Improving UI test automation performance in Angular application within monorepo with multi-layered testing architecture

Software Engineering
Master’s Degree Program in Information and Communication Technology
Department of Computing, Faculty of Technology
Master of Science in Technology Thesis

Author:
Anh, Vo

Supervisors:
Tuomas Mäkilä (University of Turku)
Sampsa Rauti (University of Turku)

May 2025

Master of Science in Technology Thesis
Department of Computing, Faculty of Technology
University of Turku

Subject: Software Engineering
Program: Master's Degree Program in Information and Communication Technology
Author: Anh Vo
Title: Improving UI test automation performance in Angular application within monorepo with multi-layered testing architecture
Numbers of pages : 61 pages, 3 Appendix pages
Date: May 2025

Test automation has become an essential practice in modern software development, which enhances the efficiency and reliability of the development process. Developers and testers can focus on higher level testing activities such as exploratory testing while test automation can handle the functionality, component, and integration level of the system. Specifically in the context of web applications, user interface testing is critical to ensure that the end-user experience meets expectations. However, it can be time consuming and unreliable without test automation, specifically when tests depend on visual, DOM structure, and the functionality of the user interface. These challenges highlight the need for a robust testing strategy that integrates multiple layers of testing methods.

The first part of the thesis is a literature review of test automation, examining different testing layers, including unit, component, integration, and end-to-end (E2E) testing. The literature review provides a comparative analysis of these testing layers, highlights their strengths, limitations, as well as recommendations for use cases.

The second part of the thesis presents a case study in which the test architecture of an Angular application is redesigned. Currently the application lacks proper testing strategies in which the component tests, integration tests and end-to-end tests are under the same layer. This makes it difficult for the development team to maintain the test suite, slow feedback and long test execution time. The redesigned test architecture will focus on restructuring the testing load across unit, component, integration, and end-to-end testing layers.

The case study seeks to demonstrate measurable improvements in test execution time, reliability, and maintainability. Additionally, it highlights the needs for multi-layered testing approach in test automation and suggests practices for designing a reliable test automation architecture that aligns with web application development within monorepo.

Keywords: test automation, monorepo, multi-layered testing strategy.

Contents

List of Abbreviations

1 Introduction ... 1
1.1 Research background ... 1
1.2 Research objectives ... 2
1.3 The use of AI ... 3
2 Research design ... 4
2.1 Research approach ... 4
2.2 Research methods ... 4
2.3 Data collection ... 5
2.4 Survey ... 6
2.5 Limitations ... 7
3 Theoretical framework ... 8
3.1 Software testing ... 8
3.1.1 Test automation ... 9
3.1.2 Test automation process ... 10
3.1.3 Unit testing ... 11
3.1.4 Component testing ... 13
3.1.5 Integration testing ... 15
3.1.6 End-to-end testing ... 16
3.2 Web development ... 18
3.2.1 Overview of web application architecture design ... 18
3.2.2 Monorepo ... 21
3.2.3 Nx ... 24
3.2.4 Angular as CBS ... 25
4 Literature review ... 27
4.1 Unit testing and Component testing ... 27
4.1.1 Definitions and scopes ... 27
4.1.2 Complexity in test development ... 28
4.1.3 Bug detection ... 29
4.1.4 Execution time ... 29

4.1.5 Conclusion 30

4.2 Integration testing and E2E testing 30

4.2.1 Definitions and scopes 31

4.2.2 Complexity in test development 32

4.2.3 Execution time 33

4.2.4 Conclusion 33

4.3 Multi-layered testing architecture 33

4.3.1 Unit testing and Component testing: the necessary separation 34

4.3.2 E2E testing and Integration testing: the necessary separation 35

4.3.3 The multi-layered testing strategies 35

4.4 Conclusions 38

5 Case study 41

5.1 Background of the case study 41

5.2 Architecture implementation 43

5.2.1 Refinement of the test cases 43

5.2.2 Selection of testing tools, frameworks 45

5.2.3 The implementation of the new testing architecture and test execution pipeline 47

5.2.4 Challenges 49

5.3 Case study results 50

5.3.1 Test execution duration and performance 50

5.3.2 Survey results 51

5.4 Findings and discussions 52

6 Conclusions 54

7 Recommendations for future work 56

References 57

Appendices 61

Appendix 1 Survey questions 61

List of Abbreviations

API	Application programing interface
BDD	Behavior driven development
CBS	Component-based software
CD	Continuous deployment
CI	Continuous integration
DOM	Document object model
E2E	End-to-end
MBT	Model-based testing
MFA	Micro frontend application
MPA	Multi-page application
PWA	Progressive web app
SEO	Search engine optimization
SPA	Single-page application
SSG	Static site generation
SSR	Server-side rendering
UI	User interface

1 Introduction

1.1 Research background

In modern software development, test automation plays an important role in software testing processes and quality assurance. Specifically, in web application development, test automation is crucial for verifying the functionality of the complex User Interface (UI) applications, which has a number of UI elements and functionalities. Test automation reduces manual effort, ensures reliable testing outcomes and faster testing cycles. (Ahmed et al., 2014) For example, unit tests can quickly verify individual functions, while integration tests ensure that different system modules communicate effectively. Test automation also ensures consistency in results with early detection of defects (Wang et al., 2022).

In 2022, studies showed a significant rise in organizations adopting automation tools, with many focusing on improving test automation maturity across various software layers, including unit, component, integration, and end-to-end (E2E) testing (Wang et al., 2022). The maturity of test automation affects the efficiency, reliability, and cost-effectiveness of software testing. A mature test automation process is not only about running automated test scripts, but it also requires a strategic approach that aligns testing efforts with development goals to ensure scalability of the software product. It encompasses the tools, frameworks, methodologies that are used to design, execute, and maintain automated tests. Better test automation maturity results in faster feedback loops, better defect detection, and reduced maintenance costs (Ahmed et al., 2014; Wang et al., 2022).

Multi-layered testing is considered one of the effective test automation strategies. As test maturity evolves, structuring automated tests across different levels from unit to E2E testing is indispensable. Without a well-defined test implementation, in component-based frameworks such as Angular, for instance, modifying a single UI component can cause updating to multiple interdependent tests. To mitigate these challenges, Yadavali (2022) proposes a multi-layered test distribution approach. Each layer serves a specific purpose, from verifying individual components to ensuring seamless interactions between modules as well as verification of the whole system (Yadavali, 2022). Therefore, the multi-layered testing approach is crucial for comprehensive testing coverage and for addressing the challenges posed by UI testing in Angular applications. The integration of a multi-layered test structure

within a mature automation framework supports the software reliability and maintainability of software development.

1.2 Research objectives

The objective of the thesis is to explore and enhance the effectiveness of existing test automation architecture through a comprehensive understanding of multi-layered test automation architecture. The research is structured into a theoretical part of literature review and a practical case study.

The theoretical section, which includes literature review and theoretical framework, studies existing research on test automation practices with the focus on understanding the differences between different testing layers and their application in test architecture. The section studies the key roles, strengths, and limitations of each practice. Furthermore, it investigates the potential benefits and challenges of adopting such an approach in modern software development environments. The theoretical section, then, forms a solid foundation for the practical part of the thesis which is a case study.

The case study applies the insights gained from the theoretical section to redesign the test architecture of an Angular project within monorepo. Specifically, the architecture is restructured to include integration and component testing layers as separate layers. This part of the research aims to evaluate whether a multi-layered approach addresses the inefficiencies observed in the current test setup.

The research is to answer the following questions:

1. What are the differences between layers in testing architecture: component testing and unit testing, integration testing and E2E testing?
2. What are the benefits of a multi-layered testing approach in user interface testing of Angular applications?
3. Does a multi-layered approach effectively improve test automation performance in user interface testing of Angular applications?
4. What challenges are associated with its implementation in user interface testing of Angular applications?

By combining theoretical analysis with practical application, the thesis aims to provide actionable insights into the benefits and limitations of multi-layered testing strategies. The study contributes to the enhancement of test automation practices in software development, specifically testing of Angular application UI.

1.3 The use of AI

Tools: GPT-4o

The AI tool is used to improve the quality of the writing in terms of grammar, sentence structure, and align the writing styles with academic writing standards.

In the literature review sections, the tool is used to refine the written content and identify key findings from the research papers. This process includes querying the AI about specific publications and then reviewing any information it provides that is related to the topic. If AI produces incorrect information, such content is disregarded and excluded from the thesis.

While the tool is effective in retrieving and summarizing information, it is important to manually verify the accuracy of the outputs. This requires skimming and thoroughly reading the source documents before or after receiving AI-generated responses. Locating the keywords or relevant sections and confirming that the information exists within the original material are necessary.

In the case where the AI cannot directly access materials, such as online books with restricted access, the generated content is reviewed to ensure both accuracy and credibility.

I acknowledge full responsibility for the content of this thesis and accept accountability for any academic violations that may arise.

