challenges in **Group 1: "Test design and implementation"** (Figure 2.4). This group was selected because it provides a clearly scoped set of challenges that can be addressed empirically, and focusing on one group of challenges allows for more in-depth research. The selected group includes the challenges which were mentioned most frequently in the analysed papers.

Group 1 challenges are fragility, maintainability, strong coupling, test dependencies, code duplication, long execution times, and flakiness. These challenges are directly reflected in the research questions RQ2-RQ5. RQ2 focuses on fragility, and RQ3 addresses maintainability and modularity. Modularity in E2E tests targets multiple maintainability-related challenges such as strong coupling, test dependencies and code duplication [31, 35]. These challenges are strongly connected and can be targeted through the same strategies, such as adding an abstraction layer between application code and test code, which is why RQ3 targets multiple challenges. RQ4 focuses on long execution times and RQ5 on flakiness.

The challenges in groups 2 (Web Application Complexity) and 3 (Tooling & Expertise Requirements) are acknowledged but left out of the scope of this research and are suggested for future research.

3.2 Research Context

The research context is an open-source *Vuestic Admin* web application [17]. The Vuestic Admin is an Admin Template application built utilising Vue3, Vite, Pinia, and Tailwind CSS. An open-source backend application, Vuestic Admin Server, serves the API calls sent by Vuestic Admin [16]. In the Vuestic Admin application, users can manage projects and view a dashboard of helpful information. Figure 3.2 displays the Project page of the application where Projects can be viewed as "Cards", and Figure 3.3 displays the same page but with "Table" view. Figure 3.4 displays the view where a user can add or edit a project. This application was selected because it is a full-stack application with relatively complex UI and API calls. As a web application with rich features, it provides an excellent environment for researching E2E testing. The application is licensed under the MIT license, allowing the software to be used, modified, and distributed freely.

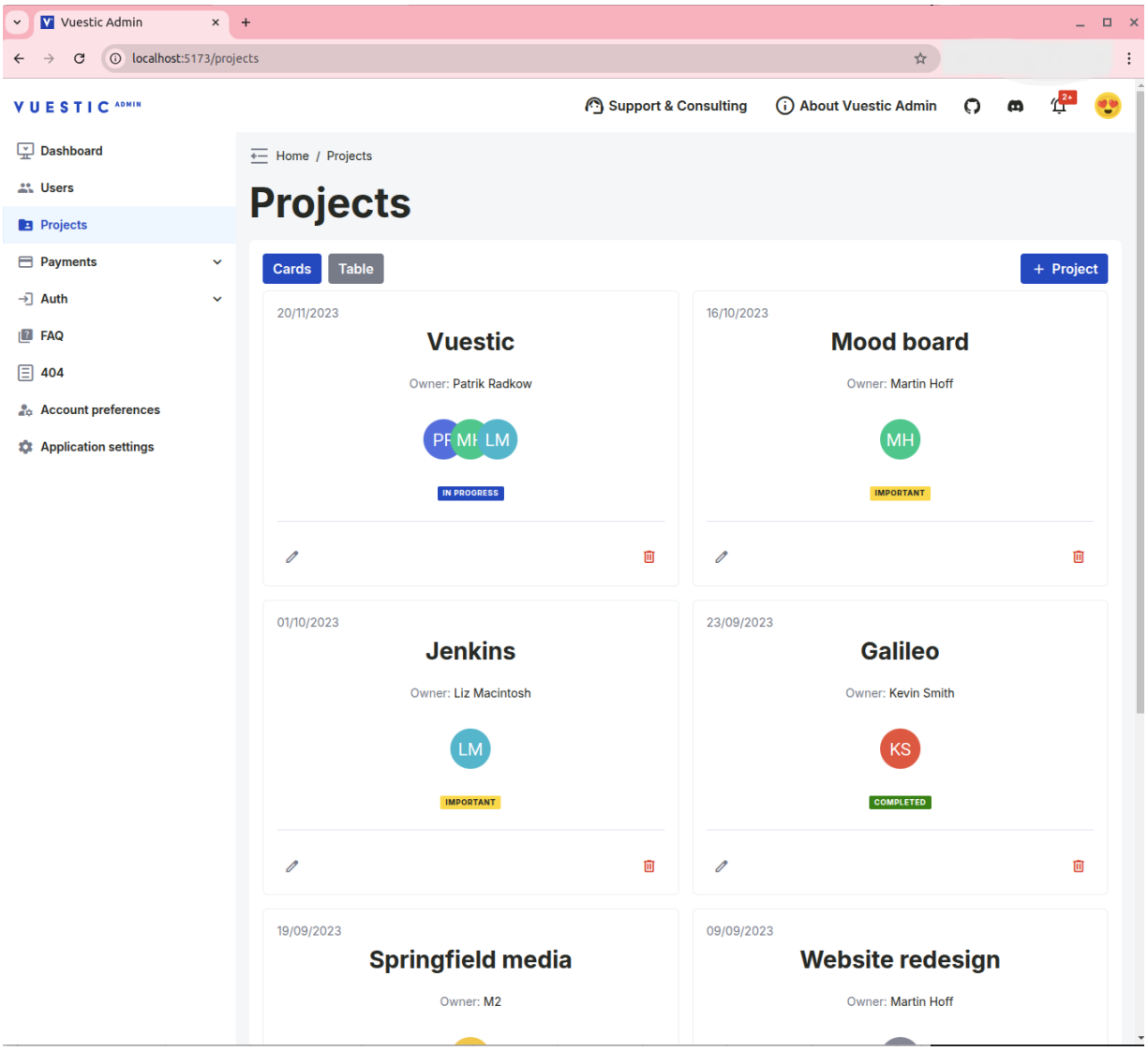

Figure 3.2: Vuestic Admin Application: Projects page, view as 'Cards'

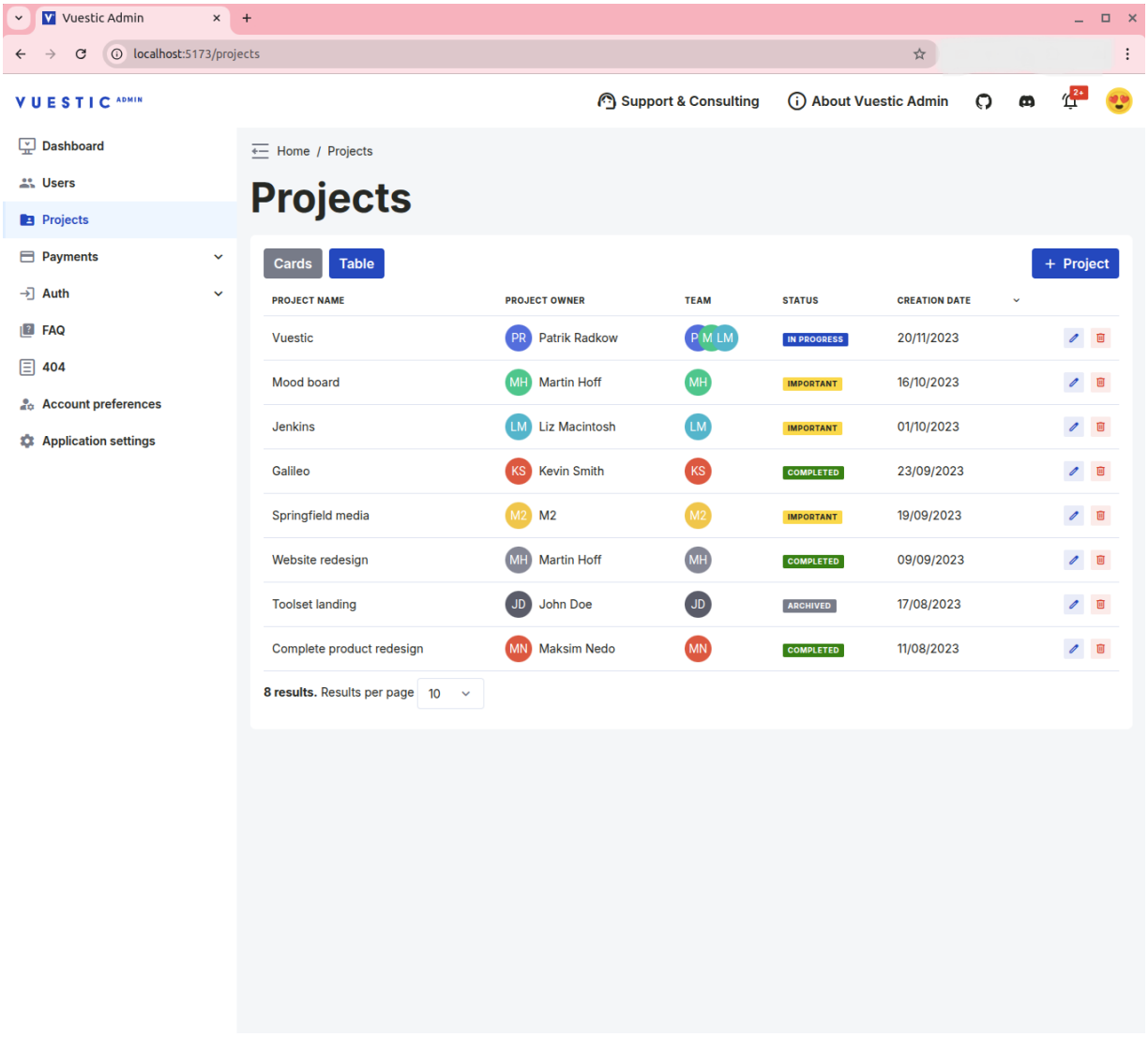

Figure 3.3: Vuestic Admin Application: Projects page, view as 'Table'

