Aalto University
School of Science

Master's Programme in Computer, Communication and Information Sciences

Incorporating Test Automation Into Existing Software Systems

A case study of incremental and maintainable practices

Markus Lippo

Master's Thesis
2025

Aalto University
School of Science

Author Markus Lippo

Title Incorporating Test Automation Into Existing Software Systems — A case study of incremental and maintainable practices

Degree programme Computer, Communication and Information Sciences

Major Computer Science

Supervisor D.Sc. (Tech.) Juha Sorva

Advisor M.Sc. Jukka Tuomas

Collaborative partner PLM-Zone Oy

Date 10 November 2025 **Number of pages** 72 **Language** English

Abstract

The importance of automated testing has increased as organizations aim to deliver software more frequently, without compromising software quality. However, the distribution of higher- and lower-level tests, differences in test creation techniques, and the principles behind continuous integration are mostly aimed towards new development projects. Furthermore, common pitfalls, such as unclear strategies and technical debt, have been identified.

This thesis explores how test automation could be introduced into an existing system, while ensuring maintainability and future extensibility. This challenge was addressed by designing and implementing a pilot for a case company. The pilot consisted of four phases: planning, tool selection, implementation, and evaluation. The pilot defined what to automate, implemented a standalone test automation system, evaluated its feasibility, and outlined directions for future improvement.

The case company's multi-version web application demanded a modular and maintainable testing approach. The pilot prioritized high-value and repetitive steps, so it translated a specification document into automated regression tests on the user interface. This meaningfully reduced manual effort. The proof-of-concept scope enabled fast validation prior to large-scale commitment. Furthermore, aligning the strategy and tool selection with the product context proved essential. While various tools were considered, Robot Framework proved the best fit.

This thesis applies established concepts to an existing software environment and highlights a gap in prior research, which often focuses on new development projects. It also confirms known trade-offs, such as increased execution time and maintenance from higher-level testing. Practical recommendations include starting with high-value tests, limiting the initial scope, selecting tools that support the product context, and defining a long-term goal. The findings demonstrate that an incremental and context-aware approach is effective for introducing test automation.

Keywords Test automation, Software testing, Regression testing, Test automation tool, Robot Framework, Continuous Integration (CI)

Aalto-yliopisto
Perustieteiden
korkeakoulu

Tekijä Markus Lippo

Työn nimi Testiautomaation käyttöönotto olemassa oleviin ohjelmistoihin — Tapaustutkimus vaiheittaisista ja ylläpidettävistä menetelmistä

Koulutusohjelma Computer, Communication and Information Sciences

Pääaine Computer Science

Työn valvoja TkT Juha Sorva

Työn ohjaaja DI Jukka Tuomas

Yhteistyötaho PLM-Zone Oy

Päivämäärä 10.11.2025 **Sivumäärä** 72 **Kieli** englanti

Tiivistelmä

Testiautomaation merkitys on kasvanut, kun organisaatiot kehittävät ohjelmistojaan yhä nopeammin — tinkimättä kuitenkaan niiden laadusta. Testauksen eri tasot, testien eri luontitekniikat sekä jatkuvan integraation periaatteet ovat kuitenkin pääosin suunnattu uusiin kehitysprojekteihin. Tämän lisäksi tyypillisiä kompastuskiviä, kuten strategian puute ja tekninen velka, on havaittu.

Tämä työ tarkastelee, miten testiautomaatio voidaan ottaa käyttöön olemassa oleviin ohjelmistoihin, samalla huomioiden ylläpidettävyyttä sekä laajennettavuutta. Työ lähestyi tätä haastetta suunnittelemalla ja toteuttamalla pilotin kohdeyritykselle. Pilotti koostui suunnittelusta, testityökalun valinnasta, implementoinnista sekä arvioinnista. Pilotti määritteli mitä automatisoidaan, kehitti itsenäisen testiautomaatiokokonaisuuden, arvioi sen soveltuvuutta, sekä kartoitti jatkokehityksen suuntia.

Kohdeyrityksen moniversiotu verkkosovellus edellytti modulaarista ja ylläpidettävää testiratkaisua. Pilotti keskittyi toistuviin sekä korkean hyödyn testitapauksiin. Pilotti muutti spesifikaatiodokumentin käyttöliittymätason regressiotesteksi, joka vähensi merkittävästi manuaalisen testauksen määrää. Konseptintodistuksen (engl. proof-of-concept) avulla pilotin toimivuus voitiin validoida nopeasti ja riskittömästi ennen laajamittaista käyttöönottoa. Lisäksi strategian ja työkalujen valinnan peilaaminen sovelluksen kontekstiin osoittautui keskeiseksi Robot Frameworkin valinnassa.

Tämä työ soveltaa vakiintuneita testiautomaatioon liittyviä konsepteja olemassa olevaan ohjelmistoympäristöön sekä korostaa puutteita aiemmassa kirjallisuudessa, joka tyypillisesti keskittyy uusiin kehitysprojekteihin. Tämän lisäksi työ vahvistaa jo tunnettuja kompromisseja, kuten korkeampien testitasojen pidemmät suoritusajat ja suuremmat ylläpitotarpeet. Työn käytännön suosituksiin kuuluvat testiautomaation aloittaminen suurimman arvon tuottavista testeistä, laajuuden rajaaminen alkuvaiheessa, kontekstiin sopivan tuotteen valitseminen, sekä pitkän aikavälin tavoitteiden määrittely. Työ osoittaa, että vaiheittainen ja kontekstiin sopeutettu lähestymistapa mahdollistaa tehokkaan testiautomaation käyttöönoton.

Avainsanat Testiautomaatio, Ohjelmistotestaus, Regressiotestaus, Testiautomaatiotyökalu, Robot Framework, Jatkuva integraatio (CI)

Contents

Abbreviations

API	Application programming interface
CD	Continuous delivery
CI	Continuous integration
NLP	Natural language processing
UI	User interface

1 Introduction

How can an organization maintain software quality when updates and new features are frequent? For many, the answer lies in automated software tests.

Software testing is the process of evaluating the quality of software by executing it with the intent of discovering errors [1]. This process can be done either manually by the tester, or with automated scripts. While manual testing remains essential, the industry has increasingly shifted towards automated tests [2]. However, despite the availability of different testing tools and increased attention to the field, software testing is still not an exact science [1].

Common pitfalls in incorporating test automation have been identified. These include establishing test automation without a strategy, underestimating the human efforts involved, separating development and testing processes, and focusing on infrequently executed tests [3]. Many implementations of automated tests have also introduced technical debt, which degrades the management and use of a testing system over time [4]. Furthermore, much of the prior research is framed around new development projects, which makes applying it to existing systems more challenging.

This thesis explores the following overarching research question:

> How can test automation be introduced into existing systems to reduce manual testing efforts, while also ensuring maintainability and support for future extension?

This question is particularly relevant for the case company, because software testing needs to account for the generic version of the web application — as well as customer-specific releases that are derived from it. Furthermore, evolutionary development and frequent updates increase the need for maintainable tests.

To answer the research question, I designed and implemented a pilot for the case company. The pilot had four goals: (1) identify and automate the most repetitive parts of testing to reduce manual efforts; (2) establish a standalone test automation system that supports expansion; (3) demonstrate the feasibility of test automation by examining viability, maintainability, repeatability, performance, and cost; and (4) identify future development paths for the pilot.

The pilot focused solely on the generic version to maintain a manageable scope, but still provide a basis for future customer-specific releases. Figure 1 displays the steps of the pilot. Firstly, I created a test plan, which defined the context and identified the need for regression tests at the acceptance level. Secondly, I researched appropriate test automation tools and chose Robot Framework for its keyword-driven approach that enhances maintainability and modularity. Thirdly, I implemented the actual pilot by developing tests, setting up a test automation system with Jenkins, and deploying the automation system on a test server. Finally, I evaluated the pilot.

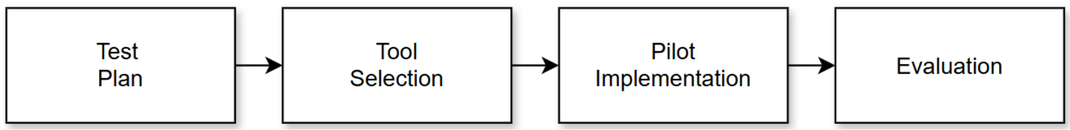

Figure 1: The pilot consisted of four phases

The pilot demonstrated that automating high-value and repetitive steps meaningfully reduced manual work. Implementing the pilot as a proof-of-concept allowed early and low-risk validation of feasibility before committing to wider adoption. Furthermore, aligning the strategy and tool selection with the product context resulted in appropriate tools and test design principles that support the required level of modularity and maintainability. This demonstrates that an incremental and context-aware approach is effective for introducing test automation.

This thesis applies established test automation concepts into practice. It confirms known tradeoffs, such as higher execution times and maintenance with higher level tests. On the other hand, it identifies gaps in prior research, which is often focused on new development projects. The thesis provides practical recommendations for organizations in similar situations, and emphasizes incremental development, context, and long-term goals.

This thesis is organized as follows. Section 2 defines software testing. Section 3 outlines test automation. Section 4 provides the test plan. Section 5 outlines the tool selection process. Section 6 describes the pilot implementation. Section 7 provides the evaluation of the pilot. Section 8 discusses the findings, practical recommendations, limitations, and future work. Section 9 concludes this thesis.

2 Software Testing

Defining Software Testing The 2020 National Academies of Sciences, Engineering, and Medicine report on Information Technology Innovation demonstrates the impact information technologies have across all areas of modern life [5]. Entire economies are dependent on software systems, either directly or indirectly. A day without software systems is a day where aircraft do not fly, banks are unable to transfer funds, factories halt production, and hospitals operate without medical imaging [5]. In addition to this widespread dependency, modern software systems are also much more complex than older systems [6]. This highlights the need for quality assurance.

Software testing is a central activity in quality assurance, but the definition of software testing has evolved over time. Earlier views emphasized error discovery in programs [7]. Newer sources identify software testing as detecting defects, gathering information and building confidence [1] [8]. Software testing does not prove the absence of defects, but instead it should uncover as many relevant defects as possible given the time and resource constraints [8]. The likelihood of finding relevant defects is increased if the tester's attitude leans towards discovering flaws instead of confirming behavior, as this directly affects test design [1].

Modern software systems are highly complex because they present an enormous number of possible input combinations and rely on integrations between existing systems. Furthermore, the software is always influenced by the underlying hardware. Trying to prove that a specific quality characteristic has been tested under all given circumstances — known as *exhaustive testing* — is not possible [8] [9] [7].

On the other hand, defects can remain unnoticed due to inadequate software testing. These defects can materialize into software downtime and outages, increased operational and maintenance costs, expensive defect fixes, reputation damage and user dissatisfaction. For safety-critical systems, the outcomes can be even more severe.

Software Quality and Lifecycle Taken together, software testing is a process of systematically trying to find defects in order to evaluate the quality of software. International standards propose nine *quality characteristics* for software [10], which are displayed in Figure 2. According to this quality model, software can be evaluated in terms of functionality, performance, compatibility, interactions, reliability, security, maintainability, flexibility and safety.

Software Product Quality								
Functional Suitability	**Performance Efficiency**	**Compatibility**	**Interaction Capability**	**Reliability**	**Security**	**Maintainability**	**Flexibility**	**Safety**
Functional completeness Functional correctness Functional appropriateness	Time behavior Resource utilization Capacity	Co-existence Interoperability	Appropriateness Recognizability Learnability Operability User error protection User engagement Inclusivity User assistance Self-descriptiveness	Faultlessness Availability Fault tolerance Recoverability	Confidentiality Integrity Non-repudiation Accountability Authenticity Resistance	Modularity Reusability Analysability Modifiability Testability	Adaptability Scalability Installability Replaceability	Operational constraint Risk identification Hazard warning Safe integration

Figure 2: Nine software quality characteristics [10]

The importance of detecting defects early was noticed already in the 1980s. The *cost-to-fix curve* outlines how the cost to fix a discovered defect increases exponentially with time [11]. While software testing should take place throughout the lifecycle, this relationship emphasizes the importance of testing early.

Figure 3 displays a generic *software development lifecycle*. A lifecycle typically begins with identifying requirements and designing the system, followed by implementation, and finally deploying and maintaining the software until it is no longer used.

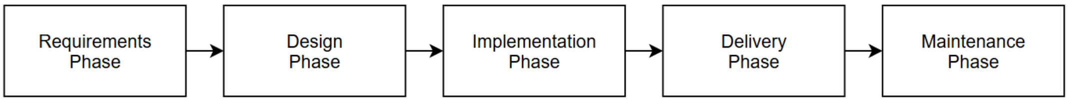

Figure 3: A generic software development lifecycle

Each phase of development can be associated with specific testing activities. To evaluate different quality characteristics as early as possible, different types of software tests are required. For instance, unit tests can be used during implementation to confirm functions behave as intended. Then, system-level testing can be used to evaluate the security and performance of an application. Before deploying the software to users, a tester could create tests on the *user interface* (UI) to evaluate usability. Finally when the software has entered operation, regression tests can be used to ensure future updates do not unintentionally break existing functionalities.

Adopting different types of software tests yields the following two benefits. Firstly, different quality characteristics of software can be evaluated by applying a wide range of test types. Secondly, utilizing more inexpensive and isolated tests, where applicable, makes detecting defects quicker and therefore reduces the cost of fixing them.

This section continues to expand on the software testing fundamentals by introducing the classical test levels, defining dynamic and static testing, and describing commonly used test types.

2.1 Test Levels

Software testing can be conducted at multiple levels. These levels include unit testing, integration testing, system testing, and acceptance testing. Each level targets the system at a different scope — the broader the scope, the higher the cost of testing [6].

Testing at Different Levels The *V-model*, displayed in Figure 4, is a well established model in software development and testing [6]. The model is shaped like a "V", where the left side represents different parts of development, and the right side represents different levels of testing. The model illustrates which parts of development these test levels evaluate.

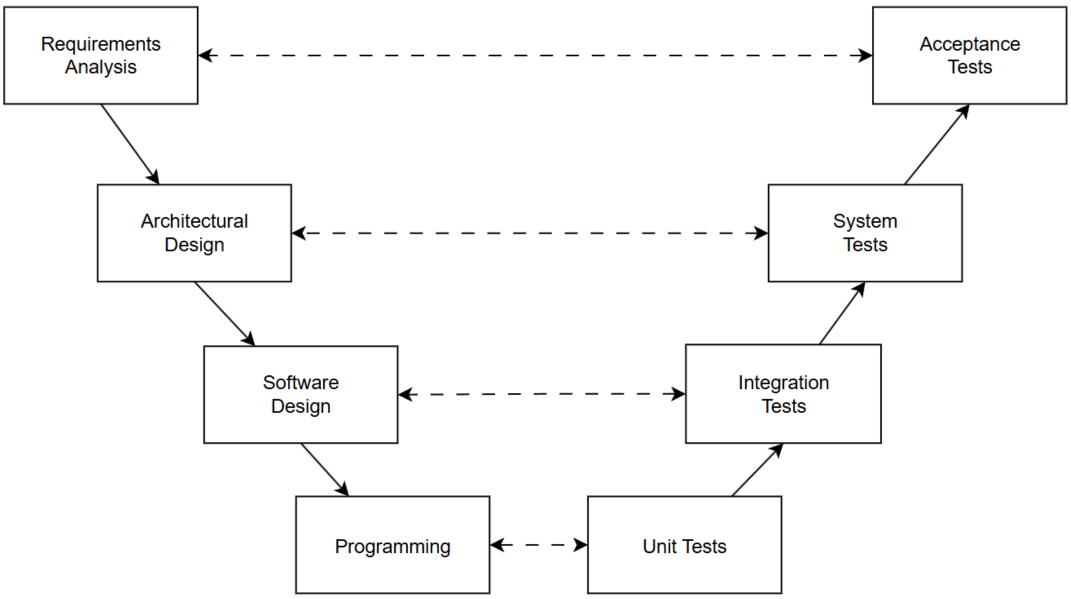

Figure 4: The V-model provides software development phases and their corresponding testing level

At the top left, development begins with requirements analysis, which yields specifications on what the system must do. *Acceptance testing* aims to verify that the finished software product aligns with the specified requirements [6]. Acceptance testing is often scenario-driven, as its purpose is to demonstrate that the system conforms to the requirements. Acceptance tests are the most expensive to create as they are not isolated and require end-to-end validation [12].