2 Research design

2.1 Research approach

There are different methodologies in research approach. A deductive approach verifies existing theories with the intention of confirming, modifying, or falsifying. In contrast, an inductive approach emphasizes the derivation of generalized theories from specific observations, data collected in real-world contexts. An abductive approach combines the elements of both, aiming to refine or develop theories based on incomplete or limited data. (Saunders et al., 2012)

Qualitative research is usually associated with the inductive approach, which is about collecting and analyzing non-numeric data to gain an in-depth understanding. Qualitative research methods focus on theory generation by studying the subjective experiences and contexts of individuals or groups of individuals. By analyzing data through surveys, observations, case studies, or literature reviews, qualitative research provides a solid understanding of the topic studied. (Greener, 2008)

In contrast, quantitative research is characterized by its reliance on numerical data and statistical techniques. This approach is suitable for studying measurable changes over time, identifying causal relationships, and applying deductive reasoning to test hypotheses. Quantitative methods are particularly valuable for researchers seeking to evaluate patterns and trends through systematic and replicable analysis (Greener, 2008).

The thesis adopts a qualitative approach to analyze data collected from the survey and the result recorded from the test execution performance in the case study. The research applies theoretical insights from the literature review to a case study focused on improving the performance of the test automation of an Angular application.

2.2 Research methods

In order to establish a solid theoretical foundation, the thesis employs literature review as the research method for the theoretical part. The literature review includes gathering and analyzing relevant research to serve as background for the study. This process offers critical evaluation and deeper insights into the topic studied. A comprehensive literature review requires analyzing and synthesizing the findings to create meaningful connections and interpretations, rather than merely summarizing existing literature (Greener, 2008). The

selection and analysis of collected databases critically impact on the quality of the review. In the thesis, the literature review focuses on the topic of test automation in general and specific test automation practices in web application UI.

The other research method that is employed in this thesis is case study. A case study involves examination of a specific subject within its real-world context. This method can include observing one or more cases, collecting and analyzing documents, and utilizing various data collection techniques to obtain qualitative insights into the topic studied. (Greener, 2008) Specifically, in the case study of this thesis, the performance of the test automation in the project is evaluated after the adoption of a multi-layered testing approach.

2.3 Data collection

In the literature review, relevant literature related to test automation practices in web applications for UI is collected. Given the extensive availability of sources on the Internet, it is crucial to evaluate their credibility carefully. Qualified studies, academic journals and research papers with transparent citations are considered for the review. The review collects studies around the topic of different testing practices, the challenges associated with existing UI test automation methods. From which, the review highlights the existing gaps in current test automation practices to lay the foundation for the case study part of the thesis.

For the case study, studies derived from the literature review are applied to refactor the test architecture of the project to address existing inefficiencies and challenges in the current setup. The case study includes designing a multi-layered testing architecture with unit, component, integration, and E2E testing layers to improve test coverage, execution time, and test reliability. The goal is to enhance maintainability and minimize test fragility in an Angular project.

The case study is structured into several phrases. In the initial phase, the current state of the project's test suite is analyzed to identify performance issues as well as test reliability. In the second phase, a revised test strategy is designed and implemented. This includes the creation of component tests, integration tests, and E2E tests refactoring. In the third phrase, execution of the revised test suite in a continuous integration environment are needed to collect key metrics data such as test execution time, and performance. These results are then compared with those from the original test suite to determine the effectiveness of the new architecture.

The case study validates the practical utility of a multi-layered testing approach in Angular applications.

In addition to performance metrics, qualitative feedback is gathered through surveys with the development team. The survey aims to assess the maintainability and reliability of the multi-layered testing architecture, focusing on different aspects such as reduction in test flakiness, and overall satisfaction with the new structure. This contributes to the study on the impact of multi-layered testing strategy. The feedback collected from the survey and performance analysis provides a comprehensive evaluation on the effectiveness of the redesigned test automation strategy.

2.4 Survey

The questions of the survey focus on the surveyed participants' observations regarding the performance of test execution after the adoption of multi-layered testing architecture. The research topic was first discussed with the case company to ensure alignment with the project's objectives and relevance to the ongoing test automation strategy. The participants consist of five team members who are involved in the development of the UI, excluding the author of this thesis. Their insights and feedback provide valuable perspectives on the effectiveness, challenges, and potential improvements for the study of multi-layered testing strategy in Angular application within monorepo.

The search for suitable participants began at the end of 2024, once the research topic had been finalized. Preliminary discussions were conducted to assess the current state of the test automation strategy within the case company and the key challenges it faces. The participants were carefully chosen based on their expertise and hands-on experience with UI web application test automation. To ensure comprehensive analysis, participants were selected from various roles within the team to gain a diverse range of insights.

In order to make sure that participants are well-prepared, the survey is done remotely, and the set of questions were sent via Microsoft Forms. This provides the participants an opportunity to familiarize themselves with the questions, reflect on their experiences, and gather relevant insights. The survey was conducted in English as well as all the answers are recoded for documentation and analysis.

2.5 Limitations

Test automation in web application UI has various testing methodologies, tools, and practices. The thesis specifically focuses on four commonly used testing approaches in the modern web application UI framework, such as unit testing, component testing, integration testing, and E2E testing, which are selected for application in the case study. Given the extensive number of literatures, research papers, books, and industry articles available on the topic of test automation, the literature review in the thesis is based on a carefully curated selection of relevant and accessible sources.

Though, the academic and scholarly resources on the topic of the multi-layered approach in test automation are limited. While several studies address individual testing strategies, there is a gap in comprehensive research that integrates these methodologies into a structured test automation framework. Hence, the thesis considers insights from professional blogs, technical articles, and industry reports, which provide practical, real-world perspectives on test automation implementation. However, it is important to acknowledge that some of these sources may not fully reflect the latest technical advancements, as web development and automation testing have evolved significantly over the past decade. Therefore, the theoretical guidelines presented in the literature review serve as foundational references for the case study.

The case study in this thesis is based on an existing software project currently under development. One of the primary challenges in implementing test automation for this project is the limited availability of tools and frameworks, which restricts the choice of testing solutions. Additionally, due to security policies and compliance requirements within the case company, the project can only utilize approved and validated tools and frameworks. As a result, the case study relies exclusively on Playwright, Cypress, and Jest as the chosen test automation frameworks. The selected frameworks have been evaluated based on their compatibility with the project’s technology stack, and their ability to address different layers of testing.

3 Theoretical framework

3.1 Software testing

Software testing has accounted for a significant portion of development budgets. A study in 2010 estimated worldwide costs of software testing at €79 billion, projected to exceed €100 billion by 2014 (Garousi et al., 2017). According to the 2020 Continuous Testing Report, among 500 large and enterprise-level organizations in North America and Europe, over 55% of them have embedded testing as a continuous process, while the others planned to follow (Capgemini, 2020). These numbers demonstrate considerable growth and the need of software testing in the software industry.

Software testing plays a crucial role in the software development process which ensures the quality of the product meets the defined requirements in terms of functionality, product specifications and user satisfaction. Software testing evaluates different aspects of a software product, such as reliability, usability, and performance. (Sawant et al., 2012) By identifying defects at the early stage of software development process, the cost and time to fix the defects reduces in the later stage of the development process. Moreover, it enhances the overall quality of the software in terms of security, scalability, and usability. (Garousi et al., 2017)

According to Quadri and Farooq (2010), the primary goal of software testing verification and validation is about confirming that the software functions as intended. The testing process should be balanced, addressing user expectations, technical limitations, and project requirements comprehensively. Traceability, which is about the documentation of both successful and failed test cases to avoid redundant efforts and facilitate future improvements, is also an essential metric. The testing process should aim for deterministic outcomes to ensure a structured approach. (Farooq et al., 2010)

In recent years, with the adoption of Agile development methodologies, which require on-demand delivery, frequent releases, and continuous feedback, software testing has a significant role in the quality assurance of the product. Testing in Agile environments is not a one-time activity, but a continuous process integrated into each stage of development, to ensure that the product aligns with user needs and technical requirements. (Al-Saqqa et al., 2020)

In addition, there are different methodologies in software testing such as unit testing, which validates individual components in isolation; and integration testing, which focuses on interactions between different modules of the system. System testing evaluates the entire application against specific requirements, while E2E testing simulates real-world scenarios to ensure that the workflows function as intended. Additionally, performance testing and usability testing are to assess the system's behavior in terms of user satisfaction (Sawant et al., 2012). The integration of different types of testing such as functional, non-functional, regression, and acceptance testing, has become standard to address the different needs of software projects (Jose, 2010).

Therefore, software testing is an indispensable part of the software development process which ensures the reliability and functionality of software applications, especially in the case when the complexity and expectations of modern software continue growing. Software testing helps developers deliver high-quality products while reducing risks and enhancing user satisfaction.