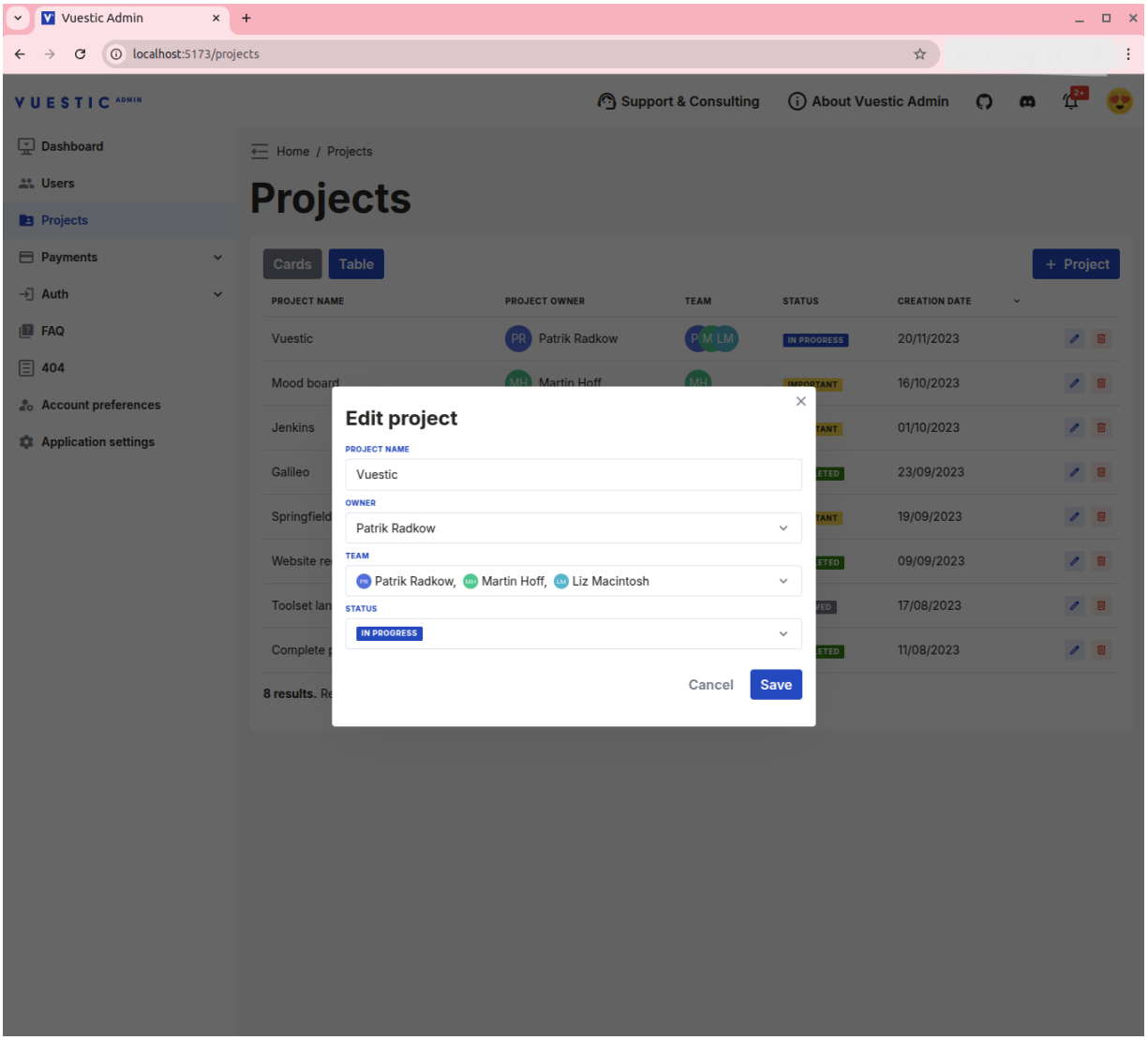

Figure 3.4: Vuestic Admin Application: Edit project view

The tests developed as part of the DSR artefact development were implemented using programmable E2E testing. Programmable E2E tests initially take more time to develop than record and replay tests. However, programmable tests are reported to require significantly less maintenance effort [33]. The test suites were developed using the Cypress testing framework. Cypress is a modern open-source testing framework designed specifically for web applications and built on JavaScript [11]. Another standard and industry de facto tool for writing programmable E2E tests is Selenium WebDriver. Selenium WebDriver is an open-source tool that automates web browsers by providing a programming interface to interact with web pages as a user would [54]. Cypress was selected over Selenium because of its suitability for testing web applications specifically and because it is reported to be faster and more efficient than Selenium [46].

3.3 Research Process

This research followed the Design Science Research (DSR) process (see Figure 3.1). The DSR process consists of the following steps:

1. Identify & Motivate the Problem
2. Define Solution Objectives
3. Design
4. Development
5. Demonstration
6. Evaluation & Verification

In this thesis, the first step was conducted through a Systematic Literature Review (Section 2.3), which answered RQ1: "What are the challenges of E2E testing?".

Three artefact development iterations were completed to build an artefact, an E2E testing suite, to address the challenges. Each iteration focused on one or two research questions and produced E2E test suites, which were then used in demonstration and evaluation.

Table 3.1 provides an overview of all the test suites developed in this research. The test suites were developed to address specific E2E testing challenges identified in the literature and to provide comparison baselines. The test suites were used to empirically

Table 3.1: All of the Test Suites Developed in This Research

ID	Name	Test Suite Style/Type
TS1	Comparison Test Suite	Locating web elements with common but possibly fragile locators
TS2	Data Attribute Locator Test Suite	Locating elements with robust data-cy locators (where possible)
TS3	POM Test Suite	Implementation of Page Object Model, built on the Data-cy Test Suite
TS4	Execution Time Comparison Test Suite	Built on the POM Test Suite, added two more test cases
TS5	Stubbed API Calls Test Suite	Stubbed API calls in all of the test cases

Table 3.2: Test Suite Development Process

Iteration	RQ's	Test Suites Developed	Test Suites Used in Demonstration
1	RQ2	TS1, TS2	TS1, TS2
2	RQ3	TS3	TS2, TS3
3	RQ4, RQ5	TS4	TS3, TS4

test different approaches for improving E2E testing. Table 3.2 summarises the iterative process of developing the test suites. In the first iteration, two test suites, the Comparison Test Suite (TS1) and the Data Attribute Locator Test Suite (TS2) was developed and used for the demonstration. In the second and third development iterations, one new test suite was added in each iteration: POM Test Suite (TS3) and Execution Time Comparison Test Suite (TS4), building incrementally on the previously developed test suites. GitHub Copilot, an AI-powered code completion and chat tool [24], was used to assist in developing the test cases presented in this study.

The process of developing the test suites is described in detail in the following subchapters 3.3.1-3.3.4. Each subsection details how the DSR process was followed from defining solution objectives, through design and development to the demonstration of the solution. The final step, Evaluation & Verification, is presented in the Discussion (Section 5).

Figure 3.5: Test Cases in Test Suite 1-3

1. **X.1:** Successfully opens projects as cards
   (a) Click "Cards"
   (b) Check that the first card contains the project name "Vuestic"
2. **X.2:** Successfully opens a table of projects
   (a) Click "Table"
   (b) Check that the page contains the text "8 results"
3. **X.3:** Contains a project in the table
   (a) Click "Table"
   (b) Check that the first row's "Project name" column contains "Vuestic"
   (c) Check that the first row's "Creation date" column contains "20/11/2023"
   (d) Check that the first row's actions column contains a clickable element
4. **X.4:** Allows a project to be edited
   (a) Click "Table"
   (b) Find the first project name and click the edit button on that row
   (c) Replace Project Name text with "Vuestic!!!"
   (d) Save changes
   (e) Check that the table of projects contains the edited project name

Table 3.3: Locator types in Test Suite 1 and Test Suite 2

Test Case	Test Suite	Locators
X.1	TS 1	text, CSS attribute selector
X.1	TS 2	data-cy
X.2	TS 1	text
X.2	TS 2	data-cy
X.3	TS 1	text, XPath
X.3	TS 2	data-cy
X.4	TS 1	text, HTML tag selector, CSS attribute selector, CSS class selector
X.4	TS 2	data-cy, aria-label

3.3.1 RQ2: Addressing Fragility

Define Solution Objectives

This first Design Science Research (DSR) iteration addressed RQ2: "How can the fragility of E2E tests be mitigated?". Fragility is one of the most significant challenges in E2E testing. Academic literature identifies fragile locators as the most common cause for E2E test fragility; one study found that fragile locators caused 73.62% of test breakages [25]. Therefore, this research proposes using robust locators as a solution to mitigate fragility.

Design and Development

To implement robust locators to mitigate E2E test fragility, data attribute locators were used. Data attribute locators are HTML data-* attributes that are added to the application code. Using data-* attributes is a best locator practice recommended by the official Cypress documentation, as they are not affected by changes in the application code [10]. Two test suites were designed and developed to assess the impact of robust locators. Test Suite 1 (TS1), the **Comparison Test Suite**, includes baseline "control group" tests that use familiar but potentially fragile locators, such as text content, CSS attribute selectors, and XPath selectors. Test Suite 2 (TS2), the **Data Attribute Locator Test Suite**, has the same test cases as TS1, but it locates elements using `data-cy` attributes. One exception in TS2 is test case 2.4, which uses `aria-label` as a locator due to limitations in a Vuestic UI component, `VaInput`. Table 3.3 lists what locators were used for each test

```
1     it('1.1. successfully opens projects as cards', () => {
2         cy.contains('Cards').click()
3         cy.get('[class="va-h4 text-center self-stretch overflow-hidden
          ↪  line-clamp-2 text-ellipsis"]')
4           .first()
5           .should('contain', 'Vuestic')
6     })
7
```

Figure 3.6: Test code for test case 1.1

```
1     <VaButton
2         // added the row below
3         data-cy="save-project-button"
4         click="validate() && $emit('save', newProject as Project)">
5     {{saveButtonLabel}}
6
```

Figure 3.7: Button element code after adding data-cy attribute

case in both TS1 and TS2. Both test suites include four test cases listed in the Figure 3.5. A test code for the test case 1.1 is presented in Figure 3.6. Figure 3.7 presents the code for a button element, when `data-cy` attribute has been added to it.