The architectural design defines the structure and behavior of the software system. *System testing* validates the system or major subsystems against their architectural design. Similarly to acceptance testing, system testing exercises the system as a whole. However, the purpose of system testing is to find as many defects as possible instead of strictly demonstrating conformance [6].

The software design stage details interfaces, outlines data flow, and specifies responsibilities between components. The corresponding testing activity is *integration testing*, which verifies that the components collaborate correctly [6]. Integration tests often verify communication formats and transactions, and instead of end-to-end validation, it utilizes mocks or stubs in place of other components. Integration tests are used to exercise actual component interactions, yet they remain faster than system tests.

Finally, the programming stage yields the actual functions. Its counterpart is *unit testing*, which verifies that a single function works according to its design [6]. Unit tests are deterministic due to their level of isolation. They also provide faster feedback and are more inexpensive to create due to their speed of creation [12].

In addition to these levels, *regression testing* is an essential practice throughout the lifecycle of the software. Regression testing refers to the re-execution of tests to ensure that after new changes are introduced, previously verified functionality has not been broken [6]. While regression testing is not a distinct level in the V-model, it is still central to the lifecycle of the software.

Other levels of testing have also been identified for different development contexts, including web development. Web applications rely on predefined and pretested technologies (e.g. browsers or frameworks) and the boundaries between test levels are often overlapping. Other levels such as code quality, functionality, usability, performance, and security have been proposed [9].

However, this thesis continues to use the more standard categorization of unit, integration, system and acceptance tests. Each level plays a role in software testing, since each level evaluates a different part of development [6]. Furthermore, employing different levels creates a compounding effect on confidence, as each level filters out specific types of defects.

Distributing Test Levels The *test pyramid* is another relevant model for test levels in software testing [12]. While the original model is somewhat dated due to its terminology, its central idea remains consistent, and is displayed in Figure 5. While all test levels are needed, the distribution should consist of more lower-level tests.

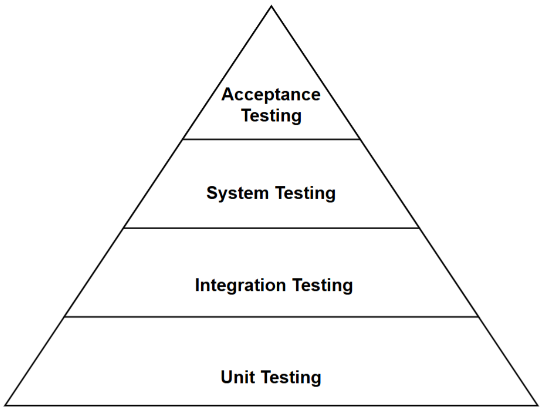

Figure 5: A recommended distribution of unit, integration, system, and acceptance tests

The test pyramid is based on the cost-effectiveness of lower-level testing [12]. Because lower-level tests are faster to create, they are less expensive to create and maintain. Lower-level tests are less prone to flakiness or maintenance issues due to their level of isolation. This simplicity enables them to be implemented earlier in the development lifecycle, which is beneficial as defects are far cheaper to address early on [7].

On the other hand, an ice-cream-cone approach, where the distribution of tests is flipped, results in increased costs. Tests are far more expensive to create and maintain, and the flakiness of tests is increased. Defects that are discovered at this stage are already too late to fix cost-effectively, and their root causes can remain unclear. Furthermore, even extensive coverage with acceptance tests may not exercise all internal code paths, leaving defects uncovered.

The V-model and test pyramid are both aimed at new development projects. By contrast, if an existing system requires tests to be written from scratch, the benefits of early defect fixes are already lost. While lower-level tests are fast to create, achieving meaningful coverage for existing codebases is time-consuming. System and acceptance-level tests, on the other hand, provide direct feedback and validate alignment with the system requirements, but suffer from greater maintenance and costs. No established testing models have been proposed specifically for existing systems.

2.2 Dynamic Testing

Dynamic testing evaluates the software by executing it. Figure 6 displays the three techniques to dynamic testing [13].

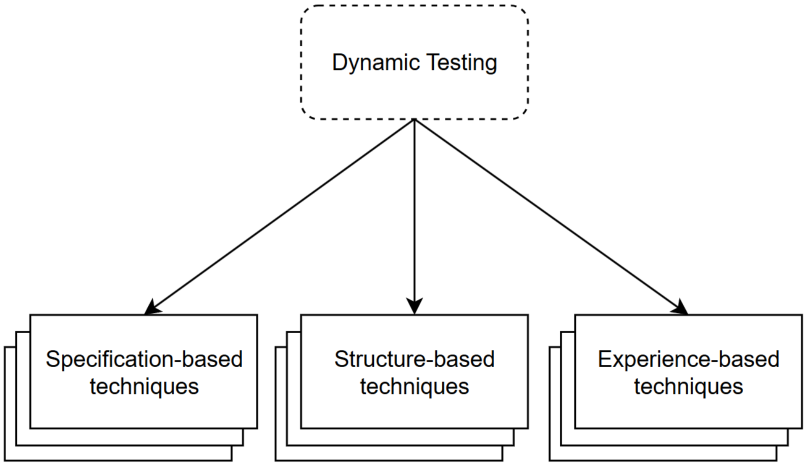

Figure 6: Dynamic testing is based on specifications, structure of the source code, or experience [13]

Specification-Based Testing *Specification-based testing*, more commonly known as black-box testing, is a testing technique which tests the software solely based on its requirements and specifications [9] [13]. Specific techniques for specification-based testing include decision table testing, scenario testing, boundary value analysis, equivalence partitioning, and more [13]. In practice, the tester requires no knowledge of the internal source code, but rather only validates inputs and corresponding outputs.

Specification-based testing is used to ensure the software aligns with its requirements and documentation. Additionally, the tests can be designed early and independently of the actual implementation. On the other hand, specification-based testing cannot guarantee how much of the internal software has actually been tested. The risk of uncovered defects is greater if the tester does not observe the internal implementation [9]. Furthermore, finding every possible defect is impossible because the tester would have to test every possible combination of inputs [9]. Lastly, the quality of tests depends on the quality of the documentation.

Structure-Based Testing *Structure-based testing*, more commonly known as white-box testing, is a testing technique which tests the actual internal implementations of the software. The goal of white-box testing is to gain test coverage by exercising the statements, paths, branches, and conditions of the code during a test [9] [13]. Structure-based techniques include statement testing, branch testing, decision testing, branch condition testing, modified condition decision coverage testing, and more [13].

In practice, structure-based tests are based on the actual implementation of functionalities. For example, let us consider a function that calculates the discount rate in an online store that applies discounts based on customer type, total purchase amount, and discount codes. In this instance, the tester would examine the source code and design tests that intentionally execute each conditional branch with input combinations of valid and invalid customer types, purchase totals, and discount codes. In addition, this approach allows tests to verify thresholds and boundary conditions of the implemented logic [13].

Structure-based testing is particularly powerful in uncovering defects on specific combinations of inputs. Additionally, it detects defects caused by human error, such as logic flaws, off-by-one errors, and incorrect handling of data structures. Unlike specification-based testing, this approach provides concrete test coverage [9] [13]. On the other hand, this style of testing requires knowledge about the source code. Additionally, higher test coverage does not necessarily mean that the software aligns with the requirements.

Experience-Based Testing *Experience-based testing* relies on the knowledge and intuition of the tester. In this approach, tests are not based on requirements or the implementation, but on the tester's knowledge [13]. Exploratory testing is a well-known example of this approach, where the tester interacts with the system and guesses where defects could occur while operating the software. Experience-based testing includes error guessing and checklist-based testing [13].

Experience-based testing is powerful in its ability to uncover defects that might be more time-consuming to uncover with more formal techniques. Additionally, this approach is effective for discovering defects from the end user's perspective. Experience-based testing is easy to conduct and can be used in situations where requirements or documentations are limited, resources are constrained, or when testing has not been formally planned. On the other hand, the results from this approach depend heavily on the tester's expertise, and the results cannot be as easily reproduced.

To conclude, dynamic testing can further be divided into three different approaches. These approaches use the requirements of the software, the structure of the source code, or the tester's experience to design tests. While each approach has its benefits and drawbacks, combining the approaches results in more effective testing [13] [9]. Tests are often designed with a mixture of these approaches. For instance, when a developer tests their own code, they may design tests based on the requirements of the software and the actual implementation, while also relying on their experience by trying to guess common defects. However, distinguishing between the different test techniques clarifies the strengths and weaknesses in using different sources of information when creating tests.

2.3 Static Testing

Static testing evaluates the software without executing it. Static testing is conducted by analyzing artifacts such as design documentation or source code. Figure 7 displays different techniques in static testing [8].

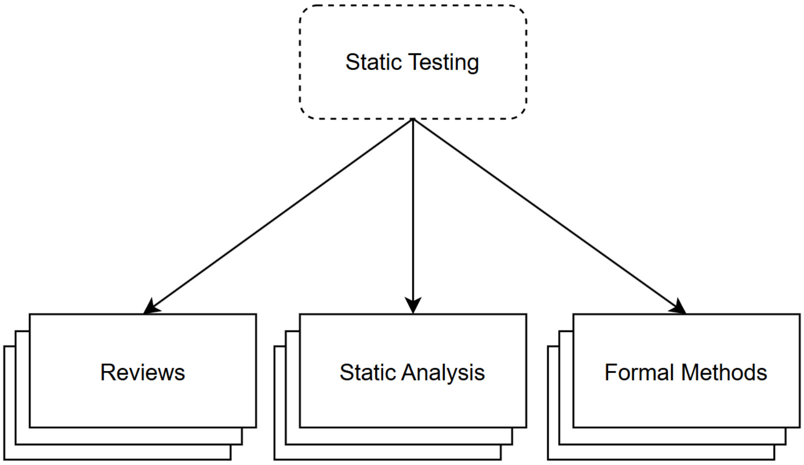

Figure 7: Static testing analyses the software by conducting reviews, static analysis on the source code, or by using formal methods [8]

Static testing encompasses reviews (e.g. ad-hoc meetings, walkthroughs or technical reviews), static analysis that inspects source code with linters, compilers or code analyzers, and formal methods that apply mathematical models to prove correctness. Although International Standards classify formal methods separately, they are discussed here because these approaches also evaluate software without executing it [8].

Static testing is effective for evaluating the software from a different perspective. Static testing is economical and allows defects to be identified much earlier in the lifecycle [14]. Additionally, depending on the domain, these software systems might require mathematical methods to prove correctness. If no testing process is yet in place, reviews are a quick method of validation. Moreover, static testing is a good way to ensure consistency, maintainability, and code quality.

Static testing also has weaknesses. It often has limited defect detection compared to dynamic testing, as it cannot discover runtime or integration defects [14]. The quality of static testing also depends on the participants involved, and mathematical models might be either too time-consuming or impractical.

2.4 Test Types

Table 1 lists *test types* defined by International Standards [13]. These test types describe categories of tests in terms of which quality characteristics are evaluated, without dictating specific test levels or techniques.

Test type	Description
Functional testing	Tests that the software behaves according to the requirements or intended functionality.
Accessibility testing	Ensures that the system is accessible to users with disabilities.
Compatibility testing	Validates that the software works with different browsers and operating systems.
Conversion testing	Validates that data is correctly converted between systems.
Disaster/recovery testing	Tests whether the system can recover from crashes or failures.
Installability testing	Tests whether the software can be installed, uninstalled, or updated correctly.
Interoperability testing	Ensures the system interacts correctly with other components or systems.
Localization testing	Verifies that the software behaves correctly with different languages and formats.
Maintainability testing	Evaluates how well the software can be modified.
Performance testing	Evaluates the performance of the system by testing load, stress, capacity and recovery capabilities.
Portability testing	Tests whether the software can be run on different hardware or environments.
Procedure testing	Verifies that procedures, such as a backup procedure, are supported.
Reliability testing	Ensures the system remains reliable and performs the intended functions under specific conditions.
Security testing	Evaluates the security of the software from different perspectives such as data protection or attack mitigation.
Usability testing	Evaluates the ease of use of the system.

Table 1: Different test types aim to evaluate one or more software quality characteristic

While these test types can be executed on different test levels and with different techniques, they are often associated with specific approaches. For instance, functional testing is often conducted on the unit level to test a specific function. On the other hand, some test types benefit more specific techniques — for instance, the usability testing of the system might benefit more from experience-based testing, whereas functional testing might benefit more from structure- or specification-based tests. Finally, certain test types, such as security or maintainability testing, benefit more from static testing tools and techniques.

Depending on the software system, not all of these test types are equally relevant, however. For example, installability testing and portability testing are not as relevant to web-based applications, as these applications are already deployed on existing and tested infrastructure. Nevertheless, identifying the different types of tests highlights the approaches to evaluating quality characteristics.

3 Test Automation

Dynamic software testing can be conducted either manually by the tester, or with automated scripts. *Manual testing* and *automated testing* serve complementary purposes. For instance, manual testing excels at ad-hoc testing whereas automated testing excels at repeatable test coverage. Even though organizations value automated testing, it is often under-utilized [15].

When *test automation* is suitable for a given context, it provides faster feedback loops, increases test coverage and provides reusable tests that can be run at scale. However, this comes at an initially higher cost in building and maintaining tests. Testers also require training about operating the testware. The *return on investment* of test automation depends on the frequency of execution and the stability of functionalities. The benefits are also increased by prioritizing suitable test levels that minimize maintenance [15].

This section continues to define automated testing. First, it outlines manual testing and motivates the need for test automation. Then, automated testing, relevant challenges, and continuous integration (CI) are discussed.

3.1 Manual Testing

Manual testing is predominant at higher test levels because it allows the tester to operate the software via the UI [16]. Manual testing is particularly beneficial for tests that are executed infrequently, or when resource constraints impact the ability to develop automated tests [8] [16].

Manual testing excels at experience-based testing, as human intuition and creativity allow unexpected defects to be found. Moreover, real-time decision making during test execution allows for more flexible tests. It is also helpful for assessing user-facing quality characteristics, such as usability or accessibility. The return on investment of manual testing is greatest in ad-hoc testing or infrequently executed tests due to the low upfront costs [16].

While the upfront costs are lower, the long-term costs of manual testing are much higher[16], which implies that relying on manual testing is unsustainable in the long run. As the number of tests grow, manual testing becomes impractical due to increased execution times. Additionally, the results from these tests can be inconsistent, since the quality depends on the expertise and rigor of the tester.

Nevertheless, manual testing continues to exist [16], as it is particularly useful for detecting certain defects or when automation has not yet been implemented. However, it should not be carried out completely ad-hoc, since it is most effective when supported by prioritization, quality monitoring, and overall strategy [16].

3.2 Automated Testing

Automated testing refers to executing tests with scripts or specialized tools instead of manually executing them [8]. Unlike manual testing, automated testing is applicable for all levels of testing. While the initial investment for automated tests is higher, they are especially cost-effective for frequently executed tests [8]. Benefits from automated testing include repeated execution at scale, increased test coverage due to more comprehensive and structured tests, and objective machine-based quality evaluation.