3.1.1 Test automation

In software testing, there are two main types of testing activity: manual testing and automated testing. In manual testing, test cases execution and test result observation is done by human activities, while in test automation, the testing is executed by different tools, scripts, which offer faster execution and scalability. Manual testing is more useful and common in exploratory testing, and usability evaluations where the user's opinion and feedback is beneficial. However, manual testing is time-consuming, prone to human error, and less efficient for regression testing. On the other hand, automated testing is beneficial for regression testing and large-scale projects that require frequent and repetitive validation activities. (Garousi et al., 2017; Jose, 2021)

According to the Continuous Testing Report (2020), test automation has a significant impact on software testing activities. A leading global organization is reported to have achieved up to a 70% reduction in release timelines as automation has enhanced testing efficiency, reduced execution times and accelerated release cycles. Additionally, 45% of 500 surveyed organizations have increased their ratio of automated functional tests, which enables faster execution and broader test coverage. Test automation has become an important part in improving the productivity and reliability of software testing efforts. (Capgemini, 2020)

Test automation offers efficiency, faster execution, and improved accuracy as automated tests reduce human errors and ensure consistent results across regression cycles. Additionally, automation allows scalability, enabling organizations to test complex systems across multiple platforms, environments efficiently. Cost savings are another benefit with the long-term reduction in manual effort as well as investment required for setup and training. (Jose, 2020)

However, test automation still faces challenges. Setting up automation frameworks requires significant initial investment in tools and skilled personnel. The Continuous Testing Report (2020) indicates that managing test environments is one of the greatest challenges to continuous testing and Agile delivery, with 36% of respondents spending more than half their time on building and managing these test environments (Capgemini, 2020). Additionally, maintaining automated tests can be resource-intensive when the test cases are not well-implemented, and the application requires frequent updates to test scripts. Automation is also not suitable for all scenarios, as exploratory and usability testing often relies on human intuition and judgment. Furthermore, smaller organizations face hurdles in adopting automation due to cost constraints and a lack of skilled personnel. (Garousi et al., 2017)

3.1.2 Test automation process

According to Garousi & Elberzhager (2017), in test automation, there are different tools that are used to execute predefined test cases, compare outcomes, and generate reports with minimal human intervention. Typically, the process follows a structured sequence: planning, designing, developing, executing, evaluation and reporting. The testing process is presented below in Figure 1.

Figure 1. Test automation process.

The planning phase defines the objectives, scope, and strategy for automation, such as identifying the functionality, feature or component that would be best, possible for automation, and selecting appropriate tools. (Garousi et al., 2017) For example, the objective is to automate the login functionality in a banking application. The scope includes testing various scenarios, such as valid and invalid login credentials, session timeouts. Jest is chosen

as the automation tool for its ability to handle browser-based interactions, and the timeline is aligned with upcoming release cycles.

The design phase focuses on creating test cases and automation scripts aligned with application requirements. This phrase is about defining inputs, expected outcomes, and test workflows. For instance, to test the login functionality of a banking app, different scenarios are required such as valid, invalid, and edge-case credentials. (Garousi et al., 2017)

In the development phase, these test cases are implemented using selected tools, frameworks, or languages (Garousi et al., 2017). For example, scripts simulate entering usernames and passwords into form fields, clicking the login button, and verifying the success or error messages.

In the execution phase, automated test scripts are executed against specific applications under test environment. Tests are executed across multiple configurations or environments to ensure consistency and compatibility. (Garousi et al., 2017) Taking the login functionality of the banking app as an example, login scripts are run to verify that users can log in seamlessly across all platforms like desktop web application, mobile web application.

Following the execution phrase, the analysis phase reviews the results, comparing them to expected outputs to identify defects or inconsistencies. This phase is critical for refining both the application and the test scripts. (Garousi et al., 2017) For example, a defect is found after the login is sent. Root cause analysis reveals a configuration issue in the backend API. A detailed report is generated, highlighting failed scenarios, root causes, and suggested fixes, which is shared with the development team for resolution.

Finally, the continuous improvement phase is about iterative refinements to the test automation process. Insights from earlier cycles are applied to update test scripts, improve in test coverage, and address newly identified challenges. (Garousi et al., 2017) For instance, based on the analysis phase, improvements are made to the test scripts, such as adding validations for error messages and expanding coverage for edge cases.

3.1.3 Unit testing

According to the ISO/IEC/IEEE (2010), unit test can be determined as:

> “1. testing of individual routines and modules by the developer or an independent tester.

2. a test of individual programs or modules in order to ensure that there are no analysis or programming errors. ISO/IEC 2382-20:1990, Information technology — Vocabulary — Part 20: System development.20.05.05.

3. test of individual hardware or software units or groups of related units”

(ISO/IEC/IEEE, 2010)

According to Dooley (2017), unit testing is primarily a form of white-box testing, which allows developers to test components in isolation. Unit testing is one of the common software testing methods focused on verifying individual units of code, such as functions, methods, to ensure they work as intended. Hence, unit testing is vital in identifying and addressing defects early in the software development lifecycle to improve overall software quality and reliability. (Dooley, 2017)

In today’s software development era, almost every programming language has its own unit testing framework, which offers the use of small, executable unit tests. For example, JUnit for Java, NUnit for C#. Unit testing also becomes a mandatory part of any software development process. (Daka et al., 2014)

The key characteristics of unit testing are defined in Robert Martin's "FIRST" principles. Unit tests should be

- Fast: the test should be simple and small enough for frequent execution to quickly detect issues without hindering development.
- Independent: each test should be operated in isolation, unaffected by dependencies between tests
- Repeatable: the test should provide consistent outcomes regardless of when or where they are executed.
- Self-validating: the test result or test log should provide a clear pass or fail outcome without manual inspection.
- Timely: the test should be written when necessary and can be executed when required.

These characteristics make unit testing reliable and efficient for validating individual components of a system. (Dooley, 2017)

Code coverage is another important metric of unit testing, which evaluates how thoroughly a test suite validates the codebase. Coverage ensures that most of the essential parts of the code are tested, including all executable statements, conditional branches, and logical expressions. There are three key metrics in unit test coverage: statement coverage, branch coverage, and path coverage. Statement coverage ensures that every line of code is executed during testing, branch coverage verifies that all possible conditional branches in the code are tested, and path coverage evaluates whether all possible execution paths have been exercised. While high code coverage provides valuable insights into testing thoroughness, achieving 100% coverage can be impractical due to the complexity of the system. (Dooley, 2017)

One of the primary benefits of unit testing is its ability to detect defects early in the development process, improve code quality, and facilitate refactoring. Daka and Fraser (2014) highlight that developers use unit testing as considerable practice for producing high-quality software. Over 47% of surveyed practitioners emphasize that fixing code to pass unit tests is their primary strategy for handling failing tests (Daka et al., 2014). Golian et al. (2022) mentioned that unit tests are considered as cheapest and less intensive resource compared to integration or end-to-end tests. This makes unit testing a cost-efficient approach for defect identification and prevention early in the development cycle. (Golian et al., 2022) Hence, unit testing plays a crucial part in software quality assurance, defect prevention, cost reduction, and code maintainability in software development.

However, unit testing faces several challenges. Maintenance is a significant concern as when software functionality changes, tests must be updated to reflect changes in functionality. In the report of Daka and Fraser (2015), 25.9% of developers find themselves frequently repairing tests to align with modified code behavior, while 15.6% occasionally delete failing tests due to outdated or irrelevant assertions. Another challenge is dependency management as unit tests must operate in isolation, which often requires the use of stubs or mocks to simulate external systems. However, these techniques can introduce complexities, particularly in large codebases. (Daka et al., 2014)

3.1.4 Component testing

According to the ISO/IEC/IEEE (2010), component testing can be determined as:

> "Testing of individual hardware or software components" (ISO/IEC/IEEE, 2010).

According to Myers et al. (2011), component testing ensures that each unit is robust and aligns to its functional specifications. Component testing validates the functionality, usability, and behavior of individual software components in isolation. These components are defined as the smallest executable units within a system and are tested independently to verify that they meet specified requirements. (Myers et al., 2011)

Component testing offers early detection of defects, which reduces complexity in the debugging process and lowers the overall cost of development. Identifying and resolving issues at the component level minimizes the risks of bugs during system integration, saving significant time and resources (Myers et al., 2011). According to Zheng (2012), well-tested components contribute to the overall system's reliability, as independent validation of components reduces the risk of integration errors (Zheng, 2012).

Jose (2021) highlights the importance of integrating testing into Agile workflows to maintain software quality and adaptability (Jose, 2021). Integrating component testing in the early testing process is a considerable approach. By automating the process of testing components, the CI pipelines provide early feedback to developers, reduces the likelihood of defects into later development stages.

There are various techniques in component testing practice. For example, Zheng (2012) discusses scenario-based testing, which focuses on evaluating component behavior under specific operational conditions. This technique ensures that components perform as expected in practical use cases, highlighting potential integration issues early. Scenario-based testing is particularly effective in validating components designed for complex environments. (Zheng, 2012) In addition, automated testing tools are indispensable in modern component testing. Selecting appropriate tools to enhance testing efficiency and ensure reliable outcomes (Jose, 2021).

Despite its benefits, there are challenges in component testing. Components often rely on external systems or dependencies that must be simulated using techniques like mocking and stubbing. Hence, it is important to design test cases that account for these external interactions to ensure that components perform reliably under diverse conditions (Meyers et al., 2011). Zheng (2012) points out that the absence of a unified approach to component testing often hinders its effectiveness, particularly in complex systems with numerous interdependent modules (Zheng, 2012).