Demonstration

The test suites' ability to handle changes in the application under test was demonstrated through implementing a set of changes to the user interface (UI). The changes were implemented in the source code of the Vuestic Admin Application. The changes followed the UI change classification model by De Luca et al. [13]. According to the model, changes in web applications can affect Tags, Tag Attributes, Tag Attribute Value, or Text. Each of these could be changed through Removal, Modification, Insertion, and Position Change. Implemented changes include text content, HTML tag position, and attribute name changes. Table 3.4 presents all of the changes made to the Vuestic Admin Application UI. Code of

Table 3.4: Changes made to the Vuestic Admin application, adapted from [13]

ID	Change type	Change
C1	Text content modification	"Table" -> "List"
C2	Text content removal	Removed text "Cards" from a button
C3	Tag movement	Swap the positions of the first and second columns
C4	Tag removal	Remove first column of the table
C5	Tag insertion	Add a new column to the table
C6	Attribute value change	Change edit button 'type="button"' to 'type=submit'
C7	Attribute name change	Change project name class attribute name "classs"
C8	Attribute removal	Remove project name header class attribute
C9	data-cy removal	Remove 'table-option' data-cy attribute from Project row
C10	Position change / Added functionality	Add a new input field to the project edit form

the developed test suites and the implemented changes can be found in the thesis' GitHub repository [18].

Comparison Test Suite (TS1) and Data Attribute Locator Test Suite (TS2) were run against a state of the application with the changes described in Table 3.4. The results of the test runs can be found in Table 4.1 and they are described in Section 4.1.

3.3.2 RQ3: Improving Maintainability and Modularity

Define Solution Objectives

This second DSR iteration addressed RQ3: "How can the maintainability and modularity of E2E tests be improved?". The goal was to improve the maintainability of test suites, even in cases where test breakages occur, despite using robust locators implemented in the first iteration. Besides the breakages caused by fragility, maintainability challenges are often caused by code duplication, strong coupling, and test dependencies, all of which are challenges from Group 1: Test Design and Implementation Challenges (Figure 2.4). Maintainability, code duplication, strong coupling, and test dependencies can be addressed

through improving test code modularity [35, 63]. Page Object Design pattern was selected as a solution to improve modularity, because it is a widely recommended approach in the academic literature to improve maintainability by reducing code duplication and coupling (for example [31, 63]).

Design and Development

In this iteration, a new Test Suite **POM Test Suite (TS3)** was developed using the Page Object Design pattern. The new test suite was built using the **Data Attribute Locator Test Suite** (TS2) developed in the previous section as a starting point. TS3 development began by developing the Page Object for the "Projects" page, following the guidelines in [43]. Figure 3.8 presents the Page Object code. For example, the Page Object class contains methods to click the "Table" button and locate the "Project name" element. Page Object methods are then used in the test code to get elements and perform user actions. An example code for test case 3.1 when using POM is presented in Figure 3.9.

Demonstration

A change to the application code was implemented to demonstrate the differences in maintenance effort between using Page Objects and not using them. All `data-cy` locators were centralised in a single file, and their naming convention was updated to a consistent format (e.g., "ProjectsPage-cards-option") to avoid duplicates as the test suite grows. The code changes are available in the GitHub repository [18]. This change caused all test cases to break in both the **Data Attribute Locator Test Suite** (TS2) and the **POM Test Suite** (TS3). The impact of Page Object Model on test maintainability is presented in Section 4.2.

```
export class ProjectsPage {
  clickCardsButton() {
    return cy.getByLocator('cards-option').click()
  }

  getCardProjectName() {
    return cy.getByLocator('card-project-name')
  }

  clickTableButton() {
    return cy.getByLocator('table-option').click()
  }

  getTableProjectName() {
    return cy.getByLocator('project-name')
  }

  getTableProjectCreationDate() {
    return cy.getByLocator('project-creation-date')
  }

  clickFirstEditProjectButton() {
    return cy.getByLocator('edit-project-button').first().click()
  }

  getProjectNameInput() {
    return cy.getByAriaLabel('$t:inputField')
  }

  clickSaveProjectButton() {
    return cy.getByLocator('save-project-button').click()
  }

  getProjectRow() {
    return cy.getByLocator('project-row-0')
  }
}

export const projectsPage = new ProjectsPage()
```

Figure 3.8: A Page Object representing the Projects page

```
1     const projectsPage = new ProjectsPage()
2
3     it('3.1. successfully opens projects as cards', () => {
4         projectsPage.clickCardsButton()
5         projectsPage.getCardProjectName().should('contain', 'Vuestic')
6     })
7
```

Figure 3.9: POM Test Suite (TS3), Test Case 3.1.

3.3.3 RQ4: Improving Execution Times

Define Solution Objectives

This third DSR iteration addressed RQ4: "How can the execution times of E2E tests be reduced?". E2E test suites generally require significantly longer execution times compared to unit and integration tests. These long execution times are commonly caused by waiting for server responses, initialising browser instances, and rendering complex user interfaces [35]. In continuous delivery environments, long E2E test execution times can become a bottleneck in the deployment pipeline [58]. This iteration aimed to optimise the test suite execution times while maintaining the advantages of comprehensive E2E testing. The objective was to explore stubbing API calls and implications for the execution times and other aspects of the test suites.

The selected approach implemented stubbing of the API calls to optimise execution times. API stubbing means that while executing tests, API calls are intercepted by the test code or framework, and then predefined data is returned (for example, from a file) without waiting for the API call to execute. This solution was selected because one of the primary contributors to long execution times is slow API responses. API responses themselves add to the waiting time in some test cases, and they are often also responsible for providing data for dynamic content updates [35, 51].

Design and Development

Two additional test cases were added to extend the test suites from four to six test cases. This was made to ensure a more complete set of test cases exists for the execution time comparison. The **Execution Time Comparison Test Suite (TS4)** was developed by extending the previously developed **POM Test Suite (TS3)**, which uses `data-cy` locators and the Page Object Model design pattern, and by adding the two new test cases. Then, a new test suite, **Stubbed API Calls Test Suite (TS5)**, was developed to implement API stubbing. Both test suites, TS4 and TS5, contain six test cases. The two new test cases are presented in Figure 3.10.

The following stubbing strategies were implemented in **Stubbed API Calls Test Suite (TS5)**:

1. **Stubbing before each test execution:** Originally, in the **Execution Time Comparison Test Suite (TS4)**, each test case sent an API call to retrieve project data from the server before test execution. In **Stubbed API Calls Test Suite (TS5)**, this process was optimised by replacing API calls with a fixture (JSON file) containing predefined project data. The API request was intercepted, and the fixture data was returned instead.

2. **Test Case 5.4 (Updating a Project):** Originally in TS4, this test case sent a PUT request with updated project data to the API and then retrieved updated data from the API for the projects list with a GET call. These PUT and GET requests were stubbed in TS5.

3. **Test Case 5.6. (Deleting a Project):** Similar to Test Case 5.4, in this test case in the TS4, DELETE and GET API calls were sent to the server to first delete a project and then retrieve an updated list of projects. These DELETE and GET API calls were stubbed in TS5.

Figure 3.10: Added Test Cases in Execution Time Comparison Test Suite (TS4) and Stubbed API Calls Test Suite (TS5)

1. **X.5:** Contains correct information in all table cells
   (a) Click 'Table'
   (b) Find the second project in the table
   (c) Check that all of the cells for that project row contain correct values: Project Name, Project Owner Name, Team Avatar Label, Project Status, Project Creation Date
   (d) Check that the edit and delete buttons for that project exist
2. **X.6:** Allows a project to be deleted
   (a) Click 'Table'
   (b) Get the first project in the table
   (c) Click the delete button
   (d) Click ok on the modal confirmation
   (e) Check that the deleted project does not exist in the table

Demonstration

A script was developed to execute the test suites 100 times and record their execution times. The backend and frontend were hosted locally, and Cypress tests were run in the same environment, on the local machine. The results are presented in Section 4.3.

3.3.4 RQ5: Addressing Flakiness

Define Solution Objectives

The last research question to be addressed was RQ5: "What strategies can reduce flakiness in E2E tests?". Flakiness in E2E test suites mean that the test runs fail unpredictably. The non-deterministic nature of web applications often causes flakiness. The execution order and timing of events may vary between each test execution. This flakiness can be addressed using explicit rather than implicit waits, such as fixed-time thread sleeps.

Explicit waits mean that the test waits for specific states or conditions in the application before proceeding [40].

Explicit waits are automatically handled in the Cypress E2E testing framework. Cypress has built-in functionality that allows tests to wait for content to become visible. For example, in Cypress code `cy.contains('Table')`, if the text content is not found on the first attempt, Cypress automatically retries until it becomes visible. If the content does not become visible before a timeout is reached, the test fails [12]. Selecting Cypress as the testing framework was the selected solution to address flakiness when planning and implementing the test suites in this research. Since Cypress has built-in waiting and retrying, it automatically reduces waiting-based flakiness.

Design and Development

Since the assumption was that using the Cypress Testing Framework and following test development best practices is sufficient to address flakiness, a separate test suite to address this issue was not developed. The **Stubbed API Calls Test Suite (TS5)** was used to test its performance from a flakiness perspective.