While useful, automated testing also introduces a different set of challenges, including higher initial costs [15], increased maintenance overhead [3] [4], possible technical debt [4], and reliability issues [17]. Additionally, effective test automation requires a new set of skills and infrastructure setup [3] [18]. These challenges are discussed further in Subsection 3.3.

A wide range of tools is available for automating tests. Because the boundaries between test levels are often overlapping, especially for web applications, the choice of an automated testing tool depends on the intent of testing. For example, JUnit, pytest or Jest are often used for isolated unit tests. Tools like REST Assured, Postman, pytest with plugins and other environment configuration tools allow integration testing of components, services and Application Programming Interfaces (APIs). For system level or acceptance level testing, browser automation frameworks such as Selenium or Playwright are often used.

3.2.1 Approaches to UI Automation

Test automation tools that test the user interfaces are particularly relevant for this thesis, and these approaches have seen advances in the industry. International Standards propose three common approaches, which are *programmable testing*, *capture-replay testing*, and *keyword-driven testing* [8]. Additionally, a fourth, more recent approach exists: testing based on *natural language processing* (NLP) [19].

Programmable Testing In programmable testing, tests are written and maintained like source code. With these tools, the tests are written with a programming language and use frameworks such as Selenium WebDriver for automating the browser interactions. This approach enables tester direct control over their tests, which enhances modularity and maintainability, and offers support for more complex scenarios such as loops, conditionals and exception handling. This approach requires a greater initial time investment as well as programming knowledge from the tester. Some programmable testing frameworks also support capture-replay methods with extensions.

Capture-Replay Testing Capture-replay tools are attractive due to their initially fast test creation [19]. The tools require fewer programming skills from the tester. In essence, these tools abstract the underlying technical implementations and provide testers with a toolkit to record their interactions. For modern capture-replay tools, this toolkit often includes the recorder itself plus additional reusable components and functions. During testing, these actions are then replayed back.

Keyword-Driven Testing International Standards identify keyword-driven testing [20]. However, this is more of an approach to test creation instead of a strict tool category. Instead of programming the exact actions that should be performed, the actions are abstracted behind human-readable keywords or functions. Then, tests are created by combining these modular keywords that can either represent small actions or longer sequences of steps. This makes testing more understandable and maintainable, and makes tests easier to create [20]. In practice, a tool uses keyword-driven testing if it encompasses actions behind keywords or other modular components.

Natural Language Processing (NLP) Testing NLP-based testing is a relatively new approach where natural language instructions are translated or parsed into corresponding test cases [21]. Test creation with NLP has been identified as faster than programmed tests but slower than capture-replay [21]. The key benefit around NLP based testing is that tests are less tied to element selectors, and the tests are instead interpreted and mapped to elements [22]. As tests only capture the overall logic of user interactions instead of specific selectors, the tests should be more reliable and allow non-developers to create them. However, drawbacks exist, such as limited expressions and trial-and-error test creation [21]. Additionally, a key trade-off is the fact that the tester no longer has clear control over the tests. NLP-based testing is currently an interesting area in the field due to the implications of machine learning and artificial intelligence.

3.2.2 Comparing UI Automation Approaches

Research shows a cost difference between programmable tests and capture-replay tools. The initial investments for programmable tests are considerably higher when compared to capture-replay tools. However, the programmed tests are also cheaper to repair and maintain across releases. In most cases, programmable tests become more cost-effective as the number of releases grows [19]. This stems from the increased control over tests. Programmable tests often use centralized and reusable patterns that make the maintenance of tests more cost-effective. Capture-replay tools, on the other hand, enable fast test creation at the expense of maintainability [19].

As capture-replay tools do not necessarily encourage strong test design rules, they can lead to larger maintenance issues or even total re-engineering of tests [3]. However, more modern tools have somewhat overcome this limitation, for example by enabling the use of reusable recorded components. Modifying a test case can be more time-consuming than recording a new test from scratch [21]. Additionally, hard-coded values are an ongoing issue with capture-replay tools that affect the modularity of such component reuse [21].

Nevertheless, all the above approaches serve different purposes. User interfaces with stable, but unique pages favor capture-replay tools, whereas modular and frequently changing pages are better suited for programmable testing. Adopting a keyword-driven approach further enhances readability and modularity, which makes tests easier to understand [20]. NLP-based testing is particularly useful in contexts where interactions can clearly be expressed in natural language. All things considered, programmable tests still seem a better choice for new development projects due to the cost effectiveness of programmable tests over multiple releases [19].

The question between choosing programmable, capture-replay, keyword-driven, and NLP-based testing remains ambiguous, especially for existing systems. The fewer future releases and rapid UI coverage make capture-replay or NLP-based tools attractive. On the other hand, if the system is still evolving or the structure of the application supports modularity, programmable tests might be a better alternative. In cases where testing relies heavily on higher-level tests, programmable testing could be more advantageous due to the support for more complex tests. Conversely, if testing adheres to the test pyramid, capture-replay or NLP-based testing tools may be applicable for the fewer high-level tests.

3.3 Test Automation Challenges

While the field contains research and case studies on incorporating test automation, no single solution on how to design, implement, and maintain a test automation system exists. Numerous pitfalls and challenges have been recorded (e.g. [3]). Relevant issues include organizational factors and testware challenges. Furthermore, challenges with multi-version testing are particularly relevant for the case company.

3.3.1 Organizational Factors

Test automation is not only limited to technical activities, but it also includes socio-technical challenges that involve interactions between people, processes, and testware. *Organizational factors* are often the "make-or-break" of incorporating test automation [18]. Unrealistically high expectations, lack of management, and constraints with time and funding can all derail the automation effort [18].

Incorporating test automation requires a significant upfront investment. Because of this, the benefits of automation often materialize only when it is incorporated consciously. The absence of a strategy, guidelines, responsibilities, and future vision leads to worse outcomes [3]. A strategy should articulate what, why and when to automate.

The biggest return on investment from test automation comes primarily when frequently executed tests are automated [3]. Moreover, business risks and cost-to-maintain influence the choice of what to test. Tracking the coverage of what is automated is useful for determining gaps in testing, whereas unknown coverage undermines the confidence in testing [3].

Organizations should complement manual testing with automated tests — not replace it with them [3] [18]. Human involvement is still required for the testing efforts, whether it be conducting specific types of tests or managing the automated testing process. These activities require human involvement and skills, and underestimating this involvement stalls adoption of test automation. Lack of training and proper skills can also cause reworking and refactoring overhead [3] [18].

Introducing test automation too early results in changed requirements, which incur additional refactoring and maintenance overhead for tests. Introducing it too late diminishes the benefits of preventative testing and early defect detection. Automation should be introduced incrementally with clear ownership, otherwise existing problems only accelerate [3] [18].

3.3.2 Testware Challenges

Testware challenges often influence the success of incorporating test automation, and technical debt is a key challenge in test automation [4]. Technical debt degrades the use and management of the testing system over time. These systems often accumulate technical debt, as teams often treat testware as less important than the software and its source code. Additionally, management or speed pressures might force testers to create unoptimized tests and infrastructure [4]. Tool selection also plays a role, since inappropriate tools can cause technical debt or other hidden costs that surface much later [3].

A big factor in technical debt is the lack of modularity and reusability which leads to high maintenance costs. The absence of systematic test design practices further increases the maintenance costs, as scripts do not follow a pattern [4]. Other tooling issues, such as limitations or usability issues, can also hinder the testing effort.

Fragile and flaky tests are tests that pass or fail the same source code nondeterministically [17]. This issue is particularly relevant for higher-level tests because they are not as isolated, which increases the likelihood of the test breaking. A root cause of flakiness in user interface tests are asynchronous timing issues [17]. In essence, these timing issues happen when scripts expect certain elements to be present, but they are not. Increasing waiting periods have no effect on flake rates and slow down test suites [17]. A majority of these problems are caused by inappropriate testware and scripts, but some defects are caused by problems in the source code [17].

The size and complexity of the testing system are often underestimated. When the software product is large, the corresponding test suites tend to reflect this size. Larger testing systems require more dedicated attention, and the lack of resources or an overly complex system make developing tests alongside the software impossible [18].

The lack of proper configuration management, infrastructure, and environment control reduces the reliability of the system [4]. Without proper infrastructure management and stable environments the tests produce unreliable results. Unreliable results reduce the trust in the system and incur longer debugging time.

3.3.3 Multi-Version Testing

In the context of the case company, a *multi-version environment* refers to a setup where the company maintains a single generic release and parallel customer-specific releases. These customer-specific releases keep the same shared core but may apply certain customer-specific overrides. In practice, the case company maintains and evolves all of these releases. While testing multi-version environments does not necessarily differ from regular testing activities, it introduces additional overhead.

This introduced overhead can amplify the conventional challenges of test automation in multi-version setups. Organizational factors (e.g. planning, responsibilities, environments, test traceability) are greater because the required coordination is increased. For instance, test execution needs to account for version differences and a defect in the shared core may require subsequent tests across multiple releases. As a result, organizational involvement may need to be more deliberate than in traditional approaches. Additionally, challenges to testware are likely aggravated as the number of required environments is increased and tests need to account for more flexibility. For example, maintaining separate environments for each customer release increases operational complexity, and tests may require variables and configurations to support version-specificity.

Concrete testing strategies tailored to multi-version environments have not been proposed in the field. However, insights from overlapping research areas might prove helpful. For instance, studies on vulnerability discovery methods (e.g. [23]) can help identify shared component vulnerabilities and guide where test efforts should be concentrated. Similarly, research in test case selection and prioritization (e.g. [24]) can help optimize test execution by determining the most effective order of test cases to improve fault detection across multiple releases.

A practical advantage of a shared core is that defects tend to propagate across versions, meaning that vulnerabilities can be discovered without explicitly testing every version [23]. On the other hand, a shared core enables shared test scripts to be executed on different releases. The amount of shared code directly dictates the amount of shared vulnerabilities and defects between versions [23]. Changed features or previously failed tests are still disproportionately more valuable than general test coverage and yield better quality and time gains on the long run [25]. However, if the shared core logic were to diverge between customer versions, the benefits to test reuse would be diminished and testing efforts would increase tremendously.

3.4 Continuous Integration

Continuous Integration (CI) is a development practice that automatically integrates code changes, compiles or builds the software, and runs relevant tests on it [26]. It is particularly relevant for software development models that deliver in frequent releases (e.g. agile or evolutionary development), where CI ensures code changes are appropriately merged, built, and tested. This process can be conducted as frequent as multiple times per day [26].

Similarly to regression testing, CI ensures that new code and commits are integrated successfully into the software without breaking existing functionalities. CI is enabled by pipelines, which codify each stage into executable steps that can either pass or fail, thus determining whether an update advances and is ultimately accepted into the codebase.

CI and automated tests yield an increased number of detected defects, reduce manual work, improve the quality of the software, and raise confidence in releases [27]. By correctly applying CI and automated tests, defects can be detected early before they propagate into deployment, where they become significantly more costly to fix [28]. Automated tests and CI enforce the *shift-left principle*, where tests are performed as early as possible — for instance, even before the code is merged into the main codebase.

Automated tests in CI still contain drawbacks. Firstly, the infrastructure and pipelines require initial setup and ongoing maintenance efforts. Secondly, pipelines require skills and training that is non-trivial [26]. Thirdly, pipelines can cause bottlenecks if builds are long and feedback is delayed [28]. Fourthly, higher-level tests can have increased drawbacks in CI: slow execution times, environment inconsistencies between local and CI, as well as false failures from flaky tests [27], which can make debugging more time-consuming, create alert fatigue, force pipeline re-runs or even require manual investigation.

In addition, CI is often associated with *continuous delivery* (CD). CI focuses on integrating changes and testing the codebase, whereas CD focuses on automatically deploying and releasing software changes to production environments. While closely related to CI, CD primarily concerns with deployment, release and monitoring of software instead of test automation, and therefore falls beyond the scope of this thesis.

4 Pilot Test Plan

This section acts as the documentation for the pilot's test plan. It outlines the motivation and objectives for the pilot. Then, it outlines the strategy and approach to testing. Finally, it provides the risks and deliverables of the pilot.

4.1 Motivation and Objectives

The demand for systematic quality assurance increases as the system matures. The case company provides a web-based cost control solution for collaborative construction projects, where users belong to different organizations and project parties. Users operate under roles that require different subsets of views and functionalities, which has made testing more cumbersome as each role needs to be explicitly tested. Furthermore, the software contains a *generic version* and multiple *customer-specific releases*, which increases the amount of needed test coverage.

The development follows an *evolutionary model*, where the product evolves through updates as new requirements emerge. Frequent updates to both the generic version and customer-specific releases have made quality assurance especially difficult, as old and new features need to be validated accordingly.

Automating tests is one way to address this challenge. Although Section 3 outlines that code-based tests are maintainable, earlier test automation efforts became difficult to maintain and adapt to changes, partly because the implemented tests may not have adhered to clear strategies or design principles.

This thesis acts as a pilot for maintainable and incremental adoption of test automation, and the pilot is a part of a broader effort to improve the case company's build, test and release processes. The pilot focuses on automating tests for the generic version, which serves as the baseline for customer-specific releases, ensuring the initial scope remains manageable. By developing the pilot, the thesis simultaneously aims to answer the overarching research question:

> *How can test automation be introduced into existing systems to reduce manual testing efforts, while also ensuring maintainability and support for future extension?*

The pilot has four goals, and each goal addresses the research question from a different perspective. The goals are listed below.

G1 Identify and automate the most repetitive parts of testing to reduce manual testing efforts.

G2 Establish a standalone test automation system that supports expansion.

G3 Demonstrate the feasibility of test automation by examining viability, maintainability, repeatability, performance, and cost.

G4 Identify future development paths for the pilot.

4.2 Strategy and Approach

Automating frequently repeated tests leads to the greatest benefits, as discussed in Section 3. Consequently, the first goal — identifying the most repetitive parts of testing — is most important and dictates the approach to testing.

Test Design For the case company, the most time-consuming and repetitive part of testing is the manual regression testing of the UI. In particular, the UI is tested with an *access matrix*, which is a document that defines user roles and their permissions to specific views and functionalities. Automating these tests is a good starting point for the testing effort, as it provides a sufficient level of quality assurance and validates the most critical end-user requirements. However, it needs to be acknowledged that this approach overemphasizes higher-level tests, which can have their own issues that were outlined in Section 2.

Table 2 illustrates a simplified access matrix. In this example, the administrator user has permissions to all views, the contractor role has permissions to two views, and the subcontractor role has permissions for one view only. This example illustrates the repetitiveness of thoroughly testing each view and role. In reality, the access matrix spans over 6 different user roles and 20 different views — with half of these views having their own sub-functionalities.

	Admin	**Contractor**	**Subcontractor**
Admin View	Allow	Deny	Deny
View 1	Allow	Allow	Deny
View 2	Allow	Allow	Allow

Table 2: Simplified access matrix

The pilot aims to automate regression tests on the user interface. Particularly, the tests should translate the access matrix document into corresponding test cases. A test suite should test whether a given role is permitted or not permitted to the views and functionalities outlined in the document. This approach is considered specification-based testing at the acceptance level, as the tests primarily evaluate whether the software conforms to the specified behavior.

In addition to being the most repetitive and time-consuming part of manual testing, the access matrix document also provides a clear source of knowledge for tests as it outlines test cases and their inputs and expected outputs. Simultaneously, this information is used to determine whether a given test passes or fails. Not only is this useful because it removes the ambiguity in designing tests, but it also makes testing more traceable and provides overview into what has been tested.