3.1.5 Integration testing

Integration testing is defined by the ISO/IEC/IEEE (2010) as

> "The testing conducted to evaluate the compliance of a system or component with specified functional and non-functional requirements when components are integrated together". (ISO/IEC/IEEE, 2010)

Golian et al. (2022) define integration testing as the process of validating data flow and communication between integrated units, ensuring that they interact correctly and produce the desired outcomes. It serves as a bridge between unit testing and E2E testing, which focuses on isolated components, and system testing, which validates the entire application. (Golian et al., 2022) This layer of testing is especially vital for detecting defects in interfaces, communication protocols, and data exchanges.

With the ability to test the software components as a whole, integration testing helps detect and resolve interface-related issues. Integration testing identifies problems in the interaction between components, such as data transfer errors, or unexpected dependencies between modules (Jose, 2021). By focusing on interactions and workflows, integration testing ensures that combined components behave as expected under different scenarios. Well-implemented integration testing creates reliability in the components' compatibility within complex systems (Zheng, 2012). Additionally, in distributed systems, integration testing verifies communication between microservices, databases, and third-party APIs, which allows testers to evaluate component interactions in isolation (Jose, 2021).

In Agile methodologies, by adopting integration testing as part of the continuous integration pipeline, when new code integrates into the existing system, automated integration tests validate changes incrementally. This iterative validation approach minimizes the risk of introducing regressions as new features are added (Zheng, 2012).

There are different practices in integration testing. According to Sawant et al. (2012), there are two different types of integration testing strategies that are top-down and bottom-up**.** The "top-down" approach tests high-level modules first, progressively integrating lower-level components. Conversely, the "bottom-up" approach validates foundational modules before incorporating higher-level functionalities. (Sawant et al., 2012) Another widely adopted methodologies for integration testing is Behavior-Driven Development (BDD). This approach bridges the gap between technical and non-technical stakeholders by using human friendly language to define test scenarios. Tools like Cucumber facilitate BDD by enabling the

creation of test scripts in plain language, all stakeholders share a clear understanding of system behavior. BDD does not only improve communication but also enhances the quality of test cases, making it a valuable strategy for integration testing. (Trad, 2023)

As integration tests are executed in a specific environment, a realistic test environment that mirrors the production environment is critical. Misconfiguration or variation in the test environment can lead to discrepancies in test results. Conversely, correct environment setup ensures that integration tests validate component interactions under actual operating conditions. This is also considered as a challenge in integration testing. (Jose, 2021) Another challenge in integration testing is dependency management as external dependencies, such as third-party APIs or cloud services, add further complexity that affects the test reliability (Golian et al., 2022). Data management in integration tests is also a challenge, particularly when different scenarios require unique database configurations. Hence, it is important to ensure consistency in data management during testing process. (Trad, 2023)

3.1.6 End-to-end testing

According to ISTBQ (2023), E2E testing is defined as

> "A test type in which business processes are tested from start to finish under production-like circumstances" (ISTQB, 2023).

According to SmartBear (2025), E2E testing plays a crucial role in the software development life cycle that is used to verify the functionality and performance of an application as a whole. E2E testing evaluates the entire product from the perspective of the end user. This involves starting from isolated test criteria to evaluate the entire user journey by simulating the steps a potential customer might take when interacting with the software. (SmartBear, 2025)

In complex software systems that rely on interconnected subsystems such as databases, interfaces, and external applications, a failure in any of these components can compromise the entire product. Hence, E2E testing does not only examine the software's interfaces and behavior but also the performance of these underlying subsystems. This provides comprehensive test coverage across all layers. As E2E testing adopts a user-centric approach, different scenarios and environments are covered in the testing process to simulate real-world scenarios to ensure that user journeys meet expectations. Additionally, by identifying defects that arise from component interactions, E2E testing ensures that the software meets both functional and non-functional requirements. It helps detect defects across subsystems, such as

API miscommunications or data mismatches, which are not captured in unit or integration tests. This comprehensive approach reduces maintenance and additional efforts for defect fixing by catching errors earlier in the development cycle. (SmartBear, 2025)

According to SmartBear (2025), organizations increasingly integrate E2E testing into their development workflows to ensure stability and reliability across all layers of their software (SmartBear, 2025). In Agile environments, E2E test validates the end-to-end functionality of workflows, ensuring that individual components and their integrations meet user requirements (Mollah et al., 2023). This approach aligns with Agile's iterative nature by enabling early defect detection and continuous validation of system-wide behavior. Additionally, by adopting a systematic approach, E2E test cases can be developed as soon as user stories are defined. This process can happen simultaneously with the implementation of user stories within the application, ensuring that customer requirements are continuously verified throughout the development cycle. (Mollah et al., 2023)

Effective E2E testing strategies emphasize prioritizing critical workflows and adopting a top-down approach, which starts with defining the end-user experience and dividing workflows into actionable test cases (SmartBear, 2025). Additionally, according to Björkman (2023), the typical process of the E2E test consists of five distinct phases: test-case design, test scripting, execution, evaluation, and reporting. The first phase is based on user requirements and workflows. This step focuses on identifying critical user actions and interactions to cover essential features of the application as achieving 100% coverage may not be practical or efficient. Once the test cases are defined, they are scripted using an automated testing tool. During the execution phase, the test scripts are run in a controlled test environment that mimics production settings. Björkman (2023) highlights the importance of a reliable test environment where test data can be reset between runs to avoid inconsistencies. After executing the tests, the results are analyzed to determine whether the application meets the expected criteria. Lastly, the final phase is about documenting the testing outcomes, including success rates, failures, and execution times. Comprehensive reporting ensures that the development team can address defects effectively and refine future tests. (Björkman, 2023)

In Agile environments specifically, as user stories serve as the foundation for development and testing activities, Mollah and Bos (2023) emphasize that E2E testing in Agile environments should be user-centric, systematic, and adaptable to iterative development

cycles. This is to ensure that customer requirements are fulfilled without introducing restrictive changes on the Agile process. (Mollah et al., 2023)

As E2E test is about testing the whole workflow that mimics user experience, and test environment plays a crucial role in the test reliability, the complexity of managing test environments that are production-equivalent can be challenging (Björkman, 2023). Incorrect environment configuration can lead to unreliable results, making it difficult to detect defects effectively. Another technical challenge in E2E testing in testing web applications is that web applications use dynamic loading techniques, where DOM elements are loaded asynchronously by the browser. This makes it difficult to determine when elements are ready for interaction. Automated tests face the issue of lacking a mechanism to reliably identify when actions can be performed on elements or when a page is fully loaded. To address this, test case implementations must carefully synchronize the interaction between the web application and the test code. (Björkman, 2023)

3.2 Web development

3.2.1 Overview of web application architecture design

In today's modern web development, there are a variety of web application UI architectures. From traditional static sites and monolithic architectures to more modern solutions, each of which is designed to meet specific needs of the system depending on performance, scalability, user experience, and maintainability. (Shah, 2024)

Static sites are websites generated at build time with pre-rendered HTML. Static site does not require runtime server processing, which is beneficial for search engine optimization. However, there are limitations when using static sites in the case of handling dynamic content. As the volume of content increases, managing static files can become challenging. Hence, this model is ideal for websites that do not require frequent updates, such as blogs or documentation sites. (Shah,2024)

On the other hand, monolithic architecture integrates all components of an application into a single, unified system. There are two approaches in monolith architecture, which are layered and modular approaches. The layered approach organizes the application into layers: the user interface, business logic, and data access layers. The layered approach is considered suitable for smaller projects with limited scope. However, there are challenges in terms of scalability

and adaptability when the application grows, even with minor modifications. Shah (2024) emphasizes that the modular approach can help mitigate these issues by allowing modules to be developed, tested, and deployed independently. Though, according to Su & Li (2024) these modules still operate under the overall limitations of the monolithic system due to issues with interdependence and resource sharing. (Shah, 2024)

While in monolithic architecture, the frontend and backend are integrated into a unified system, headless architecture separates the frontend from the backend. The frontend is built independently and retrieves data through APIs. This decoupling does not only enhance flexibility and scalability but also supports omnichannel content delivery. (Shah, 2024)

Another major web development architecture is multi-page applications (MPAs), where each page is rendered on the server before being delivered to the client. There are two primary approaches in MPAs, which are server-side rendering (SSR) and static site generation (SSG). Server-side rendering method provides complete HTML pages that are rich with metadata, which is considered beneficial for search engine. However, reloading of the entire page for each user interaction causes slow transitions between pages and high server load. Therefore, the SSR approach is suitable for content-heavy sites, such as news websites or large blogs, e-commerce websites were ensuring that every page is fully optimized for SEO outweighs the disadvantage of slower user interactions. Static site generation (SSG), on the other hand, excels at fast loading time as it involves pre-rendering HTML pages during the build process. According to Borstch (2024) SSG is particularly suitable for websites with content that does not change frequently, such as blogs, documentation, and marketing pages. Despite its advantage in search engine optimization, SSG is less effective for dynamic sites, as any updates require a complete site rebuild. (Shah, 2024)