Demonstration

A script was developed to execute the test suites 100 times to see whether there were any flakiness errors present. The backend and frontend were hosted locally, and Cypress tests were run in the same environment, on the local machine. The results are presented in Section 4.4.

4 Results

This chapter presents the results of implementing selected solutions to mitigate fragility, improve maintainability, reduce execution times, and address flakiness in end-to-end (E2E) tests. Each section presents results for research questions RQ2-RQ5 (RQ1 results are presented in Section 2.3). Section 4.1 reports the results of addressing fragility by utilising more robust locators. Section 4.2 presents the results of adapting the Page Object Model to improve maintainability and modularity. Section 4.3 reports the impact of API stubbing to improve test execution times, and Section 4.4 presents the results of using the Cypress E2E testing framework to address flakiness.

4.1 Results of Addressing Fragility

This section presents the results of the first design iteration, which addressed RQ2: "How can the fragility of E2E tests be mitigated?". Two test suites, **Comparison Test Suite (TS1)** and **Data Attribute Locator Test Suite (TS2)**, were executed against the Vuestic Admin application. TS1 used common but potentially fragile locators while TS2 used robust `data-cy` locators. A set of changes to the Vuestic Admin application UI was implemented to test how the test suites handle changes in the application under test.

The results are presented in Table 4.1. Overall, **Data Attribute Locator Test Suite (TS2)** was more robust than **Comparison Test Suite (TS1)** when the application under test changed. Each test suite executed four test cases against ten different UI changes, resulting in 40 executions per suite. TS1 produced **14 failures** and TS2 produced only **6 failures**. The use of `data-cy`-locators in TS2 made the tests mostly unaffected by changes in the UI elements. In the TS2, failures were caused by removed elements or removed data-cy locators. Section 4.1.1 describes the results for each UI change in detail.

4.1.1 Results for each implemented UI change

C1: Text Content Modification

This change changed the text on a button in the Application Under Test (AUT) from "Table" to "List". The button in question in the original AUT can be seen in Figure 3.3. As a result of the renaming, Comparison Test Suite (TS1) test cases 1.2, 1.3, and 1.4 failed because they used the button's text to locate the correct button. In this use case, the end-user clicks the buttons to change the view of the project listing. Changing the button's text will unlikely affect the user's ability to perform that action. Therefore, a test that aims to verify the functionality of changing the view (and not the button's exact text) should not fail due to this modification. The `data-cy`-approach in Data Attribute Locator Test Suite (TS2) ensured that the tests remained stable as long as the button was visible on the web page and functional from the user's perspective.

C2: Text Content Removal

Here, the text "Cards" was removed entirely from the button. Text removal led to a failure in test case 1.1, as it depended on the "Cards" text to find the button. Test case 2.1 in TS2 was unaffected because it located the button with the `data-cy` attribute. Tests in both suites (X.2-X.4) that did not reference this button continued to function correctly regardless of the change.

C3: Tag Movement

In the AUT, projects can be displayed in a table view (Figure 3.3). Test cases X.2-X.3 focus on functionality related to that table. When the first and second columns of that table were swapped, test cases 1.3 and 1.4 in TS1 failed. These tests used XPath (1.3) and the first table row tag (1.4) as locators, which were expected to find columns in a specific order. However, tests 2.3 and 2.4 in TS2 passed despite the changes because they used the `data-cy` attribute to locate the needed columns.

C4: Tag Removal

In this change, the first column of the "Projects" table was removed entirely. Test cases 1.3. and 1.4. in TS1 failed because they could not locate the desired cell in the missing

column. In this case, using the `data-cy` attributes did not prevent the tests from breaking. Since the tests 2.3. and 2.4. in TS2 expect a "Project Name" field to be visible on the web page, they fail when it is removed.

C5: Tag Insertion

Adding a new column to the "Projects" table similarly caused test cases 1.3 and 1.4 to fail because they expected columns to be in a particular order. However, test cases 2.3 and 2.4 passed because they used the `data-cy` attribute to locate the correct column successfully.

C6: Attribute Value Change

This change changed the attribute value by changing the button type from "button" to "submit". Test case 1.4, which used a locator $button[type = "button"]$ to locate the "edit" button on the web page, failed due to this change. The corresponding test case 2.4. in TS2 passed because it used the `data-cy` attribute to locate the "edit"-button.

C7: Attribute Name Change

The class attribute name was changed from "class" to "class**s**". This caused test case 1.1 to fail since it used the class attribute to locate the element. Test case 2.1, which used the `data-cy` attribute, was again unaffected by the changes.

C8: Attribute Removal

This change removed the project name header class. This caused test case 1.1 to fail because it depended on the class attribute to locate the correct element. As in previous cases, test case 2.1 was unaffected by this change.

C9: data-cy Removal

This change simulated an accidental or intentional removal of a `data-cy` attribute. In this case, test cases 1.1-1.4 remained unaffected. Tests 2.2-2.4 in the TS2 failed because they relied on the removed `data-cy`.

C10: Functionality Addition

Lastly, adding a new input field to the Project edit form (Figure 3.4) caused test cases 1.4 and 2.4 to fail. The test case 2.4 contains the only element in the TS2 Data Attribute Locator Test Suite that was located using `aria-label` instead of a `data-cy` attribute due to constraints from the Vuestic UI library `VaInput` element. Both tests failed since both test cases 1.4 and 2.4 located the element with the first match to the input `aria-label`, depending on the relative position of the element, which was changed by the field addition.

Table 4.1: Test Results for Comparison Test Suite (TS1) and Data Attribute Locator Test Suite (TS2)

Change	**Test Case**	**Comparison Test Suite (TS1) Result**	**Data Attribute Locator Test Suite (TS2) Result**
No Changes	X.1	Passed	Passed
	X.2	Passed	Passed
	X.3	Passed	Passed
	X.4	Passed	Passed
C1: text content modification	X.1	Passed	Passed
	X.2	Failed	Passed
	X.3	Failed	Passed
	X.4	Failed	Passed
C2: text content removal	X.1	Failed	Passed
	X.2	Passed	Passed
	X.3	Passed	Passed
	X.4	Passed	Passed
C3: tag movement	X.1	Passed	Passed
	X.2	Passed	Passed
	X.3	Failed	Passed
	X.4	Failed	Passed
C4: tag removal	X.1	Passed	Passed
	X.2	Passed	Passed
	X.3	Failed	Failed
	X.4	Failed	Failed

Continued on the next page

Change	**Test Case**	**Comparison Test Suite (TS1) Result**	**Data Attribute Locator Test Suite (TS2) Result**
C5: tag insertion	X.1	Passed	Passed
	X.2	Passed	Passed
	X.3	Failed	Passed
	X.4	Failed	Passed
C6: attribute value change	X.1	Passed	Passed
	X.2	Passed	Passed
	X.3	Passed	Passed
	X.4	Failed	Passed
C7: attribute name change	X.1	Failed	Passed
	X.2	Passed	Passed
	X.3	Passed	Passed
	X.4	Passed	Passed
C8: attribute removal	X.1	Failed	Passed
	X.2	Passed	Passed
	X.3	Passed	Passed
	X.4	Passed	Passed
C9: data-cy removal	X.1	Passed	Passed
	X.2	Passed	Failed
	X.3	Passed	Failed
	X.4	Passed	Failed
C10: added functionality	X.1	Passed	Passed
	X.2	Passed	Passed
	X.3	Passed	Passed
	X.4	Failed	Failed

Test Suite	POM	Lines of Code Updated	# of Files Updated
Data Attribute Locator Test Suite (TS2)	no	14	4
POM Test Suite (TS3)	yes	10	1

Table 4.2: Number of Code Lines Updated to Fix Test Suites

4.2 Results of Improving Maintainability and Modularity

This section presents the results of the second iteration addressing RQ3: "How can the maintainability and modularity of E2E tests be improved?". The **POM Test Suite** (TS3) was developed, and the previously developed **Data Attribute Locator Test Suite** (TS2) was used when demonstrating how Page Object Model (POM) improved maintainability. For demonstration purposes, a code change was implemented which caused all test cases to break in both test suites.

The results are summarised in Table 4.2. In **Data Attribute Locator Test Suite** (TS2), fixing the tests required changing 14 lines of test code across four files. In the **POM Test Suite** (TS3), no changes were needed in the test code itself. Only the Page Object code was updated, with 10 lines modified in one file. Even in a small test suite like the one in this study, the POM improves maintenance effort by reducing the number of lines that need to be edited and centralising the changes to one file. Using POM also improves the modularity of the test code by adding a layer of abstraction between the test code and the application implementation code. Using the POM to increase modularity in the test code base addresses the challenges of test code duplication and strong coupling.

4.3 Results of Improving Execution Times

This section presents the results of the third iteration, which addressed RQ4: “How can the execution times of E2E tests be reduced?”. Two test suites were developed: **Execution Time Comparison Test Suite** (TS4) and **Stubbed API Calls Test Suite** (TS5). Both test suites were executed 100 times, and execution times were recorded. The backend and frontend were hosted locally, and Cypress tests were run in the same environment, on

the local machine.