A full test run should include the entire set of test cases from the access matrix document. This test run is considered complete if all of the six roles have passed. As each role corresponds to a test suite, the full test suite run contains six test suites. Each test suite contains 20 views, and these views are tested with one or more test cases. These test cases vary from simple navigational checks to more complex interactions.

Because the aim is to automate these tests, they need to be either scripted or pre-defined to allow for automated execution. Although Section 3 highlights that programmable approaches are maintainable, the case company is considering less strictly programmable tools, and instead favors capture-replay, NLP-based, or keyword-driven approaches.

Automation System The pilot should be conducted first on the local environment that hosts the generic version of the software. Testing the software at the acceptance level requires end-to-end validation, which includes the server and the database. As these tests interact with dynamic data, the test environment should include a working test database. To keep tests repeatable, this environment needs to remain consistent. This consistency is guaranteed by resetting the test database and clearing the browser memory.

After the local implementation has been developed, the implementation can be deployed to a test server. Test runs would be executed manually by the tester, or in a scheduled interval. The level of automation is initially low, and the test automation system should contain the test scripts, a Jenkins job that runs the tests, and an integration that reports the test results to Jira. Jenkins is an open-source automation server, whereas Jira is a project management and issue-tracking tool that the case company uses.

Approach Summary In summary, the chosen testing approach should conduct *regression tests* at the *acceptance level*. Regression tests are used to ensure that the software continues to conform to the role-specific behavior defined in the access matrix document. Because tests are derived directly from documentation, they are *specification-based* tests. The conformance is tested through *functional tests* that target the UI. Therefore the selected automation tool needs to support UI testing through *capture-replay*, *NLP-based*, or *keyword-driven* test creation. This enables an automation setup where the tester can run tests either locally or on the test server. Local testing should support flexible test creation and modification, while test server is reserved for formal test runs where results are automatically reported to Jira.

4.3 Pilot Risks

The following part describes the risks associated with the pilot. As discussed in Section 3, test automation suffers from common pitfalls and challenges. Relevant pitfalls are summarized here and used to evaluate the feasibility of the pilot after the implementation. These risks can be split into *project risks* and *product risks* [8].

The project risks relate to the management of the testing efforts. Table 3 outlines these risks, which are mainly concerned with the adoption of the pilot and the management of processes and resources [8].

Category	Project Risks
Adoption	Pilot adoption is hindered by resource constraints or attitudes.
	Knowledge transfer gap or overall lack of experience in the tools and infrastructure hinders adoption.
Processes	No long-term strategy for testing exists.
	Development and testing remain as separate processes.
	Unclear responsibilities regarding test maintenance and execution.
	Unclear processes regarding the handling of discovered defects.
	Testing remains ambiguous and hinders traceability.
Infrastructure	Tests lack clear version control practices.
	Test environment and integrations require more maintenance than expected.
Estimation	The number of discovered defects is much higher than expected and delays overall testing progress.
	Underestimating the human efforts to design, implement, and maintain tests.
Tooling	Selected automation tool is inappropriate for the chosen approach.
	Test execution time grows significantly as coverage increases.
	Selected automation tool does not scale or remain maintainable.

Table 3: Project risks concern the management of testing activities

The second type of risk is the product risk, which relates to the quality of the product [8]. Table 4 outlines these risks, which are mainly concerned with product quality, test flakiness and conflicts with the specifications.

Category	Product Risks
False Coverage	Pilot creates false confidence and critical regressions remain unnoticed.
	Test data is not representative of real data and misleads outcomes.
Defects	Tests reveal a high number of defects.
	Decisions in code conflict with specifications.
Stability	Test flakiness from timing, page loading, and element visibility issues leads to false failures.
Scalability	Test scripts do not scale.
	Changes to UI require frequent modifications of test scripts.
Environment	Tests are not fully replicable due to version and environment differences.
	Test server or database is in a state which disrupts test execution.

Table 4: Product risks concern the software and the testing process

4.4 Pilot Deliverables

The pilot should produce two types of *deliverables*. Firstly, the pilot should produce dynamic test deliverables, which are different documents produced from executed tests. Depending on the chosen tool, these documents might include reports, logs, screenshots, and Jira issues from successful and failed test runs. Additionally, the automation server Jenkins produces deliverables related to execution times, execution dates and success rates. The dynamic deliverables play a crucial role in adopting test automation into the case company's workflow.

Secondly, pilot deliverables are documents created to guide the adoption of the pilot. These deliverables include at least this thesis and its test plan, version-controlled test scripts, the infrastructure setup and a how-to-guide that covers test creation and execution. Additionally, a document that outlines which steps of the access matrix have been translated to tests is produced. These deliverables are used to increase the traceability of the pilot and reduce possible knowledge transfer gap.

5 Pilot Tool Selection

The choice of tool can influence whether a given test automation system incurs technical debt or other hidden costs, as discussed in Section 3. This section details the tool selection process, which is displayed in Figure 8. This process began with outlining the requirements and identifying possible tool candidates from various sources. Then, systematic elimination was used to progressively narrow down the pool of tool candidates. Finally, four tools were selected for a practical evaluation, leading to the final tool choice.

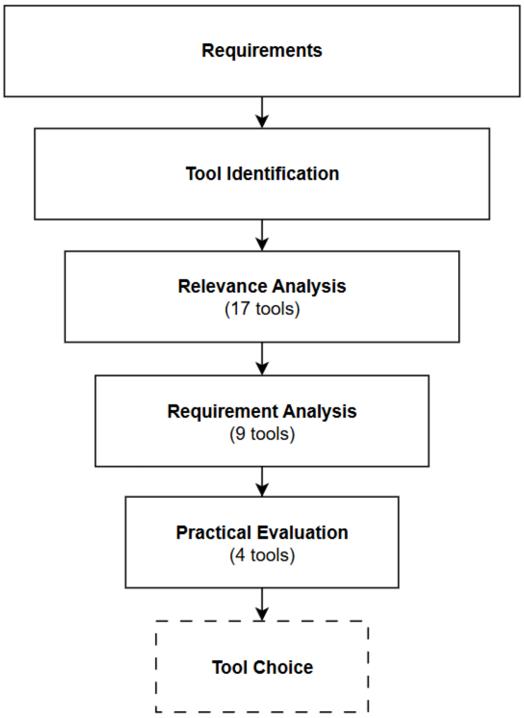

Figure 8: Pilot tool selection steps

5.1 Tool Requirements

The testing tool plays a key role in ensuring the success of test automation efforts. For this pilot, the chosen test approach limits the selection of tools to those capable of browser automation. Furthermore, the case company and I identified some of the most important requirements for the tool. These requirements are used to eliminate tools which are not relevant or appropriate for the pilot, and they are listed below.

R1 **Test creation** The tool should allow tests to be created and maintained with minimal programming. Suitable test creation techniques include capture-replay, NLP-based testing and keyword-driven approaches. Test creation should support dynamic data.

R2 Flexibility The tool should support testing different parts of the software, including atleast the UI and the API. Tests should be run either manually, in scheduled intervals, or after certain events. If the tool does not natively support different execution methods, then integration possibilities to automation servers or CI tools are acceptable.

R3 Cost The pricing should be transparent and justifiable in relation to the value the tool provides.

R4 Integrations The tool should include atleast the following integration possibilities: failed test results should be reportable to Jira for issue tracking; tests should be version-controlled; the tool should support CI pipelines.

R5 Ecosystem The users of the tool should have similar needs as the case company to ensure that the tool is appropriately matched to the testing needs. The tool should remain supported in the future.

R6 Maintainability and repeatability Tests should be maintainable to account for changes and new features. Tests should be resilient to flakiness to ensure repeatability across environments. The tool should remain effective and intuitive if the number of tests, test suites, and testable versions grows.

One initially considered requirement proved to be redundant during the research phase of the tools. Almost all of the considered tools had some level of platform lock-in due to non-code test creation, which abstracts the underlying technical implementations and ties tests to that specific platform.

5.2 Identification and Analysis

Tool identification The tool selection process phase began with tool identification. Initially, I gathered a pool of 17 candidate tools from three different sources. Candidates were gathered from online searches, prior knowledge, or were provided to me by the case company. Table 5 lists these tools and divides them into three categories.

Programmable tools (4)	No-code and low-code tools (9)	Test execution and management platforms (4)
Selenium Playwright Cypress Robot Framework	Katalon Studio Mabl Tricentis Testim Ranorex Studio Leapwork BugBug Reflect Functionize Datadog	Sauce Labs BrowserStack TestRail LambdaTest

Table 5: All test automation tools considered during the selection process

Relevance analysis The tool identification resulted in 17 possible tools identified. The next phase, relevance analysis, identified which tools were relevant for the pilot. For each researched tool, I identified the name, general description, cost and relevance. Those tools that were not deemed relevant were eliminated.

Table 6 lists programmable tools. Table 7 lists no-code and low-code tools, which employ capture-replay or NLP-based techniques. Table 8 lists test execution and management platforms, which provide the infrastructure for test execution and management.

Programmable tools				
Name	**Description**	**Cost**	**Relevance**	**References**
Selenium	The industry standard browser automation framework with multi-language support. Tests are coded, or recorded with extensions. Mature system with strong community support. Not suitable for non-developers.	Free, open-source.	Eliminated.	selenium.dev
Playwright	A newer browser automation framework with multi-language support. Tests are coded, or recorded with extensions. Released in 2020, backed by Microsoft and gaining popularity. Not suitable for non-developers.	Free, open-source.	Eliminated.	playwright.dev/java
Cypress	End-to-end testing framework that runs inside the browser. Tests are written with JavaScript and TypeScript, or recorded with extensions. Released in 2017, and popular with modern frontend communities. Not suitable for non-developers.	Free, paid extras.	Eliminated.	cypress.io
Robot Framework	Keyword-driven testing framework using external browser automation libraries. Established in 2008, a strong ecosystem and an alternative to purely code-based frameworks.	Free, open-source.	Proceeds.	robotframework.org

Table 6: Overview of widely used programmable test automation tools

No-code and low-code tools				
Name	**Description**	**Cost**	**Relevance**	**References**
Katalon Studio	Test automation platform with built-in IDE. Tests are capture-replay or programmable. Ready to use out of the box.	Free and commercial tiers.	Proceeds.	katalon.com, docs.katalon.com
Mabl	Test automation platform focusing on visual test creation, NLP, and capture-replay. Advanced test logic can be programmed. Ready to use out of the box.	Commercial tiers.	Proceeds.	mabl.com
Tricentis Testim	Test automation platform focusing on low-code and capture-replay. Code snippets possible with JavaScript. Ready to use out of the box.	Commercial tiers.	Proceeds.	testim.io, help.testim.io
Ranorex Studio	Test automation platform with a built-in IDE. Tests are capture-replay or programmable. Ready to use out of the box.	Commercial tiers.	Proceeds.	ranorex.com
Leapwork	Test automation platform focusing on no-code, visual test automation using reusable workflows and blocks. Ready to use out of the box.	Commercial tiers.	Proceeds.	leapwork.com
BugBug	A lightweight test automation platform where tests are created with capture-replay. Optional JavaScript snippets supported. Ready to use out of the box.	Free and commercial tiers.	Proceeds.	bugbug.io, docs.bugbug.io
Reflect	Test automation platform with NLP and capture-replay. Ready to use out of the box.	Commercial tiers.	Proceeds.	reflect.run/docs, reflect.run
Functionize	Test automation platform with capture-replay test creation. Ready to use out of the box.	Commercial tiers.	Proceeds.	functionize.com, functionize.com/testing
Datadog	A monitoring platform with included test automation capabilities. Tests are capture-replay. Core purpose does not align with the pilot needs.	Commercial tiers.	Eliminated	datadoghq.com/docs, datadoghq.com

Table 7: Overview of no-code and low-code test automation tools

Test execution and management platforms				
Name	**Description**	**Cost**	**Relevance**	**References**
Sauce Labs	Provides the infrastructure to test and monitor externally created tests on different combinations of browsers and devices. Core purpose does not align with the pilot needs.	Commercial tiers only.	Eliminated.	saucelabs.com
BrowserStack	Provides the infrastructure to test and monitor externally created tests on different combinations of browsers and devices. Also offers visual and no-code testing capabilities. Core purpose does not align with the pilot needs.	Commercial tiers.	Eliminated.	browserstack.com
TestRail	Platform to create, track, and manage test cases and activities with reporting and integration capabilities. Core purpose does not align with the pilot needs.	Commercial tiers.	Eliminated.	testrail.com
LambdaTest	Provides the infrastructure to run externally created tests on various browsers and devices. Also includes built-in test creation capabilities. Core purpose does not align with the pilot needs.	Commercial tiers.	Eliminated.	lambdatest.com

Table 8: Overview of platforms for test execution and management activities

Tools that were eliminated lacked relevance to the pilot and were excluded from further analysis. These tools either did not support capture-replay, keyword-driven, or NLP-based test creation or their primary purpose extended beyond the needs of the pilot (e.g. comprehensive test execution and monitoring platforms). While the first round of elimination excluded almost half of the tools, the pool of candidates still contained nine tools. The first stage of analysis did not provide notable differences in these nine tools.

Requirement analysis For the requirement analysis, I examined these nine tools in more detail by reviewing their documentation or product pages. This analysis examined how the tools aligned with the outlined tool requirements. Table 9 lists these results.

Test automation tools evaluated against the six criteria							
Name	**Test creation**	**Flexibility**	**Cost**	**Integrations**	**Ecosystem**	**Maintainability and repeatability**	**References**
Robot Framework	Keyword-driven test creation. Requires some know-how about HTML and environments.	UI and API testing. Manual, scheduled and test runs possible, but require manual setup.	Free and open-source.	Supported, but require manual setup.	Established in 2008. Strong ecosystem with strong documentation and community support.	Supports reusing modular keywords, where changes need to be edited only once and applied across tests.	robotframework.org
Katalon Studio	Capture-replay and programmable test creation.	Cross-browser UI testing. API testing. Manual, scheduled and automated test runs.	Free limited tier and commercial tiers.	GitHub and Jira integration.	Released in 2015. Matured system and trusted by both big and smaller companies.	Modular test components. AI-powered test object identification is mentioned.	katalon.com, docs.katalon.com
Mabl	Tests are created with AI assistance or a test recorder.	Cross-browser UI testing. API testing. Manual, scheduled and automated test runs.	Commercial.	GitHub and Jira integration.	Released in 2017. Matured system with hundreds of enterprises as users.	AI-leveraged test healing is mentioned.	mabl.com
Tricentis Testim	No-code testing with reusable components. AI assistance with Testim Copilot.	Cross-browser UI testing and manual with manual, automated and scheduled runs.	Free limited plan and commercial tiers.	GitHub and Jira integration.	Released in 2014. Used by companies such as Salesforce and Autodesk.	AI and ML are mentioned to help identify elements.	testim.io, help.testim.io
BugBug	Capture-replay test creation with custom JavaScript snippets supported.	Chromium-based browser UI testing. Manual, scheduled or automated runs.	Commercial. Limited free tier, then $99/month and $299/month tiers.	GitHub integration. Jira integration is on the roadmap.	Released 2021. Work-in-progress. The basics are stable and new features are being developed.	Reusable components make test editing easier.	bugbug.io, docs.bugbug.io

Table 9: Test automation tools evaluated against the six criteria

Test automation tools evaluated against the six criteria							
Name	**Test creation**	**Flexibility**	**Cost**	**Integrations**	**Ecosystem**	**Maintainability and repeatability**	**References**
Ranorex Studio	Capture-replay or programmable test creation.	Cross-browser testing. Manual or automated test runs.	A pricing request is required.	GitHub and Jira integration.	Released in 2007. Matured application trusted by companies such as Cisco, Philips and Dell.	UI objects are stored in a central object repository.	ranorex.com
Leapwork	No-code, visual test automation using reusable workflows and components.	Cross-browser UI testing. Manual, automated and scheduled runs.	Pricing request is required.	Jira and GitHub integration.	Released in 2017. Trusted by companies such as Paypal and Visma.	AI is mentioned to help in test healing.	leapwork.com
Reflect	Capture-replay and NLP-based test creation.	Chrome and cross-browser testing depending on tier. API testing. Manual, scheduled and automated testing.	Commercial. Different tiers include \$212/month and \$530/month.	GitHub and Jira integration.	Started in 2019. Trusted by hundreds of companies.	AI is mentioned to help in test healing.	reflect.run/docs, reflect.run
Functionize	Capture-replay test creation.	Cross-browser UI testing. API testing. File export testing is mentioned.	A pricing request is required.	GitHub and Jira integration.	Released in 2015. Trusted by companies such as McAfee and Uponor.	Self-healing locators are mentioned.	functionize.com, functionize.com/testing

Table 9: Test automation tools evaluated against the six criteria (continued)

The requirement analysis provided deeper insights into each tool, yet they still shared similar characteristics. Almost all of the tools offered capture-replay and reusable test components. Many of the tools also advertised more complex capabilities such as test healing or artificial intelligence, but these claims could not be assessed further. Despite the similarities, the tools did contain some differences: for instance, the price varied significantly between tools. Furthermore, certain tools were more mature and other tools were considerably newer.