Single-page applications (SPAs) are considered as the solution for the drawbacks of MPAs. In SPAs, the application first loads a single HTML page, then dynamically updates the content in response to user actions. This provides a smoother, app-like experience for. However, in SPAs, the initial loading phase may be slower since the entire application must be downloaded at once, and dynamically generated content can complicate SEO efforts. Therefore, SPAs are especially effective for applications that require high interactivity and real-time updates, for example Gmail. (Shah, 2024)

Being considered as a middle ground between the pure client-side execution of SPAs and the server-driven rendering of MPAs, isomorphic or universal applications offer another balanced

solution by using the same codebase to render content on both the client and server. This approach improves the initial load time by pre-rendering content on the server while still supporting dynamic interactivity on the client, which not only enhances performance, but also improves SEO. Due to the hybrid nature of it, there are challenges in terms of maintainability and debugging. (Shah, 2024)

Progressive web apps (PWAs) integrate features typically associated with native mobile applications. PWAs provide offline functionality, push notifications, and rapid load times, which promotes user engagement and accessibility. Though, the development of PWAs can be complex. (Shah, 2024) Compared to both SPAs and MPAs, PWAs offer a more comprehensive solution with fast interactivity and robust performance for users.

Micro frontend applications (MFEs) introduce a paradigm shift by breaking down the frontend into small, self-contained modules. For example, Spotify has applied MFE architecture. Features, such as search functionality, playlist management, and account settings, are developed as separate micro frontends. MFEs allows specific updates and modifications without affecting the entire application. Each module is developed, tested and deployed independently, which is beneficial for large-scale applications that require frequent updates. Different from monolithic systems, MFEs excel at scalability and agility, which is crucial for fast iteration environments. (Shah, 2024)

In server-driven architecture, the server controls the structure and content of the user interface. Server-driven architecture offers real-time updates by delivering API-driven content based on user actions, which minimizes client-side complexity and supports fast feature deployment. Server-driven architecture is particularly effective in the cases where uniformity across multiple platforms is required. However, the dependence on a network performance remains challenging in server-driven architecture. (Shah, 2024)

According to the study by Shah (2024), the four major UI architectures that are shaping web development are Monolithic Applications, Single Page Applications (SPAs), Progressive Web Apps (PWAs), and Micro Frontend Apps (MFEs). Monolithic architecture is characterized by simplicity and speed but face challenges in dynamic content and scalability. SPAs provide a more interactive interface at the expense of SEO and initial load performance. PWAs and MFEs represent modern, flexible solutions that combine the best features of traditional and dynamic applications. The choice among these paradigms ultimately depends on the project requirements, technical constraints, and strategic goals of the system. (Shah, 2024)

3.2.2 Monorepo

According to the study by Brito et al. (2018), monorepo is defined as single version-controlled repositories that house multiple software projects. The projects are logically independent or related. The projects share a unified configuration for dependency management, testing, as well as application building logic. Despite being under one monorepo, each project is responsible for its own business logic and does not serve a common purpose or business objective. For example, Google's internal repository holds disparate components such as the Google Search backend, internal libraries, and the Angular framework within the same codebase. (Brito et al., 2018)

Monorepo is categorized into monstrous monorepo and project monorepo (Brito et al., 2018). Monstrous monorepo manages an extensive amount of codebases, which contains billions of lines of code and projects developed by tens of thousands of engineers. Monstrous monorepos are applied by large corporations such as Google, Microsoft, and Facebook (Jaspan et al., 2018). Project monorepo, on the other hand, the codebases are maintained in one single repository. For example, open-source projects like Babel or Ember. (Brito et al., 2018)

The study by Brito et al. (2018) defines the five main characteristics of monorepo.

- First, centralization means that the entirety of an organization's codebase is maintained within a single repository (Brito et al., 2018). For example, Google’s codebase has over 2 billion lines of code (Jaspan et al., 2018).

- Secondly, standardization, which promotes a common toolset across all projects in the repository, such as building, testing, and reviewing code are shared and enforced uniformly (Brito et al., 2018). At Google, the unified build system, code browsing interface, and code review tool, creates a cohesive developer experience. Standardization also supports easier onboarding of new developers and consistent enforcement of coding guidelines across teams (Jaspan et al., 2018).

- Thirdly, visibility, which ensures that engineers within the organizations can view and search for all source code (Brito et al., 2018). According to Jaspan et al. (2018), 79% of surveyed engineers rated searchability of the codebase as important or extremely important to their development velocity, while 67% found it vital to code quality (Jaspan et al., 2018).

- Synchronization is described as trunk-based development, where engineers commit to a shared repository (Brito et al., 2018). This creates continuous integration and reduces long-lived divergence between code branches. With the increase of distributed version control systems like Git and Mercurial, branching models became increasingly popular, enabling developers to work independently and later merge their contributions. Although distributed version control systems originally posed scalability challenges for hosting large monolithic repositories, recent advancements have addressed many of these issues. Both centralized and distributed version control systems can technically support monorepo. (Jaspan et al., 2018)
- Lastly, completeness refers to the principle that all dependencies needed to build or test a project must exist within the same repository (Brito et al., 2018).

In the context of web application development, one of the primary advantages of monorepo is code reusability. According to the study by Ramzan (2024), when working under monorepo, development teams can easily share UI components and utility services across applications. This significantly reduces duplicated effort and creates consistency in application UI. (Ramzan, 2024) In addition, as can be indicated from the survey conducted by Ramzan (2024), developers experienced fewer version conflicts and more reliable builds when dependencies were unified under a single configuration system (Ramzan, 2024). The study by Jaspan et al. (2018), also reports that 54 engineers at Google cited centralized dependency management as a major advantage of monorepo (Jaspan et al., 2018). Developers also benefit from standardized tooling and build processes that reduce setup times for new projects. The monorepo reduced onboarding time and made it easier to launch new web applications by reusing common functions and build configurations. With a monorepo, developers can easily see and contribute to code outside their immediate project area, fostering cross-functional collaboration. Shared repositories encouraged inter-team learning and code sharing. The survey in the study by Ramzan (2024) shows that respondents citing increased awareness of project status and dependencies. (Ramzan, 2024) Similarly, according to the case study on Google by Jaspan et al. (2018), engineers often committed code to external projects, reflecting the collaborative nature of monorepos (Jaspan et al., 2018). Furthermore, Continuous Integration and Continuous Deployment (CI/CD) pipelines become more coherent, enabling organization-wide enforcement of testing and deployment policies (Ramzan, 2024).

Despite the considerable benefits, maintaining a monorepo for web application development presents several technical and organizational challenges. For example, dependency version conflict. The finding in Ramzan (2024)'s study mentions that migrating legacy projects into a monorepo revealed inconsistency in library versions, which required substantial refactoring and alignment (Ramzan, 2024). Developers noted that dependency updates could break existing functionality, especially when different projects relied on conflicting versions of the same package (Jaspan et al., 2018).

Additionally, as the monorepo scales, the extensive volume of the monorepo codebase can be overwhelming. The study by Ramzan (2024) reports that developers struggled to navigate the structure and identify ownership boundaries within the unified repository. In the study by Jaspan et al. (2018), 20 participants are reported to have reduced cognitive load as a benefit of the monorepo, while others found the same benefit in multi-repo systems due to smaller, more focused codebases.

Moreover, the effort to standardize tools and workflows can be substantial. CI/CD pipelines must support shared testing environments, parallelized builds, and incremental deployments. (Ramzan, 2024) Particularly, build and test loads can slow down development cycles. The study by Jaspan et al. (2018) mentions that 60% of engineers preferred the monorepo setup but simultaneously noted long build times as a key drawback, suggesting that complexity can undermine productivity if not carefully managed (Jaspan et al., 2018). Hence, to support a monorepo effectively, organizations must invest heavily in scalable version control systems, build tools, and distributed CI/CD pipelines. Google, for instance, uses a custom version control system and tailored build infrastructure to manage its codebase (Jaspan et al., 2018).

In conclusion, monorepo is a significant model in web application development for large-scaled projects where shared knowledge, tooling consistency, and integrated development are required. Specifically, in Angular application where multiple applications depend on common modules, services, or UI components, the monorepo pattern helps simplify the development process of the shared libraries. The monorepo ensures that modifications to shared code are applied across applications in use, reduces duplication as well as the risk of inconsistency. Monorepo helps mitigate problems with dependency version alignment as all projects in the workspace share the same "package.json" and "tsconfig" base configurations, ensuring coherence and reducing the chance of version mismatches. (Angular Adventurer, 2025)

However, the successful implementation of monorepos depends on strategic investment in tooling, architecture of the project.