Table 4.3: Summary of Test Execution Status and Result, Execution Time Comparison Test Suite (TS4) ORIGINAL

Metric	Value
Total execution time (successful runs)	1942.856
Average execution time (successful runs)	19.625
Passed runs	99
Failed runs	1

Table 4.4: Summary of Test Execution Status and Result, Stubbed API Calls Test Suite (TS5) ORIGINAL

Metric	Value
Total execution time (successful runs)	1954.688
Average execution time (successful runs)	19.946
Passed runs	98
Failed runs	2

Initial Test Results

- **Test Suite 4:** Average execution time of **19.63 seconds**, with a total execution time of **1942.86 seconds.**
- **Test Suite 5:** Average execution time of **19.95 seconds**, with total execution time of **1954.69 seconds.**

The initial results are displayed in Tables 4.3 and 4.4. When both the backend and frontend were hosted locally, network delays in API calls did not significantly impact Cypress E2E test performance. TS5 even recorded a slightly longer execution time. A more noticeable difference would likely emerge if the backend and frontend were deployed in a remote environment and the tests were executed within a CI/CD pipeline. To simulate this scenario in a local environment, a 500 ms simulated delay was introduced into the API calls at the backend. Figure 4.1 displays the original backend code, and Figure 4.2 displays the additions made to simulate the network latency.

Table 4.5: Summary of Test Execution Status and Result, Execution Time Comparison Test Suite (TS4) SIMULATED

Metric	**Value**
Total execution time (successful runs)	2602.544
Average execution time (successful runs)	27.110
Passed runs	96
Failed runs	4

Table 4.6: Summary of Test Execution Status and Result, Stubbed API Calls Test Suite (TS5) SIMULATED

Metric	**Value**
Total execution time (successful runs)	2260.330
Average execution time (successful runs)	23.065
Passed runs	98
Failed runs	2

Test results with Simulated Delay

In this phase, the test suites were executed against a backend with a simulated 500 ms network delay.

- **Test Suite 4:** Average execution time of **27.11 seconds**, with a total execution time of **2,602.54 seconds.**
- **Test Suite 5:** Average execution time of **23.07 seconds**, with total execution time of **2,260.33 seconds.**

The results are displayed in Tables 4.5 and 4.6. With the simulated network delay, Test Suite 5 outperformed Test Suite 4, demonstrating the benefits of the implemented optimisation.

```
const express = require("express");
const router = express.Router();
const userRoutes = require("./users");
const projectRoutes = require("./projects");
const rootRoutes = require("./root");

router.use("/users", userRoutes);
router.use("/projects", projectRoutes);
router.use("/", rootRoutes);

module.exports = router;
```

Figure 4.1: Vuestic Admin Server original backend code

```
const express = require("express");
const router = express.Router();
const userRoutes = require("./users");
const projectRoutes = require("./projects");
const rootRoutes = require("./root");

// Add delay middleware to simulate network latency
const addDelay = (req, res, next) => {
        setTimeout(() => {
                next();
        }, 500); // 500ms delay
};

// Apply the delay middleware to all routes
router.use(addDelay);

router.use("/users", userRoutes);
router.use("/projects", projectRoutes);
router.use("/", rootRoutes);

module.exports = router;
```

Figure 4.2: Vuestic Admin Server backend code with simulated network latency

4.4 Results of Addressing Flakiness

This section presents the results related to RQ5: "How can flakiness in E2E tests be reduced?". No new test suite was developed. The selected solution to address flakiness was to select the Cypress Testing Framework as the tool to develop tests with. The solution was tested by running the **Stubbed API Calls Test Suite** (TS5) 100 times and checking if flakiness occurred. Since Cypress has built-in mechanisms to tackle flakiness, the expectation was that the test suite would not have any flakiness.

However, the first execution of **Stubbed API Calls Test Suite** (TS5) revealed flakiness in the test suite, and 50 % of the 100 test runs failed. The failure was caused by `ResizeObserver loop completed with undelivered notification - error`. The error wasn't caused by a functional bug, and it was added to Cypress's ignore list so that the testing could proceed. After explicitly ignoring the error, results were better: 98/100 runs succeeded. The remaining two failures were `A callback was provided to intercept the upstream response, but a network error occurred while making the request` -errors.

The results are summarised in Table 4.4. While Cypress's retry functionality and the suppression of irrelevant errors reduced flakiness, some network-related flakiness remained. Addressing the flakiness by relying on Cypress's built-in functionality was not a successful strategy.

5 Discussion

This Chapter presents answers to the research questions, interprets their implications, and links the findings to previous research. Sections 5.1–5.5 discuss RQ1–RQ5: identifying challenges, mitigation of fragility, improving maintainability, reducing execution times, and addressing flakiness. Section 5.6 discusses the limitations of the study and suggests directions for future research.

5.1 RQ1: What are the challenges of E2E testing?

This study conducted a Systematic Literature Review (SLR), analysing 53 papers that discussed the challenges of end-to-end testing (E2E) of web applications. From these papers, 15 distinct challenges were identified. Maintainability and fragility have been highlighted in prior studies as the most significant challenges of E2E testing [35, 48]. The findings in this study are consistent with prior work, as maintainability and fragility were the most frequently reported challenges, each mentioned 27 times in the analysed articles. Other frequently mentioned challenges were difficulty in test creation (17 mentions) and long execution times (15 mentions).

Previous research on E2E testing challenges has often provided lists of challenges without categorising them into groups. For example, one study identified three open challenges: fragility, strong coupling and low cohesion [57]. Another study highlighted the challenges of scalability, lack of oracles, and test reporting [14]. A third study listed four challenges: difficulty in translating requirements into tests, tests breaking often, uncertainty about when to automate tests, and the complexity of the code base [58]. Similarly, a study surveying professionals using Selenium reported 13 different challenges, but did not categorise them into groups [35].

In contrast, this study categorised the 15 identified challenges into three groups, illustrated in Figure 2.4. The three groups are the **Test Design & Implementation** challenges, the **Web Application Complexity** challenges, and the **Tooling & Expertise Requirements** challenges. This categorisation groups related challenges together and highlights their shared underlying causes. By doing so, it provides a higher-level perspective that helps in understanding both the nature of the challenges and possible solutions.

A similar categorisation to the one in this thesis was done by Nass et al., who identified three groups of challenges in E2E testing: test execution fragility, tools and automation skills, and model-based testing challenges [48]. Their tools and automation skills category corresponds to this study's Tooling & Expertise Requirements category. Fragility and model-based testing challenges, on the other hand, are included here under the broader group of challenges related to test design and implementation.

5.2 RQ2: How can the fragility of E2E tests be mitigated?

To address the fragility of E2E tests, the Data Attribute Locator Test Suite (TS2), which used `data-cy` locators, was developed. Comparison Test Suite (TS1) was developed to use when demonstrating the effectiveness of more robust locators. Table 4.1 presents the results of the test runs for both suites. The table clearly shows that `data-cy` locators are more robust to changes than other locators: there were 14 failures when not using `data-cy`, and only 6 failures when using them. When `data-cy` attributes are injected into application code, the test cases are more resilient to modification in text content and the relative position of elements in the DOM. The results highlight robust locators as a successful mitigation for fragility. The remainder of this section is structured as follows: subsection 5.2.1 interprets and discusses the results, and subsection 5.2.2 compares the results with previous literature.

5.2.1 Robust Locators in Practice

This subsection interprets and discusses the results of improving locator robustness and their implications for practice. First, the distinction between test failures and test breakages is presented, then the robust locator strategy's relation to the testing pyramid is discussed and finally, trade-offs of the selected locator strategy are discussed.

Test Failures vs. Test Breakages

In one of the studied changes to the application under test (AUT), the removal of the project name column caused multiple tests to fail. This is an example of a "True Positive," where the test failure correctly reflects a real change in the application's user-facing

functionality. In a case where the Project Name field is removed from the application, the test should either fail to alert about a potential bug or be updated to match the new behaviour of the application. This highlights an important distinction: test failures indicate a real issue in the application, while test breakages happen when tests fail even though the functionality has not changed [60]. In the implemented changes to the AUT, removing a locator attribute led to failures even though the user-facing behaviour had not changed. This is an example of test breakage rather than test failure.

Testing Pyramid and the Use of data-cy Locators

When aiming to improve locator robustness in E2E tests, it is essential to consider what should be tested at the E2E level. Figure 2.2 displays the Testing Pyramid, which introduces the idea that the lowest-level, fast unit tests should be the most comprehensive testing. Unit tests are better suited for low-level checks, such as specific text content or the order of elements. E2E tests, on the other hand, should focus on verifying that the user can complete the most critical user paths and that the system works from the user's perspective. For example, if a test fails because the label on a button changes from "Table" to "List," even though the button still performs the correct function, this indicates a fragility issue. In such a case, the presence or exact wording of the text could be tested in lightweight unit tests, while the E2E test should verify if the view can be successfully changed by clicking the button. To support this type of functional testing at the E2E level, locators should not rely on user-facing content, such as visible text or specific positions. Instead, locators like `data-cy` attributes should be used to maintain robust test cases.

Trade-offs of Using data-cy Attributes

The most significant trade-off of not using user-facing locators is that these selectors may not reflect how the interface appears to the end user. A test using only `data-cy` attributes might pass even if the element is visually broken, hidden, or mislabeled. This highlights the importance of combining E2E testing with other levels of testing to ensure both functionality and a correct visual user experience.

Another trade-off is that the initial development time is longer when, in addition to developing the test cases, the developer must also add attributes to all elements under test. In large applications, it can become a problem if multiple locator elements are added manually, since the same `data-cy` might be used multiple times in different elements. De Luca

et al. [13] also highlight that introducing hooks into the DOM adds development overhead, especially when added manually. They propose an automatic injection of hooks as a solution. An additional concern about using data-cy attributes is that they may expose technical details to users, which could pose a security risk. However, the attributes can be stripped away from a production build if needed [44].