To truly examine the capabilities of the most promising tools, I evaluated them in practice. I selected four tools that were most promising and distinct from one another. However, the tools that were excluded were not necessarily dismissed due to a lack of quality. These tools were excluded for the following reasons: (1) the price was not economically feasible; (2) the tool required business-to-business arrangements or other practicalities that would delay this thesis; (3) the tool shared overlapping features with a tool that was already chosen.

The first tool I selected was BugBug, which is a relatively new but lightweight and streamlined platform for capture-replay testing. The second tool was Katalon Studio, which is a more mature system that supports both capture-replay and programmable testing. I chose Reflect as the third tool to examine its NLP-based test creation. Finally, I chose Robot Framework as the fourth tool to provide an alternative to paid platforms, while also examining its keyword-driven test creation.

5.3 Practical Evaluation

The goal of the practical evaluation was to assess the capabilities of BugBug, Katalon Studio, Reflect, and Robot Framework, and to evaluate each tool against the requirements and the planned testing activities. For each tool, this evaluation consisted of the initial setup or installation, which was followed by creating test suites for the generic version and one customer-specific release. Moreover, I also considered if any changes needed to be made to the current source code to support usage of the tool.

5.3.1 Tool Comparison

BugBug Table 10 assesses the tool against the requirements.

Requirement	Observations
R1: Test creation	Quick and intuitive. Dynamic data adds complexity.
R2: Flexibility	Supports execution methods. Lacks API testing.
R3: Cost	Justified for the given features.
R4: Integrations	Limited integrations, but actively under development.
R5: Ecosystem	New product which has not reached a mature state. Contains an active roadmap.
R6: Maintainability	Maintainable for small projects, but uncertain for this context.

Table 10: BugBug evaluated against the six requirements

I chose BugBug for its streamlined platform for efficient capture-replay UI testing. The tool provides the bare essentials for UI testing and allows tests to be executed locally or on the cloud with manual, scheduled and autokmated test runs. BugBug was easy to setup, and tests could be created in a matter of minutes. BugBug excels in simplicity, and it provides a very intuitive platform. Additionally, reusable components help create more maintainable tests.

However, when the number of tests increases, BugBug might turn unnecessarily cluttered. BugBug lacks API testing functionalities, and the tool is relatively new and under development. Moreover, to support dynamic values, the reusable components become much more complex than initially intended, possibly undermining the benefits of capture-replay. This reflects a known limitation with capture-replay approaches discussed in Section 3, where hard-coded values reduce maintainability.

Introducing BugBug as the test automation tool requires no additional changes to the software. While the system does not yet provide a direct integration to Jira, this could be done manually. The tool also supports future expansion into a pipeline. Additionally, the pricing for the platform is transparent, requiring a subscription of at least $99 per month.

To conclude, BugBug is streamlined and intuitive and provides the necessary capabilities for creating and managing capture-replay tests. The price of the tool is justifiable but the tool is still relatively new. A strong contender for capture-replay, but maintainability and scalability remain uncertain.

Katalon Studio Table 11 assesses the tool against the requirements.

Requirement	Observations
R1: Test creation	Intuitive, but dynamic data increases complexity. Supports both capture–replay and programmable testing.
R2: Flexibility	Supported.
R3: Cost	Transparent and reasonable for the given features.
R4: Integrations	Key integrations available.
R5: Ecosystem	Mature product with a strong community.
R6: Maintainability	File system structure improves maintainability.

Table 11: Katalon Studio evaluated against the six requirements

I chose Katalon Studio for its mature platform for UI and API testing. Additionally, the tool allows capture-replay and programmable testing. While the tool is comprehensive, it remained intuitive. Additionally, the platform provides a broader range of features. Tests are managed in a file system, which increases the maintainability of larger test suites.

This broader range of features also makes the initial learning curve steeper. Because of this the initial tests were not created as quickly. While creating tests for one version was straightforward, yet again introducing dynamic variables created unnecessary complexity that required manually changing test scripts, which diminishes the benefits of capture-replay. Additionally, the user interface of the tool was not as streamlined.

Introducing Katalon Studio as the testing tool requires no additional changes to the software. Test runs can be communicated to Jira via Katalon Studio's integrations, and future expansion into pipelines is supported. Katalon Studio has transparent pricing, and for the purposes of this pilot, the price would be either $84 or $168 per user per month.

To conclude, Katalon Studio as a test automation tool is comprehensive and provides both capture-replay and programmable testing functionalities. The learning curve of the tool is steep, and perhaps some of the features are outside the scope of this pilot. Additionally, dynamic data diminishes the benefits of capture-replay.

Reflect Table 12 assesses the tool against the requirements.

Requirement	Observations
R1: Test creation	Quick and intuitive for basic tests, but unpractical for dynamic data.
R2: Flexibility	Supported
R3: Cost	Not justified for the given features.
R4: Integrations	Key integrations available.
R5: Ecosystem	Less mature than Katalon Studio; newer than BugBug.
R6: Maintainability	NLP-based tests are uncertain, but the file system structure improves maintainability.

Table 12: Reflect evaluated against the six requirements

I chose Reflect as a possible candidate due to its capture-replay and NLP-based test creation. Both capture-replay and NLP-based test generation techniques proved to be quick. Additionally, the tool enables browser customization, enabling the tester to modify browser states as a key feature. The tool also contains capabilities for API testing. Reflect, too, structures tests in a file system which increases the maintainability of larger test suites.

While dynamic data is possible in theory, I noticed more time was spent on configuring tests to support this data instead of creating the actual test cases. Additionally, while NLP-based tests were quick, the maintainability and resilience to changes remains unclear for dynamic data and multiple tested versions. This uncertainty aligns with Section 3, which identified limited expressions with NLP-based tests.

Introducing Reflect to the current workflow requires no additional changes to the software. The platform allows the creation of tickets to Jira, but requires a person to manually trigger this on a failed test case. It allows easy expansion to a build pipeline. Reflect contains transparent pricings, either $212 or $530 per month depending on which functionalities are chosen. This pricing model is disadvantageous, however, because it does not allow tests to run locally, and has a limit for cloud test runs.

To conclude, Reflect as a test automation tool certainly has some benefits, but it does not align with the needs of the pilot. For the context of this pilot, creating tests with dynamic data is too complex, and the pricing model does not serve the pilot.

Robot Framework Table 13 assesses the tool against the requirements.

Requirement	Observations
R1: Test creation	Slower test creation, but a high level of customization enables dynamic data and multi-version testing.
R2: Flexibility	Features require manual setup or plugins.
R3: Cost	Free and open-source.
R4: Integrations	Available, but require manual setup or plugins.
R5: Ecosystem	Mature framework with a strong community.
R6: Maintainability	High maintainability, especially during larger projects.

Table 13: Robot Framework evaluated against the six requirements

I chose Robot Framework deliberately as an alternative to paid capture-replay platforms. Tests are created by combining human-readable keywords which represent actions on the user interface. Because of this, Robot Framework allows testers direct control over their tests, which increases flexibility and maintenance. Tests enable the usage of more dynamic data, and future multi-version testing is possible. The open-sourced plugins allow a wide range of testing, including UI and API testing.

However, this control comes at a cost, as it requires more initial setup time and some level of technical knowledge. Even though the keywords are intuitive, creating tests is not as quick as with capture-replay methods, as identified in Section 3. Because Robot Framework is an open-sourced tool and not a complete platform, the case company may need to invest additional time to build the same components such as integrations or scheduling, which come out-of-the-box with paid solutions.

Introducing Robot Framework as a test automation tool requires installing the required technologies, as well as slight modifications to the codebase. As certain parts of code contain nested HTML and brittle locators, introducing HTML id attributes, for example, is recommended. These attributes help with writing tests, as the tests can more easily locate elements and these elements are more resilient to changes in classes.

To conclude, Robot Framework as a testing framework is flexible and customizable, allowing developers direct control over the tests. This helps with maintainability and future expansion, which was uncertain for the capture-replay platforms. However, the tool is not plug-and-play and requires longer initial setup. Additionally, more time is spent on creating tests and developing the infrastructure that came out-of-the-box with paid solutions.

5.3.2 Tool Choice

After evaluation, the final decision came down between Robot Framework and BugBug. While BugBug is a lightweight, relatively new, fully capture-replay platform, Robot Framework is a more mature tool with better flexibility, extensibility and direct control over tests.

BugBug had several advantages, including robust capture-replay tools and a justified price model. However, the platform's limitations may become more evident during larger projects with dynamic data. Incorporating dynamic data into tests slightly diminishes the benefits of capture-replay. As tests are managed in the browser, an increased number of test suites might cause unnecessary clutter. While the tool could remain robust for the purposes of this pilot, its long-term viability is still unclear.

By contrast, when testers have direct control over their tests, they have more freedom. Robot Framework offers benefits to extensibility and maintainability, which works better for dynamic data and possible multi-version testing. On the other hand, test creation takes longer and requires some level of technical know-how, and the tool has a steeper leaning curve. In addition, certain features that came out-of-the-box, such as integrations and scheduling, require manual setup time.

Both tools could prove viable for the scope of this pilot. In the end, the case company and I chose to conduct the pilot with *Robot Framework*. The decision came down to its greater level of control over tests. As emphasized in Section 3, long-term maintainability is a key factor in successful automation, which is why we believe Robot Framework remains more effective over time, outweighing the ease of use of paid capture-replay platforms.

6 Pilot Implementation

This section explains the concrete pilot implementation and covers the most important design choices. It outlines the environment setup, the implementation of tests, the proof-of-concept automation system and its deployment. Finally, it outlines the pilot usage.

6.1 Test Environment

The Test Environment setup consisted of hosting the generic version locally, installing the required technologies, and creating a test database.

First, I hosted the generic version of the application on my local environment. The decision to test locally enabled the software to be updated iteratively as tests progressed, which proved particularly useful when adding HTML id attributes or fixing unexpected defects. As the tests target the user interface and do not directly interact with the source code of the product, no further details are required.

After the local environment was ready for testing, I installed Robot Framework. As discussed in Section 5, Robot Framework does not provide a complete platform for testing. Instead, I had to install the tool and the necessary libraries. This installation included Python, Robot Framework, and other packages for the tests.

The test database was an important consideration as test cases relied on dynamic data. The application uses a relational database. I created a test database and ensured that it contained enough data for each role, view and functionality. Unnecessary data was avoided in order to speed up testing, both for application data fetching and pre- and post-test database operations. The test database was created iteratively alongside test development, as the tests revealed what types of data were required. This process proved difficult at times, because functionalities required certain data-related conditions which were not defined in the access matrix document.

6.2 Developing Tests

Implementing the actual tests consisted of designing a pattern for the tests. This pattern determines how tests are created, which allows testing the user interface with dynamic values, different roles, and possibly multiple versions. Additionally, the tests need to ensure consistent and repeatable results with different test execution methods. Finally, I proposed the naming scheme for the HTML id attributes that are used in tests.

6.2.1 Tests

Keywords Tests are implemented with *keywords* in Robot Framework. Keywords are the building blocks of tests, and they represent actions, assertions, and other executable steps. In short, Robot Framework offers built-in keywords that enable browser automation and represent actions on the user interface. In addition, *user-defined keywords* can be created by combining existing keywords to create more complex sequences [29].

As discussed in Section 3, this structure increases the readability, modularity and maintainability of tests. Consider the following example: a keyword contains the login sequence. The readability is increased as the login logic is encapsulated by a single keyword "login". The modularity is increased because this single keyword can be used in all tests that require a login. The maintainability is increased because changes to the login sequence need to be applied only once.

Test Design Principles The user-defined keywords play a crucial role in the pilot for providing the necessary modularity to support dynamic data and possible expansion into multi-version testing. In practice, Robot Framework contains executable test cases that are located in .robot files, whereas .resource files are used to define these user-defined keywords [29]. Nevertheless, a key challenge in implementation is how these keywords should be structured and used in a way that supports usage across multiple roles, data, or versions. I approached this problem from multiple angles, and it quickly became apparent that the solution would contain a central directory of user-generated keywords that would reflect actions on the generic version of the software. Because customer-specific releases all shared the common core logic, these central keywords could be used when testing other releases, too. This is one of the benefits of the common core logic shared between versions, as discussed in Section 3. In contrast, other solutions would have had duplicated keywords, which decreases the maintainability.

This solution is displayed in Figure 9. Let us consider the following example, which is based on the simplified access matrix, shown in Table 2. The administrator user has permission to the administrator view. The central directory of generic resources contains the keywords necessary to test the administrator view. In practice, it contains a common.resource file which contains keywords that are not specific to any single view, for example the login. Then, each view is given their separate files, such as an adminview.resource, which contains all the required keywords to test an administrator view. For instance, these keywords could confirm whether a certain button is visible, whether the admin can edit a field, or whether HTTP errors are present after page load.

In this scenario, the highlighted files are of importance for creating the administrator view tests. The generic resources contain the keywords that represent all of the necessary actions on the user interface. Then, the generic version contains the actual executable tests. These tests are executed with the help of .resource files that setup relevant variables (e.g. URL, login credentials, database identifiers) for this environment and role. Using these variables, the test file calls the actual test keywords. This approach enables the sequences of actions to be written only once, which allows each role — and possibly a different release — to execute tests with dynamic variables. In addition, if proceeding with multi-version testing, each release-specific directory implements its own tests for overrides.