3.2.3 Nx

Nx is an open-source build system that provides tooling specifically tailored for large-scale applications. Nx framework accelerates computational tasks such as builds and tests, both in local development and CI environments, which contributes to performance improvements in build and testing processes. Nx enables fast project scaffolding through its built-in generators and plugin ecosystem and simplifies the integration and automation of development tools through its extensible plugin system. Nx supports the use of custom code generators and linting rules to promote consistency and maintain code quality. (Nx, 2025)

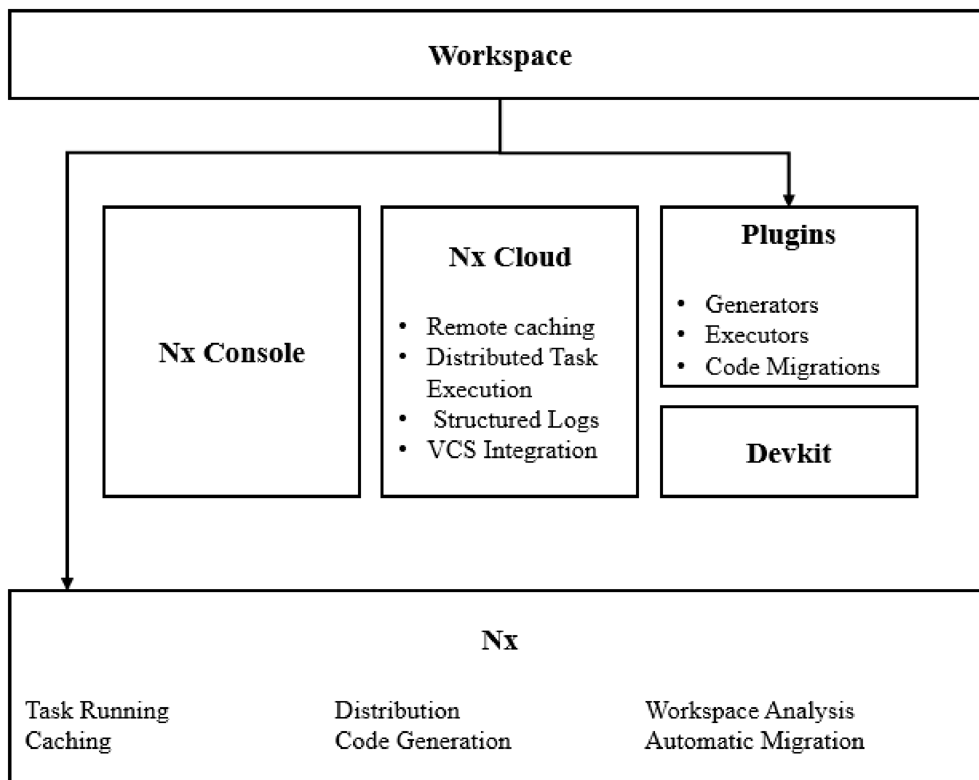

Figure 6. Nx architecture (Nx, 2025).

As it is visualized in Figure 6, architecturally, Nx has modular design, which allows teams to adopt only the components that are relevant to their specific use case. At its core, the Nx package offers workspace analysis, task orchestration, caches, distributed task execution, code generation, and support for automated migrations. For example, the “@nx/cypress” for integrating Cypress-based end-to-end testing. Additionally, Nx enhances enforces

architectural boundaries via tags in “nx.json” to prevent unintended dependencies and ensure clear module responsibilities. Shared libraries allow Angular applications to maintain consistency while minimizing duplication. (Nx, 2025) In the study by Kodali (2024), 81% of developers used shared libraries in micro frontend setups, reporting a 35% decrease in development time and a 92% improvement in UI consistency (Kodali, 2024).

In addition, according to the study by Kodali (2024), the usage of Nx in Angular applications has significantly grown by 150% in just one year. Nx is considered as a foundational tool for implementing micro frontends in Angular applications, which supports the breakdown of monolithic UIs into independent modules that can be developed, tested, and deployed independently. Angular 11 introduced the concept of Module Federation, which allows developers to load independently deployed code into the host application at runtime. The study by Kodali (2024) shows that 73% of organizations using micro frontends in Angular reported a significant reduction in build times and improved code sharing with the support of Module Federation. (Kodali, 2024) Module Federation works alongside Nx to streamline development in monorepo setups. Nx automates this integration by providing CLI commands to generate module-federated apps and manage shared libraries (Nx, 2025).

Compared to npm workspaces, while both Nx monorepo and npm facilitate monorepo setups, they differ in terms of functionality and application scale. Npm workspaces are more suitable for smaller, modular projects, which offer basic capabilities such as shared dependency management and internal package. In contrast, Nx Monorepo is designed for more complex projects. Nx offers a structural foundation for monorepo development as well as delivers the operational efficiency for large applications. (Shah, 2024)

In conclusion, Nx offers a comprehensive, scalable approach to managing modern web applications, particularly within monorepo environments. Nx’s architectural design emphasizes modularity, performance optimization, and seamless integration. In addition to micro frontend, Nx Monorepo is a powerful tool in managing other UI architectures including SPAs, static sites. Nx enables consistent design and promotes code reusability by maintaining a shared component library, facilitating faster development cycles. (Shah, 2024)

3.2.4 Angular as CBS

According to the Angular official documentation, Angular is a popular web application framework developed by Google that is used for front-end web development. Angular has

component-based architecture, which means components serve as the fundamental building blocks of Angular applications. (Angular, 2025) Structuring an application into components helps divide the application codebase into manageable and distinct sections, which promotes maintainability and reusability. In addition, Angular applications are structured as hierarchical component trees, where parent-child relationships determine data flow and rendering, which enhances clarity and modularity (Angular, 2025).

There are different definitions of "component" in the software field. According to ISO/IEC/IEEE (2010), component is defined as

> "1. an entity with discrete structure, such as an assembly or software module, within a system considered at a particular level of analysis. ISO/IEC 15026:1998, Information technology — System and software integrity levels.3.1.
>
> 2. one of the parts that make up a system. IEEE Std 829-2008 IEEE Standard for Software and System Test Documentation.3.1.6.
>
> 3. set of functional services in the software, which, when implemented, represents a well-defined set of functions and is distinguishable by a unique name. ISO/IEC 29881:2008, Information technology — Software and systems engineering — FiSMA 1.1 functional size measurement method.A.4"

(ISO/IEC/IEEE, 2010)

In the case of Angular, each component represents a distinct section of a web page. Angular components are self-contained units that encapsulate HTML templates, styles, and TypeScript logic (Angular, 2025). Therefore, component testing in an Angular application, particularly in the project of the thesis' case study, involves verifying the functionality and behavior of individual components within the application.

4 Literature review

In this section, a literature review is provided based on the understanding of the theoretical framework section and previous research works in the field of test automation. As the case study of the thesis is about Angular web application, the literature review and the comparative analysis provided primarily focuses on the test automation of Angular web application. The first part (section 4.1) of the literature review gives comparative analysis of the unit level of test automation, specifically unit test and component test and how they complement each other. The second part (section 4.2) focuses on the integration level of the test automation, specifically, a comparative analysis of E2E test and integration test is provided as well as their complementary for one another. The third part reviews the concept of multi-layered testing architecture.

4.1 Unit testing and Component testing

Unit testing and component testing play significant roles in validating the behavior, functionality of software systems on the unit levels, especially in modern Angular web applications. They help identify defects early in the development cycle and reduce the cost of bug fixes. This review studies the differences between unit test and component test based on its definitions, and coverage fields, execution time as well as how they can complement each other. Additionally, this section aims to answer the component testing, unit testing part of the first research question:

> **RQ1:** What are the differences between layers in testing architecture: component testing, unit testing?

4.1.1 Definitions and scopes

Unit test is defined as a form of white box testing which verifies a single software unit at the function or method level, to ensure they behave as expected in isolation. Unit test validates individual classes or methods without testing their interaction with other system components. (Dooley, 2017; ISO/IEC/IEEE, 2010)

Component testing also focuses on validating software components in isolation (ISO/IEC/IEEE, 2010, Myers et al., 2011). However, component testing involves testing a collection of related modules such as functionality, usability within a software component and behavior of individual software components (Myers et al., 2011). This means, component

testing extends unit testing in terms of coverage as component testing verifies the interactions and integration of software units. Unlike unit testing, in addition to testing a single software unit, component testing assesses the interactions among multiple units under a controlled test environment.

Unit testing focuses on specific code segments and aims to verify that individual functions return expected outputs, assess code coverage, detect and isolate defects at an early stage (Dooley, 2017). Unit testing helps achieve white-box testing objectives by executing all possible code paths, including conditional statements and error-handling mechanisms (Dooley, 2017). Conversely, according to Zheng (2012) component testing verifies the overall behavior of a module by assessing internal interactions, validating compliance with functional requirements, and testing how a component handles various inputs and error scenarios (Zheng, 2012).