5.2.2 Comparison to related work

The results of mitigating fragility are consistent with findings by Hammoudi et al. [25], who investigated test breakages in record and replay E2E tests. Their study introduced a taxonomy of breakage causes, which identified issues related to locators as the most common reason, responsible for 73.62% of failures. They distinguished between breakages caused by changes in attributes (e.g., text modification) and breakages caused by changes in the structure (e.g., changes to the element index). While their research focused on record and replay tests, the locator mechanisms are similar in programmable testing, making the comparison relevant and reasonable. The results of this thesis strongly support their observation, confirming that locators based on attribute values or DOM structure are highly fragile.

Similarly, Di Meglio et al. showed that inserting a new input field can break tests that use locators depending on the DOM structure [15]. De Luca et al. also note that while structure-dependent locators are highly specific and can effectively locate a unique element on the web page, they are extremely fragile when changes happen in the web page [13]. Any change in the element to be located or the elements near it can cause the test using structure-dependent locators to fail. The results in this thesis confirm the observations in previous research: even a small change to the form layout caused multiple tests to fail because the locators in those tests located elements using their relative positions in the DOM structure.

5.3 RQ3: How can the maintainability and modularity of E2E tests be improved?

The previous subsection discussed how using robust locators reduces test breakages, which leads to less time spent on test maintenance. To further improve maintainability, the Page Object Model (POM) can be used. The results in this study show that using POM

reduces maintenance effort by 28.6%. The remainder of this section is structured as follows: subsection 5.3.1 interprets and discusses the results of using POM, and subsection 5.3.2 compares the results with previous literature.

5.3.1 Improved Maintainability in Practice

Improving locator robustness enhances the maintainability of the E2E test suites by reducing test breakages, as observed in the first iteration of artefact development. In the second iteration, the objective was to enhance the E2E tests' maintainability further.

Table 4.2 illustrates that even in a small test suite, such as the **POM Test Suite (TS3)** developed in the second iteration, the number of code lines that needed updates was 28.6% lower when Page Object Model (POM) was used. More importantly, changes required without POM were spread across four different files, while with POM, only the one Page Object class file needed to be modified. This centralised solution reduces maintenance effort and cognitive load when maintaining the tests. The results confirm the assumption that using the Page Object Model improves the maintainability of the test suite.

The Page Object Model improves maintainability by reducing test code duplication and decoupling test code from the application's implementation code. In the developed artefact, the page Object class centralises functionalities, such as clicking the "Table" button, into reusable methods. Before POM implementation, each test case that clicked the button needed to implement the logic to interact with the application. With POM, the interaction is lifted to the Page Object class, and each test case can utilise the same method from there. In the event of test breakage, for example, if the tests can no longer identify the button in question for some reason, only the Page Object class code needs to be updated.

Despite the advantages of using the Page Object Model (POM), there are also some challenges. Firstly, research on using POM in E2E tests has identified that developing comprehensive Page Objects is challenging and adds development overhead [36, 56]. Secondly, strong development skills are required to implement the Page Object design pattern successfully. Thirdly, maintaining the Page Objects themselves can become a maintenance burden, especially as applications grow and Page Objects become too large and complex. However, academic literature strongly supports that the initial investment in Page Objects pays off over the lifecycle of a software project. Studies have shown that when an application grows and evolves, the initial development cost is outweighed by the saved maintenance time [35, 57].

Another important factor contributing to the maintainability of the test suites developed in this research was the choice of programmable testing instead of record and replay testing. Leotta et al. [33] compared record and replay testing and programmable testing, noting that while programmable testing requires more initial development time, it results in test suites that are easier to maintain over time. Here, we can see the same trade-off as with the Page Object Model design pattern: the initial development cost is high, but it pays off in the long term with lower maintenance costs. The longer the applications and the test suites are used, the larger the application grows, and the more updates are made, the greater the benefits of the initial investment become.

5.3.2 Comparison to Related Work

The Page Object model was selected as a solution to improve maintainability, because previous literature has shown that the design patterns and architecture affect maintenance effort [35]. The Page Object design pattern has emerged as the leading pattern for improving maintainability [21, 22, 57, 59]. This study's results align with the previous literature, confirming that despite its initial development overhead, adopting POM improves maintainability.

As Leotta et al. [35] and Yu et al. [63] also highlight, having a simple, abstract access point for the interactions with the application's user interface simplifies maintenance and improves test code reuse. This structure reduces test dependencies when test cases call independent methods from the Page Objects and do not need to rely on the internal details of the web page. The Page Object can also wrap the setup logic of the tests, such as setting up data for the tests, further reducing test dependencies across different test cases.

In addition to using POM, the selection of the programmable testing approach was another way to improve maintainability. Two studies by Stocco et al. [59] and by Leotta et al. [35] have confirmed the same result found in this thesis: maintaining programmable E2E test suites requires significantly less effort than maintaining record and replay test suites. Therefore, the decision to use programmable testing aligned with best practices and supported the goal of improving maintainability.

5.4 RQ4: How can the execution times of E2E tests be reduced??

In this study, the results show that stubbing API calls improves test suite execution times: the mean execution time was 4.04 s faster per test run. The optimisation of execution times in the E2E test suite has direct implications for software development efficiency, especially in agile development environments where test suites are executed frequently as part of a continuous integration and continuous deployment (CI/CD) pipeline. Since E2E tests take significantly longer to execute than unit or integration tests, reducing execution times can lead to significant time savings.

To highlight the impact, let us consider a scenario where a development team maintains 20 test suites, each executed upon every commit to the version control system. If a developer makes three commits per day, the total number of executions per week would be

$$20 \text{ test suites} \times 3 \text{ commits/day} \times 5 \text{ days/week} = \mathbf{300 \quad test\ suite\ executions\ per\ week}$$

Using the measured improvements from this study:

- Without optimisation (TS4): Average execution time 27.11 seconds.
- With optimisation (TS5): Average execution time 23.07 seconds.

The total time saved per execution:

$$27.11 - 23.07 = 4.04 \text{ seconds per test suite}$$

For 300 test suite executions per week, the total time saved per week is:

$$4.04 \times 300 = 1,212 \text{ seconds} \approx \mathbf{20.2 \quad minutes}$$

Extending this to a month (4 weeks):

$$20.2 \times 4 = 80.8 \text{ minutes} \approx \mathbf{1.35 \quad hours\ per\ month}$$

Then, let us consider a larger system with 50 test suites and five commits per day (per developer).

$$50 \times 5 \times 5 = \textbf{1250 test suite executions per week}$$

$$4.04 \times 1250 = 5050 \text{ seconds} \approx \textbf{84.2 minutes per week}$$

$$84.2 \times 4 = 336.8 \text{ minutes} \approx \textbf{5.6 hours per month}$$

This highlights that just a few seconds of optimisation per test suite leads to significant time saved in software development environments, where test suites are executed frequently. Reducing test suite execution times makes the feedback loop faster and increases the availability of computing resources in the CI/CD pipeline.

The most significant trade-off of using API stubbing is that it does not test the backend functionality from start to finish. If all E2E test cases use API stubbing, the database connectivity and backend logic will not be tested. Since API calls are intercepted and stubbed data is returned, the tests no longer verify that the backend is working correctly or that the database can process queries.

Additional testing can mitigate the risk associated with stubbing. For example, a simple test could be executed at the beginning of a test suite to check that the backend and database are working correctly. After this, tests covering more complex application logic can use API stubbing. This strategy aligns well with the Testing Pyramid guidance to keep the number of E2E lower than integration and unit tests [20].

As discussed previously, the API stubbing strategy has limitations. Instead of stubbing API calls, another approach is to prioritise only the most critical user-facing functionalities to be tested in E2E testing. Not every interaction or user flow requires E2E coverage. Lower levels of testing, such as unit testing, should cover larger percentages of the code and functionalities. This perspective also aligns with the testing pyramid model in Figure 2.2, [20]. The testing pyramid model suggests a strong foundation of unit tests, a moderate number of integration tests, and a relatively small number of E2E tests at the top. The number of E2E tests should never exceed the number of integration or unit tests.

According to this thesis author's best knowledge, previous research has not studied the effect and implications of API stubbing for E2E test execution times. Previous work on improving test execution times has proposed parallel execution of tests [51], resource optimisation [4], state reuse [64], and replacing fixed sleeps with explicit waits [40]. The

API stubbing strategy in this study targets network latency. To further improve the test execution times, future research should study combining the different approaches.

5.5 RQ5: What strategies can reduce flakiness in E2E tests?

The primary strategy to address flakiness in this study was the selection of the Cypress testing framework. Cypress has built-in functionality to retry locating elements until they are present, which directly targets the common causes of flakiness, such as dynamic content rendering and waiting for API responses. This led to the assumption that flakiness would not be present in the implemented test suites.

However, when the Execution Time Comparison Test Suite (TS4) and Stubbed API Calls Test Suite (TS5) were executed a hundred times when demonstrating the effectiveness of execution time improvements (related to RQ4), multiple executions revealed flakiness issues (see subsection 4.4). After the error that caused undeterministic failures in the tests was explicitly ignored, two failures still occurred across the 100 runs. Further investigation of these issues was left out of scope for this study, but the findings indicate that the strategy was not entirely effective in practice. The results align with previous literature, which has noted that explicit waits and retries are powerful ways to mitigate flakiness but do not fully eliminate it [50]. The unexpected struggles with flaky errors were a great example of why many practitioners report that flakiness is one of the major challenges of E2E testing [22].

5.6 Limitations and Future Research

This study has limitations that should be considered when considering the results and conclusions. Because the validation relied on manually implemented changes instead of a real-life development project, the main threats to both internal and external validity come from this source. The proposed solutions were not validated in a real-life software product development environment, and the changes were implemented for the purpose of testing solutions. The changes followed a structured classification model, but they might still fail to reflect the complexity of real application development. This affects the internal validity and limits the extent to which the results can be interpreted confidently. Second,

the evaluation was conducted on one open-source application, the test suites were small, and no development team took part in the research. All of these could have affected the utility of the proposed solution, which affects the external validity of this research: the results might not be generalisable beyond the context of this study.