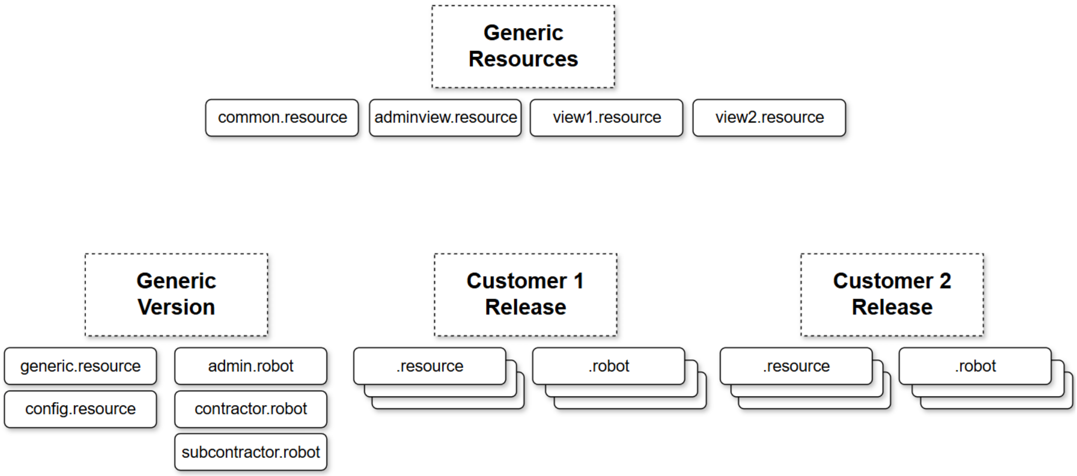

Figure 9: A central directory of resources outline keywords for interactions on the generic version. Version-specific test directions contain the actual executable files, role and version-specific variables

Remaining Challenges However, the solution still retains some level of duplication as executable role test files contain repetitive usage of keywords or test case setups and teardowns. I decided not to introduce a layer of abstraction at the role level, because it might reduce overall clarity. However, if repetitiveness of role test files becomes a greater concern in the future, restructuring these test files can minimize repetitiveness.

Moreover, another practical challenge that influenced how tests were created is the growing number of variables for each functionality and role. As the generic keywords are completely modular (e.g. without hard-coded variables), the amount of role-specific variables across all sequences of actions grows large. I did not manage to provide a meaningful restructuring that would effectively remove this problem.

6.2.2 Repeatability and Isolation

For a full test run, all test suites need to pass, as outlined in the test plan in Section 4. For the test runs to be repeatable, the test suites need to produce the same results. This means that the execution environment should remain unchanged across subsequent test runs. First, pre- and post-test environment setup is required. Before each test suite begins, the test database is restored to ensure that tests are executed with the same data. Then, post-test steps include restoring the database back to its previous state. In addition, as the application uses browser memory to store preferences, these preferences are cleared.

In addition to the testing environment, tests cases themselves should not depend on the execution of previous tests. This principle ensures that a failed test case does not cause subsequent failures. To address this, I implemented setups and teardowns also at the test case level, which ensures that individual test cases start from the same initial state. This, in practice, enables any single test case to be executed in isolation, though it increases the overall repetitiveness across role test files. It is worth noting that due to time constraints and the amount of dynamic data during the pilot, some cascading tests remain [9]. For example, inside one view a test case may create a data item that another test case later deletes.

6.2.3 Test Execution

Robot Framework allows the execution of a single test suite file, or a directory of test suites. This allows the tester to execute a full test run, or only role-specific test suites. In addition, the level of isolation and repeatability allows for tag-based test execution. In Robot Framework, utilizing tags allows the tester to only run a certain subset of tests [29]. I used tags to group test cases by the corresponding test item, for instance the administrator view.

There are several execution possibilities for the simplified access matrix example. The tester can execute a full test run that tests each role against each view. Conversely, they can also test a single role against each view. Tagged execution also enables test runs to only include tests specific to a single view.

While this approach enables testers to run only relevant tests on the local environment, I did not produce meaningful usage of it on the test server. However, tags still remain an important tool for the developer, making it easier when modifying or creating new tests.

6.2.4 Test Selectors

Finally, element selectors are a crucial aspect of user interface testing. Robot Framework allows elements to be selected by classes and attributes, among others. As noted in Section 5, the applications locators and classes are sometimes deeply nested or brittle, which can make developing tests more difficult. This proved to be true, which is why I decided to use HTML id attributes for element selection. They not only make the tests more resistant to future changes but also make the test scripts more readable.

The naming scheme is an important consideration when employing an id-based selector strategy to avoid unclear, repeated, or overly long identifiers. Because the context of an element can be seen from the surrounding HTML itself, the naming scheme follows an *action-object-id* structure. As database identifiers are already unique, I decided to use database identifiers in the HTML attributes for dynamic elements. Following a naming scheme for selectors is important, especially if multiple developers are creating tests in the future.

6.3 Test Automation System

6.3.1 Automation System

The automation system was outlined in Section 4, and it is deliberately lightweight. The automation system required installing Jenkins, setting up a Jenkins job, and creating an integration to Jira.

After installing Jenkins, it runs as a service and can be operated on in the browser. For the pilot, I created a runnable job that can be scheduled or manually launched. As mentioned above, no meaningful usage of tags was employed here. Instead, the whole test run for the generic version is launched. After a working implementation of the Jenkins job was complete, the next step was to create an integration to Jira for result reporting.

Reporting to Jira was possible in the three following ways: either utilizing Jenkin's Jira integrations, using custom scripts to Jira via their API, or by using Jira's Xray Test Management plugin. Because this plugin was commercial, and its purposes are more related to the management of testing activities, it was dismissed. Simple custom scripts were chosen over the Jenkins integration due to time constraints in order to keep reporting as simple as possible. If the proof-of-concept is deemed successful and Jenkins remains in use, its Jira integrations should be adopted instead of the more limited scripts.

6.3.2 Deployment

The deployment of the pilot consisted of creating a test server that hosts the generic version and deploying the testing system on it. Additionally, the tests were executed to gather metrics.

The deployment consisted of creating a test server (Microsoft Azure virtual machine with Windows OS), which proved to be time-consuming but does not require further explanation here. The testing tool and Jenkins were then installed on the test server. Next, the test automation system had to be deployed on the test server which required the tests to be version-controlled. The structure of the tests supported this as variables were not hard-coded and the configuration variables were able to be kept as secrets.

After installing and deploying the system to the test server, I ensured that it worked similarly to my local implementation. I implemented a weekly scheduled test run. Then, I gathered metrics for evaluation from my local environment and from the test server by executing ten consecutive full test runs, as well as six test runs with different roles and tag usage.

6.4 Pilot Usage

Figure 10 displays the final usage of the pilot, and its standalone test automation system. In practice, the testing system still requires manual involvement. However, instead of manually testing the access matrix document, the tester uses Robot Framework and Jenkins to automate acceptance-level regression tests.

The tester can execute tests on their local environment, or on the test server. The local environment remains flexible and enables tagged execution for test creation, management and debugging. Alternatively, the tester can launch official test runs on the test server. These test runs are orchestrated by Jenkins, and the test run results are reported directly to Jira for issue tracking.

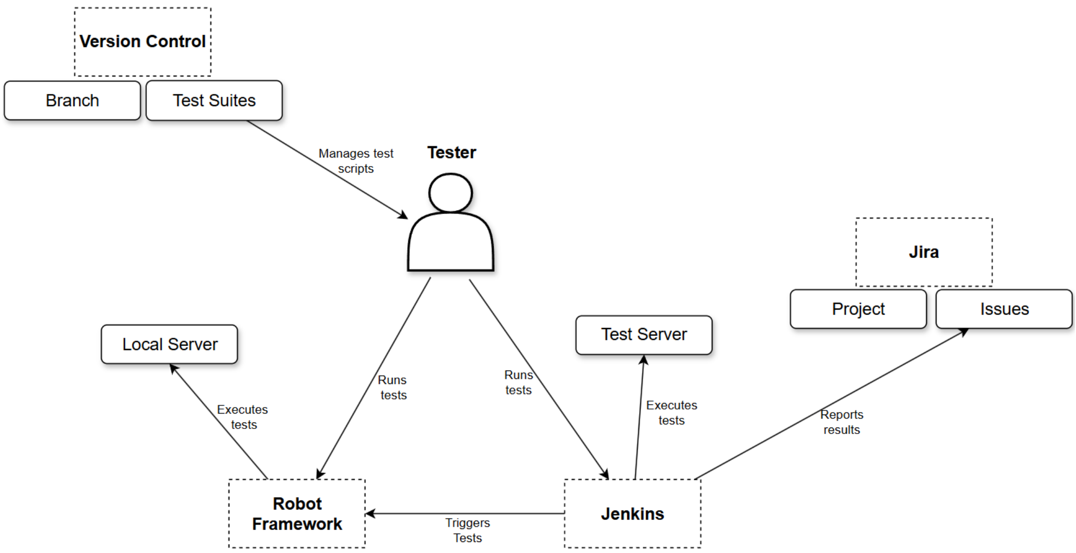

Figure 10: The testing system allows the tester to develop and debug tests on their local environment or run official and reported test runs on the test server

7 Pilot Evaluation

This section evaluates the success of the pilot by comparing the outcomes against the initial pilot goals. The evaluation remains closely tied to the case company and the pilot itself, whereas broader findings and connections to the background literature are addressed in Section 8. The evaluation is based on analysis and metrics, and each goal is evaluated. It begins by assessing the success of identifying and automating the most repetitive steps, followed by an evaluation of the standalone test automation system and its feasibility. This section concludes by outlining potential future development paths.

7.1 Automating Repetitive Tests

The first goal of the pilot was to identify and automate the most repetitive parts in testing to reduce manual efforts. Ultimately, evaluating this goal comes down to determining whether the target for testing was justified, and what percentage of the planned tests was successfully implemented.

The choice to employ higher-level tests and test the user interface instead of preferring approaches, such as unit testing server code, linked back to the purpose of the testing effort. For the case company, the purpose of the test automation effort was to reduce manual testing efforts, while balancing it with quality assurance and efficiency. In addition, as the case company's product is not safety-critical, achieving code coverage and testing every internal path was not as essential. Instead, testing the user interface at the acceptance level provided a sufficient amount of quality assurance for the end user. Choosing this approach for the initial testing efforts was justified.

In particular, the user interface was tested with the access matrix document that contains the requirements for the end users. The document contains specifications on which roles are permitted for which views and functionalities. When directly translated to tests, the access matrix document compasses around 430 individual test cases across 6 different users. These test cases vary from simple navigation steps to more complex interactions on the interface, further increasing the repetitiveness of the tests. The choice to test the interface specifically by translating the access matrix document to tests was justified.

In the end, around 45 out of 430 individual test cases were not implemented as the features were either not yet developed, or they required refactoring that were too large for the scope of the pilot. Around 90% of the access matrix was translated to tests. No test cases were left unimplemented due to difficulties related to the actual tests.

Furthermore, around 10 to 20 distinct defects were discovered and fixed. These defects were either unintentional bugs, or larger conflicting decisions between the source code implementation and specifications. These conflicting decisions were resolved in meetings. This totals to about 2% to 5% of test cases containing defects.

Figure 11 displays the implementation status of the access matrix items. To conclude, up to 90% of the planned tests were implemented. Around 5% of these planned tests contained defects which were successfully fixed. Around 10% of the planned tests were not implemented.

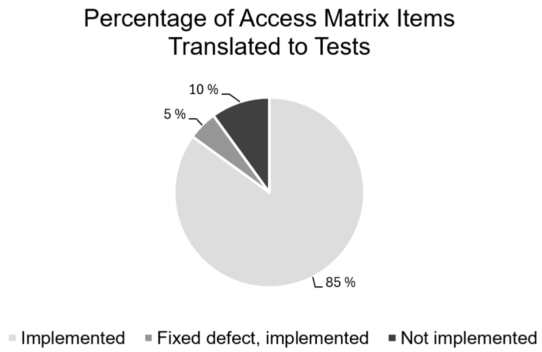

Figure 11: Implementation status of access matrix items

The testing effort revealed valuable insights. Creating the automated tests uncovered existing inconsistencies in the source code, which were resolved. This resulted in refining both the product and the access matrix document. Moreover, the repetitiveness of role-based testing highlighted concerns on whether the product could benefit from other approaches to access control. For instance, an idea to simplify access control surfaced. Lastly, creating test cases from scratch revealed certain data preconditions that were not specified in the documentation.

To conclude, the first goal of the pilot can be considered completed. The repetitive parts that were automated were justifiably identified, as testing the user interface with the access matrix document directly ensures quality for the end user. Up to 90% of the access matrix document was translated to tests, and the unimplemented test cases were either not yet developed or required future refactoring. The test automation effort not only improved the speed of regression testing the user interface, but also improved the product and its specifications.

7.2 Standalone Test Automation System

The second goal of the pilot was to establish a standalone test automation system that supports future expansion. This involves evaluating the system and its alignment with the test plan, which provided the following description of the testing system:

> In summary, the chosen testing approach should conduct *regression tests* at

> the *acceptance level*. Regression tests are used to ensure that the software continues to conform to the role-specific behavior defined in the access matrix document. Because tests are derived directly from documentation, they are *specification-based* tests. The conformance is tested through *functional tests* that target the UI. Therefore the selected automation tool needs to support UI testing through *capture-replay*, *NLP-based*, or *keyword-driven* test creation. This enables an automation setup where the tester can run tests either locally or on the test server. Local testing should support flexible test creation and modification, while test server is reserved for formal test runs where results are automatically reported to Jira.

The access matrix document was successfully translated to regression tests at the acceptance level, as justified in Subsection 7.1. Unimplemented test cases were not the result of limitations in the automation system, but rather features that were not developed yet or were scheduled for refactoring.

The pilot implements tests for the generic version of the software. As outlined in Section 6, the test scripts revolve around a central directory of keywords. These keywords represent actions on the generic version. The pilot provides these keywords as well as test suites, variables, and configurations for testing the generic version.

The pilot addresses the major drivers of technical debt, discussed in Section 3, such as lack of modularity and maintainability. It also addresses the previous test automation efforts that had failed due to unmaintainable test scripts. With this approach, the expansion into larger coverage or customer-specific releases is supported.

The pilot reached a stage where it functions as a standalone system. As Section 6 outlines, the system allows testing on the local environment. Additionally, as a proof-of-concept, a Jenkins job was implemented on a test server to coordinate test runs. Results of these test server test runs are successfully reported to Jira. While the approach remains limited, it provides traceability between test execution and issue tracking. This demonstrates that the pilot can be used in practice. Furthermore, the system leaves room for future expansion due to its test script structure and lightweight automation.

7.3 Feasibility of Test Automation

For this pilot, feasibility refers to overall viability, maintainability, repeatability, performance, and cost. These factors are evaluated by analyzing the pilot and gathered metrics.

7.3.1 Viability

Subsection 4.3 outlined the pilot's risks. The viability of the approach is evaluated by outlining which risks materialized, were mitigated, or remained non-observed. The review of these risks determines whether incorporating test automation in this context is viable. If a majority of the risks had materialized, the current test approach would not be viable.

Product risks Product risks relate to the quality of the software and the testing process. Two risks materialized but were minor, two were successfully mitigated and the rest were not observed.

Materialized risks include tests revealing a moderate number of defects, and conflicts between the access matrix document and the decisions in code. These defects and conflicts were either fixed or resolved in meetings, and even though minor, ultimately slowed down the test creation. On the other hand, these risks were expected to materialize, and resolving them helped refine the product and its specifications. Finally, the fact that the test data was not representative of real world data was not observed for the duration of the pilot. Instead, the risk transformed into complexity issues with the amount of dynamic data.

Mitigated risks included test flakiness especially early on. Test flakiness was mitigated by careful usage of keywords and asynchronous waiting. Another risk, the maintenance burden from test scripts not scaling, was successfully mitigated by choosing the modular approach that was outlined in Section 6. Although the future scalability cannot be guaranteed at this stage, the approach demonstrated scalability at least across six different roles and over 400 test cases. The risk of the test server or database disrupting reliable executions was successfully mitigated with the help of environment setups and teardowns.

Risks that were not observed included the pilot creating false confidence. Changes to UI requiring frequent changes to test scripts remained non-observed, as UI changes were not present during the pilot. Tests being not fully replicable due to version or environment differences was not observed, as the pilot was conducted only on the generic version. However, the local environment and the test server produced similar results reliably.

Project risks Project risks relate to the management of testing activities. Only one project risk materialized, some risks were mitigated, but most were non-observed as due to the short observation window.