Specifically, in testing Angular applications, each unit test focuses on a single responsibility which verifies the functionality of units of code. The unit tests are used to evaluate functions, methods, objects, types, and values. On the other hand, component testing focuses on evaluating methods, and event bindings while ensuring isolation from external dependencies. For example, verifying method outputs, as well as simulating user interactions, like button clicks, to confirm changes in the component state. (Palmer et al., 2018)

4.1.2 Complexity in test development

According to the survey on unit testing practices by Daka and Fraser (2014), developers face challenges maintaining unit tests when the application grows in complexity. As the test should stay in isolation and the coverage metrics should be retained at a certain level, there is extensive need for mocking and stubbing to isolate units. (Daka et al., 2014) Component testing in Angular applications faces the same challenge in terms of dependencies. Palmer et al. (2018) highlights that in order to make the component test module work, different dependencies are required as testing Angular components is about ensuring a fully functioning component (Palmer et al., 2018). Unlike unit tests, Gao (2000) highlights that component testing often requires more test stubs and mock objects to simulate interactions with other modules and to simulate different conditions, which adds an additional layer of complexity to the test setup (Gao, 2000). According to the study by Chasidim et al. (2018), such additional setup increases the maintenance burden, as changes to one part of a component often require updates to multiple test cases (Chasidim, 2018).

Hence, unit testing is less complex than component testing due to its focus on isolated functions and lower maintenance requirements. Component testing, by contrast, introduces greater challenges due to dependency management, as well as the difficulty of identifying the root causes of failures.

4.1.3 Bug detection

According to Dooley (2017), unit tests serve as a foundation for software reliability, which ensures that individual methods return expected outputs (Dooley, 2017). Unit tests are effective in detecting syntax errors, logic faults, and small-scale functional discrepancies within isolated units of code. However, unit testing can catch a substantial portion of low-level defects, it is not suitable for identifying broader system-wide failures (Daka et al., 2014). Additionally, research by Daka and Fraser (2014) indicates that software failures frequently arise not from isolated unit malfunctions but from incorrect interactions between units that are not covered by unit tests. In other words, unit tests fail to capture integration and interaction defects. Specifically in the case of testing Angular components which includes both code logic and UI behavior, unit testing does not guarantee full code coverage despite high statement and branch coverage as unit tests isolate functions. Golian et al. (2022) emphasize that comprehensive software quality assurance must involve a multi-layered testing approach that extends beyond unit testing (Golian et al., 2022).

In conclusion, in terms of bug detection, unit testing provides a strong foundation for detecting low-level bugs, it does not address integration issues. In order to ensure high quality in testing Angular components, Palmer et al. (2018) emphasizes the importance of verifying both logic and UI rendering (Palmer et al., 2018). Hence, complementary testing methodology, specifically component testing, are considerable to mitigate risks and enhance overall software quality.

4.1.4 Execution time

The execution time of software testing plays a crucial role in determining the efficiency of the software development lifecycle, particularly in Agile.

Unit tests are supposed to be small, simple, execute fast and provide quick feedback as unit tests do not include any interactions with external libraries or interfaces (Dooley, 2017). Hence, unit tests run in isolation and are inherently faster. Unit tests are highly effective for

catching syntax errors, logic flaws, and small-scale functional bugs, allowing development teams to detect and resolve regression instantly (Dooley, 2017).

Component testing, on the other hand, validates the behavior of a module consisting of multiple interdependent units, which makes it slower than unit testing. Gao (2000) explains that component tests often require the initialization of multiple objects, interaction with external services, and the simulation of different scenarios, which increases execution time (Gao, 2000).

In conclusion, as component tests take longer execution time, while unit tests are run continuously during development, component tests are executed less frequently. Unit tests are suitable for early defect detection of individual units and component testing for integration validation of multiple units.

4.1.5 Conclusion

In conclusion, based on the studies and reviews from the literature in this section, the answer to the first research question on the differences between unit testing and component testing is that unit testing and component testing serve different purposes. But they have complementary roles in testing Angular applications. Unit testing focuses on verifying individual units in isolation, specifically functions, method of the component logic. Component testing extends unit testing by validating the integration and interaction of multiple related units within a component. Unit testing is effective in identifying low-level defects but fails to detect integration and interaction issues. Component testing mitigates this gap by validating both code logic and UI behaviour, ensuring proper communication between software components. This is particularly relevant in Angular applications, where UI interactions play a critical role in functionality. Lastly, unit tests are executed quickly and frequently since they focus on small, isolated code units. Component tests, on the other hand, are slower and executed less frequently due to dependency handling, and complex interactions.

4.2 Integration testing and E2E testing

Unit testing and component testing focus on testing the application functionality, behavior in isolation, which do not verify whether all different parts of the application work together as expected and meet user expectation. The Software testing pyramid in Figure 2 shows that

integration testing bridges the gap between E2E test and unit test. Integration test and E2E test also focus more on testing the interactions between different components, services, and functionality of the system. (Golian et al., 2022)

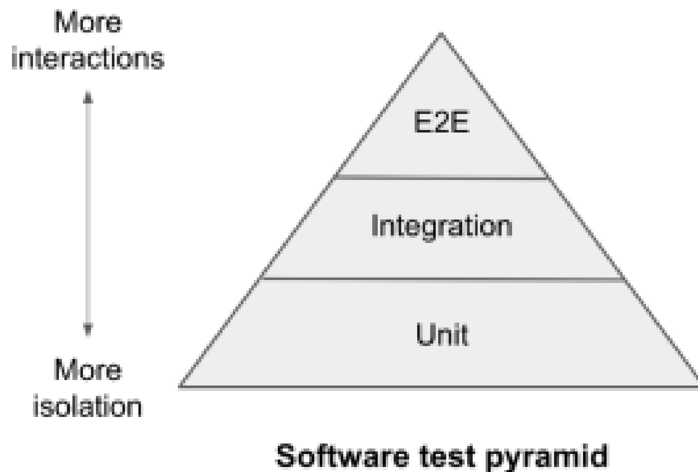

Figure 2. Testing pyramid (Golian et al., 2022).

Hence, integration testing and E2E testing are essential in web application development, specifically Angular web applications. This section provides a literature review on the key differences between integration testing and E2E testing in terms of definition, scope, complexity in test development as well as execution time of these testing methodologies. Additionally, this section aims to answer the second part of the first research question which is about the differences between integration testing and E2E testing:

> **RQ1:** What are the differences between layers in testing architecture: integration testing and E2E testing?

4.2.1 Definitions and scopes

Integration testing focuses on verifying how different software components function and interact with one another when integrated (ISO/IEC/IEEE, 2010). According to Petlovyi (2024), integration testing ensures that individual components, services, and modules can effectively and correctly communicate with each other. Integration testing does not necessarily involve user interface interaction as it validates only specific interactions between services, components, and dependencies (Petlovyi, 2024). In Angular applications, integration tests use mock service or mock versions of external dependencies such as API, database (Palmer et al., 2018).

End-to-End testing refers to a software testing methodology that validates an application's workflow from start to finish in a production equivalent environment (ISTQB, 2023). Different from integration tests, according to Mollah and Bos (2023), E2E tests in web applications are executed by automated test scripts which simulate user interactions across different system layers (Mollah et al., 2023). Specifically in Angular applications, testing frameworks like Cypress, Playwright, and formerly Protractor allow developers to write automated test scripts that simulate user actions as well as different scenarios [28]. As a result, E2E testing helps identify UI inconsistencies or issues in user's workflows. According to Petlovyi (2024), E2E testing environments include headless browsers, virtual machines, and cross-browser testing frameworks that allow automated test execution across different conditions (Petlovyi, 2024). In addition, E2E tests are executed in the environments that are deployed in a real or simulated user's environment (Björkman, 2023). Hence, Palmer et al. (2018) highlights that E2E tests are considered more expensive than integration tests.

Integration testing has narrower scope compared to E2E testing as the purpose of integration testing is to verify different components, services and data flows correctly behave as expected when integrated with others. The scope of E2E testing is significantly broader than integration testing as it aims to validate the entire application workflow rather than specific module interactions. While integration testing is particularly useful for detecting early-stage defects in Angular applications before full application deployment, E2E testing provides comprehensive validation of how the application behaves in a production equivalent environment. (Mollah et al., 2023; Petlovyi, 2024)

4.2.2 Complexity in test development

According to research, integration testing ensures that all or partially of the services and components behave as expected before full-scale integration (Trad, 2023). Hence, integration testing is relatively simpler compared to E2E testing because it focuses on verifying isolated interactions between modules rather than the entire application. According to Palmer et al. (2018) in Angular integration testing using mock versions of services, databases, there is limited need for a fully functional UI during testing. This creates a controlled environment which reduces flakiness in tests and ensures reliable results.

E2E tests validate an application from the user's perspective, it is required to interact with the UI, network operation, and verify server responses (Liu et al., 2024). This introduces challenges such as timing issues, inconsistent network conditions, and UI element changes

(Ricca et al., 2020). Research highlights that flaky tests that pass or fail unpredictable are a common problem in E2E testing, often caused by dynamic UI elements and asynchronous API responses (Ricca et al., 2019). Additionally, E2E tests are executed in a live or simulated browser environment, which increases the complexity in test development. However, research suggests that E2E testing reduces the risk of post-deployment failures, as it replicates actual user behavior, which is crucial in complex workflows (Petlovyi, 2024).