These limitations do not invalidate the results, but they do set some boundaries on how applicable the results are in other contexts. To address these limitations, future research should replicate the solutions of this study in multiple, diverse software projects.

The developed test suites in this research addressed only a subset of E2E challenges, the challenges in Group 1: Test Design and Implementation challenges (see Figure 2.4). Future research should address the remaining challenges in Group 2: Web Application Complexity and Group 3: Tooling & Expertise Requirements. Automatic injection of data attributes, automatic generation of Page Object Models, and automatic fixing of broken locators are interesting points for future research that could improve the solutions proposed in this research.

6 Conclusions

Software testing is an essential process for verifying software functionality and quality. End-to-end (E2E) testing is a type of software testing used to test the entire application workflow from the end user's perspective. In the web applications context, automated E2E tests act as an end user on the application's web user interface, and test that all the application layers, including the user interface, backend, and database, function as expected. While automated E2E testing improves efficiency and software quality, it also faces significant challenges, such as fragility and flakiness

This thesis contributes to a deeper understanding of these challenges and explores practical solutions to address them. The Systematic Literature Review identified 15 key challenges of E2E testing. These challenges were categorised into three high-level categories: challenges related to test design and implementation, challenges related to web application complexity, and challenges connected to tooling and expertise. This categorisation provides a structured view of the many challenges and eases the future work on these challenges.

This thesis focused on finding and empirically testing solutions to challenges in the first group: test design and implementation challenges. The Design Science Research approach was applied in three iterations of artefact development. The results demonstrate that focusing on E2E test design can help address the challenges. Fragility was reduced with robust locators, and maintainability was improved by using the Page Object Model to add a layer of abstraction between the application and test code. Execution times were improved by using API stubbing, and flakiness was reduced, but not eliminated, by using the testing framework's built-in wait and retry mechanisms. The research in this thesis produced an artefact: an end-to-end testing suite where these design choices have been implemented.

This thesis contributes to highlighting for practitioners that investing in test design and implementation can reduce maintenance efforts and pay off in the long run. For researchers, the results highlight the value of combining empirical experimentation with systematic literature review to integrate academic perspectives and practical solutions. The next step is to take these practices into use in the industry and focus future research on solving the remaining challenges of E2E web testing.

Despite its contributions, this thesis also has some limitations. The selected approaches were evaluated under simulated conditions, using only one open-source application. Future research is needed to generalise the findings. Also, only a subset of the identified challenges was addressed. Future research could also include implementing automatic attribute generation, automatic generation of page objects, and automatic repair of broken locators.

In conclusion, this thesis both systematises existing knowledge about E2E testing challenges and demonstrates that many of these challenges can be mitigated through skilled and careful test design and implementation. These contributions support the development of more reliable, maintainable, and efficient E2E test suites for modern web applications.

References

[1] P. Aho and T. Vos. "Challenges in automated testing through graphical user interface". In: *2018 ieee international conference on software testing, verification and validation workshops (icstw)*. IEEE. 2018, pp. 118–121.

[2] E. Alégroth, A. Karlsson, and A. Radway. "Continuous integration and visual gui testing: Benefits and drawbacks in industrial practice". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2018, pp. 172–181.

[3] D. Asfaw. "Benefits of automated testing over manual testing". In: *International journal of innovative research in information security*, 2(1) (2015), pp. 5–13.

[4] C. Augusto. "Efficient test execution in End to End testing: Resource optimization in End to End testing through a smart resource characterization and orchestration". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 2020, pp. 152–154.

[5] S. Balaji and M. S. Murugaiyan. "Waterfall vs. V-Model vs. Agile: A comparative study on SDLC". In: *International Journal of Information Technology and Business Management*, 2(1) (2012), pp. 26–30.

[6] E. Battista, S. Di Martino, S. Di Meglio, F. Scippacercola, and L. L. L. Starace. "E2E-Loader: A Framework to Support Performance Testing of Web Applications". In: *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2023, pp. 351–361.

[7] J. E. Bentley, W. Bank, and N. Charlotte. "Software testing fundamentals—concepts, roles, and terminology". In: *Proceedings of SAS Conference*. 2005, pp. 1–12.

[8] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca, and P. Tonella. "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*. 2019, pp. 154–164.

[9] R. Coppola, R. Feldt, M. Nass, and E. Alégroth. "Ranking approaches for similarity-based web element location". In: *Journal of Systems and Software*, 222 (2025).

[10] Cypress.io. *Best Practices - Cypress Documentation*. 2025. URL: https://docs.cypress.io/app/core-concepts/best-practices#How-It-Works (visited on 02/07/2025).

[11] Cypress.io. *Cypress: JavaScript End-to-End Testing Framework*. 2025. URL: https://www.cypress.io/ (visited on 01/28/2025).

[12] Cypress.io. *Waiting and Retry-ability*. 2024. URL: https://learn.cypress.io/cypress-fundamentals/waiting-and-retry-ability (visited on 03/23/2025).

[13] M. De Luca, A. R. Fasolino, and P. Tramontana. "Investigating the robustness of locators in template-based Web application testing using a GUI change classification model". In: *Journal of Systems and Software*, 210 (2024).

[14] G. Denaro, L. Guglielmo, L. Mariani, and O. Riganelli. "GUI testing in production: Challenges and opportunities". In: *Companion proceedings of the 3rd international conference on the art, science, and engineering of programming*. 2019, pp. 1–3.

[15] S. Di Meglio and L. L. L. Starace. "Towards Predicting Fragility in End-to-End Web Tests". In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 2024, pp. 387–392.

[16] Epicmax. *Vuestic Admin Server*. 2025. URL: https://github.com/epicmaxco/vuestic-admin-server (visited on 02/07/2025).

[17] Epicmax. *Vuestic Admin: Free and Beautiful Vue 3 Admin Template*. 2025. URL: https://github.com/epicmaxco/vuestic-admin (visited on 02/07/2025).

[18] Epicmax and N. Rytilä. *copy2-vuestic-admin*. https://github.com/noorary/copy2-vuestic-admin. 2025. (Visited on 03/27/2025).

[19] M. Fowler. *Page Object*. Sept. 10, 2013. URL: http://martinfowler.com/bliki/PageObject.html (visited on 01/17/2025).

[20] M. Fowler. *The Practical Test Pyramid*. 2018. URL: https://martinfowler.com/articles/practical-test-pyramid.html (visited on 03/22/2025).

[21] T. Fulcini, G. Garaccione, R. Coppola, L. Ardito, and M. Torchiano. "Guidelines for GUI testing maintenance: a linter for test smell detection". In: *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*. 2022, pp. 17–24.

[22] B. García, M. Gallego, F. Gortázar, and M. Munoz-Organero. "A survey of the selenium ecosystem". In: *Electronics*, 9(7) (2020).

[23] V. Garousi and M. V. Mäntylä. "When and what to automate in software testing? A multi-vocal literature review". In: *Information and Software Technology*, 76 (2016), pp. 92–117.

[24] GitHub. *GitHub Copilot: Your AI pair programmer*. 2025. URL: https://github.com/features/copilot (visited on 04/25/2025).

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

[26] A. R. Hevner, S. T. March, J. Park, and S. Ram. "Design science in information systems research". In: *MIS quarterly* (2004), pp. 75–105.

[27] I. Hooda and R. S. Chhillar. "Software test process, testing types and techniques". In: *International Journal of Computer Applications*, 111(13) (2015).

[28] H. Kirinuki, H. Tanno, and K. Natsukawa. "COLOR: correct locator recommender for broken test scripts using various clues in web application". In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2019, pp. 310–320.

[29] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. "Systematic literature reviews in software engineering–a systematic literature review". In: *Information and software technology*, 51(1) (2009), pp. 7–15.

[30] C. Kumar and D. K. Yadav. "Software defects estimation using metrics of early phases of software development life cycle". In: *International Journal of System Assurance Engineering and Management*, 8 (2017), pp. 2109–2117.

[31] M. Leotta, M. Biagiola, F. Ricca, M. Ceccato, and P. Tonella. "A family of experiments to assess the impact of page object pattern in web test suite development". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2020, pp. 263–273.

[32] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro. "Improving test suites maintainability with the page object pattern: An industrial case study". In: *2013 IEEE sixth international conference on software testing, verification and validation workshops*. IEEE. 2013, pp. 108–113.

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

[34] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. “Visual vs. DOM-based web locators: An empirical study”. In: *Web Engineering: 14th International Conference, ICWE 2014, Toulouse, France, July 1-4, 2014. Proceedings 14*. Springer. 2014, pp. 322–340.

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

[36] M. Leotta, A. Molinari, and F. Ricca. “Assessor: a po-based webdriver test suites generator from selenium ide recordings”. In: *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2022, pp. 389–399.

[37] M. Leotta, F. Ricca, and P. Tonella. “Sidereal: Statistical adaptive generation of robust locators for web testing”. In: *Software Testing, Verification and Reliability*, 31(3) (2021), e1767.

[38] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. “Reducing web test cases aging by means of robust XPath locators”. In: *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE. 2014, pp. 449–454.

[39] M. Leotta, H. Z. Yousaf, F. Ricca, and B. Garcia. “AI-generated test scripts for web e2e testing with ChatGPT and copilot: a preliminary study”. In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 2024, pp. 339–344.

[40] X. Liu, Z. Song, W. Fang, W. Yang, and W. Wang. “Wefix: Intelligent automatic generation of explicit waits for efficient web end-to-end flaky tests”. In: *Proceedings of the ACM Web Conference 2024*. 2024, pp. 3043–3052.