The risk that materialized was test execution time growing significantly as coverage increased. This risk turned out to be real, especially when complex functionalities were tested. While the execution time is still at a manageable level, the feedback loop of testing is slightly reduced, and performance remains an important consideration if the coverage is increased further.

Mitigated risks included the knowledge transfer gap that was mitigated by defining and delivering the pilot deliverables. While this knowledge transfer cannot be guaranteed with certainty, the deliverables support adoption. Additionally, having no long-term strategy was mitigated by the pilot providing and evaluating future development paths for testing. The risks related to the selected test automation tool were mitigated by researching and evaluating tools as well as creating a comprehensive testing structure. While these risks were not mitigated with full certainty, they were acknowledged during the pilot and were directly addressed.

Other project risks were not observed due to the limited scope of the pilot. As these risks relate more to the testing activities at the organizational level, it was not possible to observe them. Nevertheless, the future development paths provided by this pilot can aid in mitigating these risks in the future.

7.3.2 Maintainability

Maintainability is one factor of feasibility. As discussed in Section 3, a maintainable testing system reduces technical debt. The direct control over tests proved crucial for achieving maintainable tests, as observed in Section 5.

The developed solution to testing, displayed in Figure 9, enables updates to be edited only once. Updates to tests then propagate across all roles, and possibly across customer releases. This behavior was validated during the pilot, as tests were refactored or edited. While this maintainability was not observed across multiple customer releases, the solution scaled to 6 roles and over 400 test cases.

Utilizing HTML id-attributes for test selectors was identified as necessary during the practical evaluation process, as discussed in Section 5. This choice proved appropriate, because it simplified test creation, made tests more readable, and enhanced test stability against attribute changes in UI elements.

While these design choices improved maintainability, typical challenges with higher-level testing still exist. Because tests interact with broader components and data dependencies, the tests turned more complex to manage. For instance, the maintainability of tests was reduced by a growing number of test variables. The modular testing approach required testers to have explicit knowledge of the database identifiers and elements. These identifiers needed to be manually gathered either from the database or from the browser, which introduces overhead and decreases the clarity and maintenance of test scripts.

Furthermore, maintaining and updating a test database introduced additional overhead, which diverted resources from testing and quality assurance, which is typical for higher-level tests.

Each role file executes all the tests for that role. The test files also execute setups and teardowns to support a modular approach. This repeated usage of shared keywords, setups, and teardowns makes them less maintainable. This was noticed already during the pilot, but could be mitigated by refactoring. Role files were purposely kept simple early on for flexibility, but this increased overall duplication between files.

To conclude, testing remained maintainable at the observed scope. The modular approach to testing scaled to 6 roles and over 400 test cases. Typical higher-level maintainability issues still exist, such as an increased number of variables and test file duplication, which can be refactored but require ongoing attention.

7.3.3 Repeatability

Non-repeatable tests undermine test automation because they become counterproductive by generating false failures and incurring unnecessary debug time. To evaluate repeatability, I measured the pass-rate consistency of ten consecutive test runs.

The pilot achieved 100% consistency across all executed test cases, which indicates repeatability when executed consecutively in the same context. While false failures from test flakiness was an issue early on, they were successfully mitigated by the following mechanisms. Firstly, database resets and browser memory clears provide a consistent context for tests. Secondly, tests were created with explicit wait conditions instead of sleep-based waits.

Some occasional failures were observed that appeared to originate from Robot Framework itself. These failures occurred inconsistently but caused a single test suite to fail. Their exact cause could not be determined, and they were not observed during the ten consecutive test runs, either.

While tests demonstrated repeatability, they were measured under consistent and controlled conditions. These observations do not guarantee repeatability across different environments or releases. Furthermore, how the system responds to transient issues, such as network glitches, remains unclear.

7.3.4 Performance

Test execution performance fundamentally impacts testing. Extended execution times discourage developers from frequent test runs, increase feedback loops that require developers to context switch between tasks, lead to scheduling tests overnight, or make debugging more cumbersome. While longer test runs are not inherently problematic, they can separate testing from development, which was identified as a common pitfall in Section 2.

The performance of tests was measured with test execution time. These metrics were gathered from a local virtual machine running Windows and a more powerful Azure-based virtual machine, also running Windows. Test runs were executed on both environments, and their results are listed in Table 14.

In total, 12 test runs were executed, with each environment having six test runs. The six test runs were divided in the following way: tests were run either on all roles, on a complex role (e.g. most permissions), or on the simplest role (e.g. least permissions). Additionally, tests were either run with all test cases or with navigational tests only.

Test run	Local	Test Server	Speedup
All roles	16min 35s	11min 20s	32%
Complex role	4min 25s	2mmin 30s	43%
Simplest role	50s	40s	20%
All roles, navigation	3min 25s	3min 20s	2%
Complex role, navigation	35s	30s	14%
Simplest role, navigation	16s	15s	6%

Table 14: Comparison of execution times between local and test server

This revealed a pattern in performance. The testing environment demonstrated a significant speedup, especially for more complex test runs. This speedup ranged from 20% to 43%. On the other hand, navigation-only test runs remained relatively consistent between the environments, with only 2% to 14% improvement.

Table 15 lists the factorial speedup of navigation-only tests when compared to full test runs. For the local environment, the navigation-only tests were up to 7.57 times faster than full runs. The test server saw a speedup of up to 5.00 times.

Test Run	Local	Test Server
All roles	4.85x	3.40x
Complex role	7.57x	5.00x
Simplest role	3.12x	2.67x

Table 15: Comparison of navigation-only and full test runs on local and test server

These performance patterns impact deployment and infrastructure. They also influence the future development paths for the pilot regarding testing strategy and increasing coverage. This performance reflects the typical relationship between higher-level tests and execution times. The modular and repeatable test architecture also contributed to increased execution times, primarily due to redundant test steps or repeated setups and teardowns. Furthermore, database restores between test suites incur additional delays.

The performance remained suitable for the scope of the pilot. However, as simple interactions in the UI are already covered by the access matrix tests, increasing coverage further requires testing more complex sequences. This is likely to increase execution further and challenge the feasibility of the approach. These uncertainties highlight the need to balance between fast, focused tests and broader, slower coverage.

7.3.5 Cost

The pilot did not incur any direct costs. Robot Framework remains free and open-source, which was identified in Section 5. The automation system was developed around Jenkins, which is also free. The ticketing system Jira, while requiring a subscription, is already in use and paid for.

Other tools included upfront, monthly costs. These costs ranged from $100 to $20000. Furthermore, some of the researched tools included additional constraints such as limited user counts or cloud test runs. These tools were not economically feasible for this pilot, and utilizing the open-source and free Robot Framework proved effective.

Robot Framework incurred more development time due to several factors. Firstly, time was spent on implementing the integrations and infrastructure that came out-of-the-box for other tools. Secondly, as Section 3 identified, the capture-replay tools could have provided faster test creation. For the case company, this increased time to develop tests directly translates to increased costs, On the other hand, the relationship between increased development time and modular, maintainable tests that require less maintenance time in the future remains ambiguous.

Due to a limited scope, a direct translation of spent time and cost is not available. A realistic timeframe for a comparable implementation could require two to three months of development time, however. Most of this time was spent on developing individual test cases, whereas the proof-of-concept automation system was developed in less time.

The test server incurs monthly costs. However, this setup would have been necessary irrespective the tool choice, as the application and test database require hosting. While the costs of a test server can be influenced by several factors, these costs fall outside the scope of this thesis.

7.4 Future Development Paths

The fourth and last goal of the pilot was to identify future development paths for the pilot. The following part provides an actionable roadmap that consists of justified and concrete next steps. These steps were identified during the planning, implementation, and evaluation phases.

0-2 months Focus on improving the pilot and enabling smooth adoption. Emphasis is placed on developing the remaining test cases, outlining organizational processes and minor refactoring.

- **Translate remaining access matrix steps to tests**. Implement the remaining 10% of steps that were not translated to tests during the pilot. These items first require specifying or implementing the required features before adding tests. Implementing the remaining steps finishes the coverage and strengthens the traceability. Furthermore, implementing the remaining steps can surface the remaining inconsistencies.
- **Eliminate remaining cascading test cases**. Refactor remaining test cases to not depend on the data created by other tests. This increases the level of isolation and decreases the possibility of one failure causing many subsequent failures. These improvements align with the design of isolated and repeatable tests.
- **Refactor the handling of variables**. The growing number of variables was a practical issue encountered during the pilot, and it impacts the maintainability of testing. Refactor or further centralize the handling of variables to reduce the amount of variables and cognitive load for creating tests. Furthermore, improve the quality of the test database and ensure data relationships are consistent.
- **Reduce test suite duplication**. Refactor test suites to reduce the number of duplicated steps. This reduces unnecessary duplication, improves readability, makes editing multiple test suites faster, and makes tracing between tests and the access matrix easier.
- **Clarify ownership and processes**. Define the responsibilities of testing, including at least who writes and maintains tests, who runs test suites, when test suites are run (e.g., on demand, in scheduled intervals, or per release), and who fixes failures. Furthermore, determine whether some of these responsibilities could be implemented with build and test pipelines.
- **Implement a playbook**. Consider implementing a playbook that dictates how testing is conducted at an organizational level. The pilot deliverables can be used as a starting point. A playbook on testing removes the ambiguity for the tester and helps make tests more aligned with the organization-wide approach.

2-4 months Make targeted adjustments to improve operability. Consider parallel execution and improve the maturity of the automation system. Furthermore, decide on the approach to customer-specific releases.

- **Decide whether to explicitly test each customer release**. As noted in Section 3, explicitly determine whether it suffices to only test the generic version and the shared core, or whether test suites should also test specific customer releases and overrides. Introducing multiple releases to testing incurs additional overhead, but subsequently improves the quality assurance on these releases.
- **Shorten wall-clock time**. Consider the parallel execution of test suites and test cases. Implement systematic usage of tags on the server. This decreases feedback loops, makes debugging easier, and maintains viability as coverage is increased.
- **Improve the automation system**. Continue improving the proof-of-concept automation system. Consider replacing the Python scripts with Jenkins' Jira integrations. Consider other alternatives, such as Github Actions, for automation. As noted in Section 3.2, the effectiveness of test automation depends on clear processes. The automation system should evolve to align with these processes.

4-6 months Broaden the testing strategy by complementing higher-level test suites with lower-level tests. Establish CI pipelines that improve the shift-left principle.

- **Test at multiple levels**. Determine whether to continue on with more expensive higher-level tests or implement lower-level tests. As identified in Section 2, introducing lower-level tests requires upfront time investments, but pays off through faster feedback loops, improved fault detection, reduced maintenance needs, and better control for flakiness.
- **Implement build and test pipelines**. Establish CI pipelines that automatically build and run tests on pull requests and releases. Define exit criteria and reruns to combat unstable test runs. This creates regular, repeatable test execution that shortens the feedback loop and tightly connects testing with development activities.

8 Discussion

This section discusses the findings from the pilot and the thesis as a whole. It begins by summarizing the evaluation results, then connects the findings to the research question and related literature. Then, the practical recommendations, the significance of the work, the limitations and directions for future research are discussed.

8.1 Summary of Results

This thesis aimed to address the following overarching research question:

> *How can test automation be introduced into existing systems to reduce manual testing efforts, while also ensuring maintainability and support for future extension?*

I implemented a pilot to answer this question. The pilot had four goals, which all serve to answer the research question from different perspectives. Due to a limited scope, the pilot was conducted on the generic version of the software and customer-specific releases or full build pipelines were not included. The evaluation of the pilot indicated that the pilot's goals had been achieved, although some concerns still remain relevant.

The first goal focused on identifying and automating the most repetitive steps of manual testing. For the case company, choosing to automate acceptance-level regression tests on the user interface proved to be justifiable as it yielded the greatest reduction in manual workload. Furthermore, these steps were high-impact and user-facing areas of the software. Up to 90% of the intended steps were translated to automated tests.

The second goal focused on establishing a standalone test automation system. While parts of the system remain as proof-of-concept with limited flexibility, the pilot resulted in a repeatable and modular system for executing automated tests on the generic version. The resulting automation system aligned with the outlined design in the test plan in Section 4. Furthermore, the structure of the scripts also supports future expansion into larger coverage or customer-specific releases.

The third goal focused on evaluating the feasibility of the chosen approach. The pilot demonstrated that the chosen tool and the approach to testing was viable for the case company. No major risks materialized during the pilot. The approach demonstrated maintainable and repeatable testing that scaled across six different roles and over 400 test cases. Nevertheless, some concerns still exist. These include typical challenges with higher-level testing, such as increased execution times and maintenance. Moreover, the pilot did not yet observe risks related to organizational processes.

The fourth goal focused on identifying concrete development paths for the pilot and the case company. The identified development paths indicate that the pilot had been successful. These paths were either slight improvements to the existing pilot, or organizational decisions regarding processes, CI, and future testing strategies.

Beyond the primary findings, several practical observations were noted: the number of found defects was modest; tests improved product and documentation quality after addressing surfaced inconsistencies; tests revealed several hidden preconditions in data; tests revealed potential areas for future updates; maintainability issues were primarily caused by a complicated database and the number of variables; moving to stable HTML id-selectors improved robustness; and that flaky tests were fixed by using wait keywords instead of sleep-based timing.

8.2 Interpretations

Incremental and Context-Aware Introduction The pilot demonstrates that an *incremental and context-aware approach* is an effective way to introduce test automation. The pilot acted as the first step towards automation, and it focused on building an expandable foundation. The scope of the pilot was deliberately limited to ensure that the feasibility of the approach could be evaluated before committing to larger-scale automation. This reduced risks and brought practical insights for future test automation efforts. Moreover, aligning the scope with the case company's context enabled the selection of a suitable tool, appropriate test types, and test design principles that match the characteristics of the product.

Starting automation with high-value, repetitive, and well-specified test cases proved beneficial. In particular, the pilot translated a frequently tested access-matrix document into acceptance-level regression tests that meaningfully reduced manual work. The chosen test cases were traceable as each test case was explicitly documented, and ambiguities and inconsistencies in the source code were easily identified and fixed. In retrospect, the number of tests could have been slightly narrower to reduce excess time on individual test cases while still uncovering relevant challenges and validating the approach.

Proof-of-Concept and Feasibility The proof-of-concept automation system demonstrated that introducing test automation gradually was both feasible and valuable. The pilot produced a working system for test execution and reporting that fit into the case company's workflow (e.g. ticketing system) without unnecessary complexity or dependencies. This proved the feasibility of the approach without major upfront investments. However, the system alone is not yet sufficient for large-scale use, as its effectiveness still relies on processes and CI pipelines. This approach reflects on the incremental adoption, where feasibility was validated in a controlled setting before extending automation to organizational processes.

The tool choice benefited directly from the context-aware planning and tool research. Because the evaluation considered context, long-term maintainability and future use into consideration — rather than prior familiarity — the selected tool was appropriate for the products complexity and evolving testing needs. This aligns with the literature, which emphasizes that programmable and keyword-driven tests are well-suited for highly dynamic and modular UIs. Nevertheless, the pilot could not conclusively state whether alternative tools would have yielded better or worse outcomes.

Although the initial research strictly considered the monthly costs of different tools, the time spent on creating and maintaining tests also proved to be a significant factor. Previous literature emphasizes that test automation often involves substantial upfront costs. Even though Robot Framework was free, using it required considerable time investments for creating and maintaining scripts. This reinforces the idea that cost evaluation should extend beyond initial tool costs. Furthermore, it emphasizes the benefits of incorporating test automation gradually.

Challenges and Research Gaps While the approach demonstrated its feasibility, common challenges still exist. For instance, utilizing higher-level tests lead to challenges — such as higher execution times, increased complexity, and limited insight into root causes — that were identified with the test pyramid in Section 2. While these concerns remained manageable during the pilot, their impact on future efforts needs to be addressed. The testing strategy should remain proportionate to the product's characteristics and requirements.