[41] J. Mahmud, A. Cypher, E. Haber, and T. Lau. “Design and industrial evaluation of a tool supporting semi-automated website testing”. In: *Software Testing, Verification and Reliability*, 24(1) (2014), pp. 61–82.

[42] MDN Web Docs. *Introduction to the Document Object Model (DOM)*. 2025. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction (visited on 01/27/2025).

[43] M. M. Mehdi. *Page Object Model - Cypress*. 2023. URL: https://medium.com/@mmahsanmehdi/page-object-model-cypress-732ff6923e09 (visited on 02/10/2025).

[44] M. Merken. *Angular and Cypress: data-cy attributes*. 2025. URL: https://medium.com/agilix/angular-and-cypress-data-cy-attributes-d698c01df062 (visited on 02/07/2025).

[45] M. Mika. *Mini Systematic Review Guide for Master's Thesis*. 2023. URL: https://mmantyla.github.io/mini-slr-for-thesis/ (visited on 01/07/2025).

[46] F. Mobaraya, S. Ali, et al. "Technical Analysis of Selenium and Cypress as functional automation framework for modern web application testing". In: *9th International Conference on Computer Science*. 2019.

[47] J. Morán Barbón, C. Augusto Alonso, A. Bertolino, C. A. Riva Álvarez, J. F. García Tuya, et al. "Debugging flaky tests on web applications". In: *Proceedings of the 15th International Conference on Web Information Systems and Technologies-Volume 1: APMDWE*. 2019.

[48] M. Nass, E. Alégroth, and R. Feldt. "Why many challenges with GUI test automation (will) remain". In: *Information and Software Technology*, 138 (2021).

[49] K. Ngo, V. Nguyen, and T. Nguyen. "Research on test flakiness: from unit to system testing". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–4.

[50] D. Olianas, M. Leotta, and F. Ricca. "SleepReplacer: a novel tool-based approach for replacing thread sleeps in selenium WebDriver test code". In: *Software Quality Journal*, 30(4) (2022), pp. 1089–1121.

[51] D. Olianas, M. Leotta, F. Ricca, M. Biagiola, and P. Tonella. "Stile: a tool for parallel execution of e2e web test scripts". In: *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2021, pp. 460–465.

[52] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. "A survey of flaky tests". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1) (2021), pp. 1–74.

[53] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. "A design science research methodology for information systems research". In: *Journal of management information systems*, 24(3) (2007), pp. 45–77.

[54] S. Project. *Selenium WebDriver*. Version 4.0. 2025. URL: https://www.selenium.dev/documentation/webdriver/ (visited on 01/28/2025).

[55] M. Raatikainen, Q. Motger, C. M. Lüders, X. Franch, L. Myllyaho, E. Kettunen, J. Marco, J. Tiihonen, M. Halonen, and T. Männistö. “Improved management of issue dependencies in issue trackers of large collaborative projects”. In: *IEEE Transactions on Software Engineering*, 49(4) (2022), pp. 2128–2148.

[56] F. Ricca and M. Leotta. “Towards automated generation of PO-based WebDriver test suites from Selenium IDE recordings”. In: *Proceedings of the 12th International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 2021, pp. 9–16.

[57] F. Ricca, M. Leotta, and A. Stocco. “Three open problems in the context of e2e web testing and a vision: Neonate”. In: *Advances in Computers*. Vol. 113. Elsevier, 2019, pp. 89–133.

[58] R. Rwemalika, M. Kintis, M. Papadakis, and Y. Le Traon. “Can we automate away the main challenges of end-to-end testing?” In: *The 17th Belgium-Netherlands Software Evolution Workshop*. 2018.

[59] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. “APOGEN: automatic page object generator for web testing”. In: *Software Quality Journal*, 25(3) (2017), pp. 1007–1039.

[60] A. Stocco, R. Yandrapally, and A. Mesbah. “Visual web test repair”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 503–514.

[61] S. Thummalapenta, P. Devaki, S. Sinha, S. Chandra, S. Gnanasundaram, D. D. Nagaraj, S. Kumar, and S. Kumar. “Efficient and change-resilient test automation: An industrial case study”. In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 1002–1011.

[62] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra. “Robust test automation using contextual clues”. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 2014, pp. 304–314.

[63] B. Yu, L. Ma, and C. Zhang. “Incremental web application testing using page object”. In: *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. IEEE. 2015, pp. 1–6.

[64] R. Zhao, S. Zhang, Z. Zhu, Y. Shang, and W. Wang. “E2E test execution optimization for web application based on state reuse”. In: *Journal of Software: Evolution and Process*, 37(1) (2025), e2714.

[65] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu. “Automatic web testing using curiosity-driven reinforcement learning”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 423–435.

Appendix A Literature Review Sources

Table A.1: Literature Review Sources

ID	Author(s)	Year	Title
S1	Zhao et al.	2025	E2e test execution optimization for web application based on state reuse
S2	Di Meglio et al.	2024	Towards Predicting Fragility in End-to-End Web Tests
S3	De Luca et al.	2024	Investigating the robustness of locators in template-based Web application testing using a GUI change classification model
S4	Xinyue et al.	2024	WEFix: Intelligent Automatic Generation of Explicit Waits for Efficient Web End-to-End Flaky Tests
S5	Krasnokutska et al.	2024	Implementing E2E tests with Cypress and Page Object Model: evolution of approaches
S6	Huynh et al.	2024	Segment-Based Test Case Prioritization: A Multi-objective Approach
S7	Leotta et al.	2024	Ai-generated test scripts for web e2e testing with chatgpt and copilot: A preliminary study
S8	Leotta et al.	2023	Challenges of end-to-end testing with selenium webdriver and how to face them: A survey
S9	Nass et al.	2023	Similarity-based web element localization for robust test automation
S10	Brisset et al.	2022	ERRATUM: Leveraging Flexible Tree Matching to repair broken locators in web automation scripts
S11	Ngo et al.	2022	Research on Test Flakiness: From Unit to System Testing
S12	Fulcini et al.	2022	Guidelines for gui testing maintenance: a linter for test smell detection
S13	Leotta et al.	2022	Assessor: a po-based webdriver test suites generator from selenium ide recordings
S14	Zheng et al.	2021	Automatic web testing using curiosity-driven reinforcement learning

ID	Author(s)	Year	Title
S15	Nass et al.	2021	Why many challenges with GUI test automation (will) remain
S16	Olianas et al.	2021	Reducing flakiness in endto-end test suites: An experience report
S17	Nguyen et al.	2021	A machine learning based methodology for web systems codeless testing with selenium
S18	Ricca et al.	2021	Web test automation: Insights from the grey literature
S19	Olianas et al.	2021	Stile: a tool for parallel execution of e2e web test scripts
S20	Ricca et al.	2021	Towards automated generation of po-based webdriver test suites from selenium ide recordings
S21	Corazza et al.	2021	Web application testing: Using tree kernels to detect near-duplicate states in automated model inference
S22	Augusto	2020	Efficient test execution in End to End testing: Resource optimization in End to End testing through a smart resource characterization and orchestration
S23	Augusto et al.	2020	RETORCH: an approach for resource-aware orchestration of end-to-end test cases
S24	Long et al.	2020	Webrr: self-replay enhanced robust record/replay for web application testing
S25	García et al.	2020	A survey of the selenium ecosystem
S26	Yu et al.	2019	TERMINATOR: Better automated UI test case prioritization
S27	Ricca et al.	2019	Three Open Problems in the Context of E2E Web Testing and a Vision: NEONATE
S28	Medhat et al.	2019	Enhancing the automation of GUI testing
S29	Denaro et al.	2019	GUI testing in production: Challenges and opportunities
S30	Morán	2019	Debugging flaky tests on web applications
S31	Kirinuki et al.	2019	Color: correct locator recommender for broken test scripts using various clues in web application
S32	Biagiola et al.	2019	Web test dependency detection

ID	Author(s)	Year	Title
S33	Wongkampoo et al.	2018	Atom-Task Precondition Technique to Optimize Large Scale GUI Testing Time based on Parallel Scheduling Algorithm
S34	Rwemalika et al.	2018	Can we automate away the main challenges of end-to-end testing?
S35	Iyama et al.	2018	Automatically generating test scripts for gui testing
S36	Aho et al.	2018	Challenges in automated testing through graphical user interface
S37	Chiang et a.	2017	ATP: A Browser-Based Distributed Testing Service Platform
S38	Stocco et al.	2017	Apogen: automatic page object generator for web testing
S39	Leotta et al.	2016	Robula+: An algorithm for generating robust XPath locators for web testing
S40	Yu et al.	2016	Incremental web application testing using page object
S41	Hammoudi et al.	2016	Why do record/replay tests of web applications break?
S42	Stocco et al.	2016	Clustering-aided page object generation for web testing
S43	Leotta et al.	2015	Meta-heuristic Generation of Robust XPath Locators for Web Testing
S44	Leotta et al.	2015	Using multi-locators to increase the robustness of web test cases
S45	Leotta et al.	2014	Reducing web test cases aging by means of robust xpath locators
S46	Pirzadeh et al.	2014	Resilient user interface level tests
S47	Yandrapally et al.	2014	Robust test automation using contextual clues
S48	Mahmud et al.	2014	Design and industrial evaluation of a tool supporting semi-automated website testing
S49	Thummalapenta et al.	2013	Efficient and change-resilient test automation: An industrial case study
S50	Qian et al.	2011	Towards testing Web applications using functional components
S51	Daniel et al.	2011	Automated GUI refactoring and test script repair (position paper)

ID	Author(s)	Year	Title
S52	Choudhary et al.	2011	Water: Web application test repair
S53	Zhongsheng et al.	2009	An approach to testing web applications based on probable FSM