Moreover, the pilot highlighted a broader gap in research. While literature often focuses on introducing test automation into new development projects, fewer examine how it should be incorporated into existing systems. The findings from this thesis suggest that organizational constraint and product complexity significantly shape what kinds of tests and processes are most effective.

Some of the observed complexity originated from the product itself. The large relational database and multiple user roles made test creation more time-consuming and revealed possible areas for future redesign. This highlights that the context and characteristics of the product should dictate the selected tooling and approaches.

8.3 Practical Recommendations

This part presents the practical recommendations from this study. It outlines actionable guidance for each phase of incorporation, independent of the design decisions made specifically for this pilot.

Preparation and Planning Do begin by mapping out current testing practices to reveal bottlenecks and repetitive manual test cases. Examine the broader project context, including system criticality, safety requirements, and whether automation is introduced into a new development project or an existing system. Define a long-term goal for test automation, and break it down into smaller steps that can be implemented incrementally. Begin with the most high-value, repetitive, and well-specified test cases. Clearly define the intended approach to automation (e.g. unit, API, UI testing) and acknowledge the effects on long-term maintainability and cost-efficiency.

Do not initiate test automation efforts before understanding what should be automated and why. Do not aim to automate everything. Do not assume that early automation will reduce overall testing time.

Tooling Do evaluate tools and frameworks based on how they align with the outlined automation goals. Consider factors such as cost, maintainability, scalability and integrations with existing systems. Tool selection should be evidence-based to best support the identified approach and future expansion. The tool should not be chosen only for short-term usability, instead it should also support future maintenance needs and product complexity. If possible, conduct a practical comparison to validate the tool's compatibility with the chosen approach and software.

Do not choose tools solely based on popularity or prior familiarity. Do not underestimate the long-term costs, even if the tool itself is free or inexpensive.

Implementation Do build a proof-of-concept test automation system that demonstrates technical feasibility. Establish test design principles that align with the long-term goals of automation, emphasizing modularity and isolated tests that create a maintainable foundation. Integrate the proof-of-concept with key workflows such as ticketing systems or existing pipelines. Carry out possible course corrections related to tooling, testware or testing processes before scaling further. Maintain a testing playbook that describes how tests are created, executed and maintained to ensure consistency. Focus on automating tests that provide clear value, and validate the feasibility of the approach in a controlled scope before integrating it across broader processes.

Do not neglect test design principles or consistency. Do not implement beyond the proof-of-concept stage too early. Instead, keep the scope limited to enable course corrections or alternative approaches. Do not prioritize initial speed over stability, as unstable tests lead to long-term maintenance overhead.

Process Integration Do define clear processes and responsibilities, including who maintains tests, who investigates and fixes failures, and when test suites are run. Establish an organizational playbook that summarizes the necessary roles and routines. Ensure management and development share a common understanding of test automation.

Do not treat test automation as a separate activity from development. Do not solely rely on the technical feasibility, as organizational commitment and processes are equally critical.

Expansion Do refactor existing implementations to improve maintainability and stability as the system evolves. Continue to expand the automation system in alignment with the long-term goals defined during planning. Gradually extend coverage and strengthen integrations, for example by introducing continuous integration. Use practical insights from previous increments to refine strategies and priorities. Guide expansion by measurable and valuable outcomes, not just by increasing overall coverage.

Do not allow tests or testing practices to become outdated due to product changes. Do not leave the automation system isolated from workflows. Do not assume increasing coverage always improves quality. Do not scale automation beyond the maturity of the organizational processes.

8.4 Significance

The study provides a concrete case study on incorporating test automation in resource-constrained and non-safety-critical environments. It provided the case company with the foundations of a test automation system and directions for further expansion. The study highlights a practical approach to test automation, which emphasizes long-term vision, prioritization of high-value and repetitive tests, and incremental incorporation. Furthermore, it outlined and compared current state-of-the-art UI testing tools and identified Robot Framework as a strong candidate for modular, flexible and maintainable UI testing. The case study shows how a well-planned automation effort meaningfully reduced manual work and which challenges remain.

From a research perspective, the study applies software testing principles to practice. The study addresses a gap in research, which primarily focuses on introducing software testing to new development projects, by analyzing the results from a complex and existing system. The study confirms that theoretical trade-offs (e.g. differences in higher-level and lower-level testing, differences in automation tools) are observed in practice. Furthermore, the study highlights how context, strategy, and incremental adoption influence the approach to test automation — particularly for existing systems. In doing so, it also highlights the need for future case studies with longer observation windows that also examine organizational processes.

Ultimately, this thesis confirms the theoretical trade-offs but also demonstrates the importance of a context-aware and incremental approach to test automation.

8.5 Limitations

The following part outlines some of the limitations that impact this study. While these limitations do not undermine the core results, broader contexts, longer observation windows and more comprehensive metrics are necessary for more generalized results.

Single context case study This study examined a non safety-critical domain with multiple user roles and complex relational data. The product contains a generic version and multiple customer-specific releases with their own overrides. The design, implementation, and tool research were influenced by concerns, economic considerations, and prior experience. As a result, alternative approaches could produce comparable results in different contexts. A single context limits the generalization of results, thus making the findings most applicable to similar organizations, domains, or constraints. Finally, as the design, implementation and evaluation were all conducted by the author and the case company, and the findings were neither externally reviewed or compared to external sources, some level of bias may have influenced the results.

Short observation window The study covered planning, tool research, implementation, and evaluation, but was conducted in a short observation window and for the generic version only. As a result, the concrete evidence for supporting multi-version testing remains out of scope. Likewise, CI and organizational processes — which were identified as key enablers of effective test automation in Section 3 — were not observed. While the pilot was successful and provided insights within the planned scope, organizational adoption and CI are topics for future evaluation. Similarly, the long-term effects of choosing to automate higher-level tests were not evaluated. While the initial performance impacts and increased maintenance were mentioned, the overall defect yield or increased costs between levels of tests were not discussed.

Metrics and indicators Given the scope and limited observation window, some outcomes rely on indicators instead of concrete metrics. While the performance, repeatability and number of translated tests was measured, the maintainability or reduction of manual effort were not captured. Likewise, the economics of testing could not be analyzed comprehensively. For example, while the chosen tool was free, it may have incurred more development time which translated to increased costs. On the other hand, the increased initial development time could improve the maintainability of the system thus decreasing technical debt and future costs. A longer observation window and more comprehensive methods could have allowed for more precise cost-benefit analysis.

8.6 Future Work

While the observed limitations do not undermine the core results, addressing these areas with future research could enhance the significance of the results. Moreover, such research could contribute to closing the identified research gap regarding test automation and existing systems.

Future research could focus on how context, organizational processes, and tool choice influence the success of test automation — particularly when comparing new development projects and existing systems. Moreover, future research could explore which parts of systems provide the biggest return on investment, how to distribute test techniques and levels, and which organizational factors support the long-term success of test automation in these contexts.

Broader contexts Replicating and extending the approach to different organizational and technical contexts could provide meaningful insight into how organizational constraints, system criticality, and business needs impact the approach to automating tests. Furthermore, these insights could clarify how the given context shapes what should be tested, what tools are appropriate and which test techniques should be used.

Longer observation window A longer observation window would enable the assessment of cost-effectiveness, maintainability and organizational adoption over time. As noted in Section 3, management and organizational factors are the make-or-break of test automation. Thus understanding which processes, strategic decisions, or CI techniques aid in successfully incorporating test automation is significant.

Metrics and quantification Future research could also apply more comprehensive and standardized metrics to quantify maintainability, reduced manual work, and cost-effectiveness over time. By comparing these concrete metrics to existing theory, the effectiveness of different test distributions or tool choices could be evaluated in practice.

9 Conclusion

This thesis analyzed how test automation could be introduced into existing software systems to reduce manual testing efforts, avoid maintenance issues, and support long-term scalability. To address this question, the thesis conducted a pilot for a case company. The pilot identified what to automate, researched suitable UI automation tools, translated a specification document into test cases, developed a proof-of-concept automation system, and evaluated its outcomes.

The pilot found that an incremental approach enabled early validation and course correction without major upfront costs. Starting with high-value, repetitive, and well-specified test cases provided immediate benefits by meaningfully reducing manual work. Furthermore, the proof-of-concept automation system demonstrated feasibility without introducing complex integrations or dependencies. Establishing test automation within a context-aware, long-term plan proved essential to the pilot's success, because it supported the selection of appropriate tools and test design principles that could scale and support future needs.

These findings emphasize that introducing test automation is technical and strategic, and effective incorporation depends on selecting high-value test cases that provide the highest return on investment. The scope of the initial automation should be broad enough to identify potential challenges but also narrow enough to avoid unnecessary costs and complexity. The findings illustrate that while higher-level tests were valuable for reducing manual workloads, they also introduced expected tradeoffs such as longer execution times, increased maintenance needs, and the lack of visibility into root causes of failures. This highlights the importance of balancing different testing approaches. Furthermore, the study identified a gap in literature regarding how test levels and techniques should be distributed for existing systems. While the pilot successfully selected an appropriate testing tool, it cannot conclusively state how other types of tools would have performed. Ultimately, the findings reinforce that test automation is not a one-size-fits-all process.

The pilot established a foundation for test automation, but further improvements include minor refactoring, clearer processes, and integrations with CI pipelines. Re-evaluating the distribution of tests may also yield cost and performance benefits. From a research perspective, future case studies should observe the incorporation of test automation over longer periods of time, in different organizational environments, and with more comprehensive metrics.

This study provides practical recommendations for organizations in similar situations. Furthermore, it extends existing research by applying theoretical principles to an existing system, confirming known trade-offs in test levels, comparing current state-of-the-art UI testing tools, and highlighting the importance of context and strategy.

References

[1] G. Myers, T. Badgett, and C. Sandler, *The Art of Software Testing*. Hoboken, NJ, US: John Wiley & Sons, Inc., 3 ed., 2012.

[2] J. Itkonen, M. V. Mäntylä, and C. Lassenius, "How do testers do it? An exploratory study on manual testing practices," in *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement (ESEM 2009)*, pp. 494–497, IEEE, 2009. DOI: 10.1109/ESEM.2009.5314240.

[3] C. Persson and N. Yilmaztürk, "Establishment of Automated Regression Testing at ABB: Industrial Experience Report on 'Avoiding the Pitfalls'," in *Proceedings of the 19th International Conference on Automated Software Engineering (ASE'04)*, pp. 112–121, IEEE, 2004. DOI: 10.1109/ASE.2004.1342729.

[4] K. Wiklund, S. Eldh, D. Sundmark, and K. Lundqvist, "Technical debt in test automation," in *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation (ICST 2012)*, pp. 887–892, IEEE, 2012. DOI: 10.1109/ICST.2012.192.

[5] National Academies of Sciences, Engineering, and Medicine, *Information Technology Innovation: Resurgence, Confluence, and Continuing Impact*. Washington, DC, US: The National Academies Press, 2020. DOI: 10.17226/25961.

[6] A. Mili and F. Tchier, *Software Testing: Concepts and Operations*. Hoboken, NJ, US: John Wiley & Sons, Inc., 2015.

[7] R. Craig and S. Jaskiel, *Systematic Software Testing*. Norwood, MA, US: Artech House, 2002.

[8] "Software and systems engineering — Software testing. Part 1: General concepts," Standard ISO/IEC/IEEE 29119-1:2022, ISO; IEC; IEEE, 2022.

[9] L. Copeland, *A Practitioner's Guide to Software Test Design*. Boston, MA, US: Artech House, 2004.

[10] "Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models," Standard ISO/IEC 25010:2023, ISO/IEC, Geneva, CH, 2023.

[11] B. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ, US: Prentice-Hall, 1981.

[12] M. Cohn, *Succeeding with Agile: Software Development Using Scrum*. Boston, MA, US: Addison-Wesley Professional, 2009.

[13] "Software and systems engineering — Software testing. Part 4: Test techniques," Standard ISO/IEC/IEEE 29119-4:2021, ISO; IEC; IEEE, Geneva, CH, 2021.

[14] Z. Jiang, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, “On the value of static analysis for fault detection in software,” *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, 2006. DOI: 10.1109/TSE.2006.38.

[15] O. Taipale, J. Kasurinen, K. Karhu, and K. Smolander, “Trade-off between automated and manual software testing,” *International Journal of System Assurance Engineering and Management*, vol. 2, no. 2, pp. 114–125, 2011. DOI: 10.1007/s13198-011-0065-6.

[16] R. Haas, D. Elsner, E. Juergens, A. Pretschner, and S. Apel, “How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies,” in *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE ’21)*, (Athens, GR), ACM, 2021. DOI: 10.1145/3468264.3473922.

[17] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, “A Study on the Lifecycle of Flaky Tests,” in *Proceedings of the 42nd International Conference on Software Engineering (ICSE ’20)*, (Seoul, KR), pp. 1471–1482, ACM, 2020. DOI: 10.1145/3377811.3381749.

[18] K. Wiklund, S. Eldh, D. Sundmark, and K. Lundqvist, “Impediments for software test automation: A systematic literature review,” *Software Testing, Verification and Reliability*, vol. 27, no. 8, 2017. DOI: 10.1002/stvr.1639.

[19] 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)*, pp. 272–281, 2013. DOI: 10.1109/WCRE.2013.6671302.

[20] “Software and Systems Engineering —- Software Testing. Part 5: Keyword-Driven Testing,” Standard ISO/IEC/IEEE 29119-5:2024, ISO; IEC; IEEE, Geneva, CH, 2024.

[21] M. Leotta, F. Ricca, A. Marchetto, and D. Olianas, “An empirical study to compare three web test automation approaches: NLP-based, programmable, and capture&replay,” *Journal of Software: Evolution and Process*, vol. 36, no. 5, 2023. DOI: 10.1002/smr.2606.

[22] P. Pasupat, T.-S. Jiang, E. Z. Liu, K. Guu, and P. Liang, “Mapping natural language commands to web elements,” in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 4970–4980, Association for Computational Linguistics, 2018.

[23] J. Kim, Y. K. Malaiya, and I. Ray, “Vulnerability Discovery in Multi-Version Software Systems,” in *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium (HASE’07)*, pp. 141–148, IEEE, 2007. DOI: 10.1109/HASE.2007.55.

[24] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012. DOI: 10.1002/stv.430.

[25] H. Hemmati, Z. Fang, and M. Mäntylä, "Prioritizing Manual Test Cases in Traditional and Rapid Release Environments," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, (Graz, AT), pp. 1–10, IEEE, 2015. DOI: 10.1109/ICST.2015.7102602.

[26] M. Shahin, M. A. Babar, and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017. DOI: 10.1109/ACCESS.2017.2685629.

[27] X. Gan, H. Liang, and C. Brown, "Challenges, Strategies, and Impacts: A Qualitative Study on UI Testing in CI/CD Processes from GitHub Developers' Perspectives," in *2025 IEEE Conference on Software Testing, Verification and Validation, ICST 2025*, pp. 186–197, IEEE, 2025. DOI: 10.1109/ICST62969.2025.10988972.

[28] S. Sun, D. Friberg, and M. Staron, ""Good" and "Bad" Failures in Industrial CI/CD–Balancing Cost and Quality Assurance," in *Software Engineering and Advanced Applications*, vol. 16083 of *Lecture Notes in Computer Science*, (Cham, CH), pp. 75–84, Springer, 2025. DOI: 10.1007/978-3-032-04207-1_6.

[29] Robot Framework Foundation, "Robot Framework User Guide." https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html. [Online] accessed 27-August-2025.