4.2.3 Execution time

Integration tests run within a controlled development environment using mocked services, mocked backend responses, which execute faster (Trad, 2023). Conversely, E2E tests require UI interactions that simulate user workflow, in which the loading, navigation and network responses in every test result in significant waits (Liu et al., 2024). Additionally, UI tests are highly sensitive to frontend changes, leading to frequent test failures that may not indicate actual defects (Ricca et al., 2019). Running a full suite of E2E tests can take several minutes to hours depending on the complexity of the application and the number of test cases (Petlovyi, 2024). Therefore, the execution time of integration tests is significantly faster than E2E tests. Due to this high cost, E2E tests are selectively executed, focusing only on critical workflows rather than running a full test suite frequently (SmartBear, 2025).

4.2.4 Conclusion

In conclusion, the answer to the first research question on the differences between integration testing and E2E testing is that integration testing provides fast validation of UI component interactions without relying on the responses from backend APIs and services, which mitigates the challenges that E2E testing faces. Hence, integration testing is effective for early-stage validation before the full system integration. This allows E2E tests to focus on critical user interactions, offer comprehensive validation of the full application workflow. By combining both testing strategies, a fully functional system is verified.

4.3 Multi-layered testing architecture

In order to ensure the functionality and reliability of the software product, it is important to have rigorous testing strategies in web application development. A multilayer testing approach, which includes different testing methods, provides a structured framework for

validating the web application UI at different levels of granularity. This literature review aims to study the benefits of a multi-layered approach as well as answer the third research question:

RQ2: What are the benefits of a multi-layered testing approach in user interface testing of Angular applications?

4.3.1 Unit testing and Component testing: the necessary separation

The most common practices in unit testing of Angular applications that are introduced in the guidebooks such as Palmer et al. (2018), Schäfer (2021), Angular (2025) is to have component testing and unit testing at the same layer (Angular, 2025; Schäfer, 2021; Palmer et al., 2018). However, as discussed in section 4.1, unit testing is primarily designed to validate the smallest testable parts of an application like individual functions and classes. In testing of a more complex component which contains different services, dependencies, and UI interaction, more mocking and stubs are required. In order to isolate the functionality being tested, there are test doubles created, such as spies and stubs (Imtiaz et al., 2019). According to Daka and Fraser (2014), extensive dependency mocking and stubs introduce complexities in unit testing as well as increases the difficulty of achieving realistic test coverage (Daka et al., 2014). As a result, according to Jose (2021), these additional layers of test preparation increase the complexity of writing and maintaining tests (Jose, 2021).

Furthermore, when testing Angular components that require validation of UI behavior, the mocking of the UI dependencies would make the test become fragile. This practice is considered as poorly designed unit tests that focus on implementation details, according to Jose (2021) (Jose, 2021). As the tests validate an artificial environment rather than real-world behavior, which is prone to failures when implementations change, even if the component's behavior remains (Leotta et al., 2023). Hence, component testing should focus exclusively on the UI interactions of a component, ensuring that it behaves correctly in response to user actions.

As the application codebase extends, or components, services code logic are refactored that causes broken test cases, the implemented tests need to be updated accordingly to reflect changes (Balsam et al., 2024; Imtiaz et al., 2019). Manual interventions also increase the cost of test suite maintenance (Balsam et al., 2024). Hence, in complex components, combining unit tests and component tests increase the total number of unit tests which lead to increased execution time as well as the cost in test maintainability.

4.3.2 E2E testing and Integration testing: the necessary separation

E2E testing requires the full setup of a testing environment, databases, authentication mechanism as well as UI rendering, which is time-consuming (Ricca et al., 2020). In the study by Liu et al. (2024), flakiness in E2E testing is the most persistent issue. Tests frequently fail due to non-deterministic behavior by asynchronous operations, dynamic UI rendering, and network variability. The reliance on external dependencies, such as backend services, third-party APIs, inconsistencies in responses can result in unpredictable test outcomes. (Liu et al., 2024) Specifically in testing of the UI, factors such as browser compatibility issues, inconsistent page load times, and unreliable network conditions also contribute to the instability of E2E test results (Ricca et al., 2020). These issues with instability and flakiness in E2E tests prolong the execution time in test run (Balsam et al., 2024). This makes it challenging to have frequent E2E in CI/CD pipelines that require a quick feedback loop.

The fragility of E2E tests also impacts maintainability. Specifically in testing of the UI, minor UI modifications in element attributes or styles, can cause test failures even if the core functionality remains unchanged. Maintaining a stable E2E test suite demands constant updates and debugging efforts. (Imtiaz et al., 2019) Given the case when integration testing and component testing are implemented within the same layer with E2E testing, any modification in the UI can cause test failures, leading to unnecessary maintenance efforts (Yadavali, 2022). Hence, while integration testing and component testing does not require the whole environment setup, having them in the E2E testing causes redundant test execution and increased execution time.

4.3.3 The multi-layered testing strategies

Component-based frameworks like Angular introduce dependencies between tests as refactoring a UI component requires modifications to multiple dependent tests., making maintenance more costly and time-consuming (Yadavali, 2022). Yadavali (2022) proposed techniques like resilient selectors and test abstraction layers to reduce test fragility in testing web applications. Strategically distributing the tests accordingly prevents frequent test breakages due to UI changes and improves the testing scalability and maintainability.

(Yadavali, 2022) The distribution is suggested as in Figure 3.

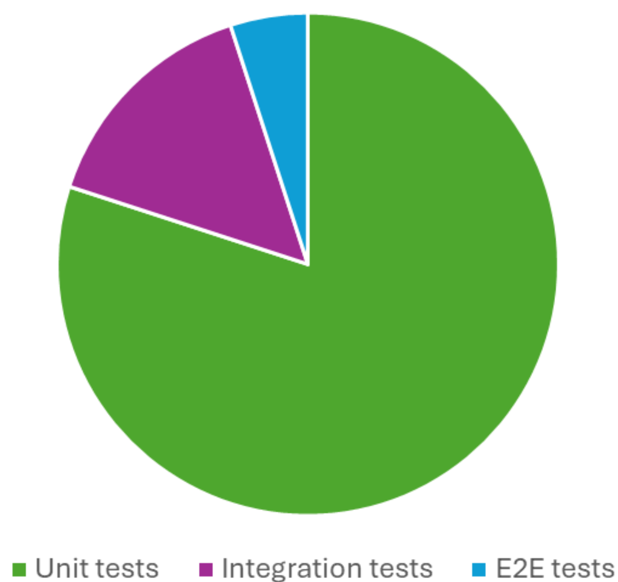

Figure 3. Test methods distribution.

- Unit tests account for 70-80% of the test suit for quick feedback.
- Integration tests account for 10-15% of the test suit focusing on component interactions.
- E2E tests focus on critical workflow with 5-10%.

(Yadavali, 2022)

It is important to have the right balance between different testing layers. A testing strategy that focuses solely on low-level methods may fail to detect defects during integration. Conversely, excessive reliance on high-level methods, such as integration and E2E testing, increases maintenance costs and introduces challenges related to test execution time and feedback loops. Several articles on the multi-layered approach in testing Angular applications suggest the testing strategies with the application of the testing pyramid. For example, the recent articles by William (2023) and Testim (2023). Unit tests are at the foundation of the testing pyramid focusing on small, self-contained units of logic and utilizing mocking techniques to decouple dependencies (Bastidas, 2023). Component testing in Angular extends unit testing by validating the rendering, lifecycle hooks, and event handling of individual components. Unlike unit tests, which focus purely on logic, component tests assess the behavior of UI elements and component interactions within an isolated environment. (Bastidas, 2023; Testim, 2021) Integration testing bridges the gap between component tests

and full E2E workflows. Integration tests validate the interactions between multiple components, services to ensure a functional integration of components (Testim, 2021). In Angular applications, integration testing is particularly necessary for checking routing mechanisms, HTTP client interactions, and application state management (Bastidas, 2023). At the top of the testing pyramid, E2E tests validate entire user workflows by simulating real-world interactions in a browser environment (Bastidas, 2023; Fowler, 2019). Bastidas (2023) also highlights poor separation between integration and E2E tests lead to redundant verifications, slowing down test execution and increasing maintenance costs (Bastidas, 2023). Therefore, best practices recommend minimizing E2E test coverage to only critical user journeys while relying on lower-level tests for most verifications (Testim, 2021).

Fowler (2019) emphasizes testing is a crucial part of code refactoring as well as the role of automated testing in Agile software development. Different testing layers enhance software quality while minimizing debugging efforts. Unit testing is considered as the foundation of automated testing, focusing on small, isolated units of code. It is the backbone that is designed to operate on specific areas of a system and run quickly, providing immediate feedback during development. However, they are insufficient for capturing integration failures, which means integration testing is indispensable. Integration testing validates the interactions between different software components to ensure that the components communicate as expected. (Fowler, 2019) This underscores the importance of a multi-layered approach in test automation of web application, where unit and integration tests ensure software stability before full-system validation by E2E testing.

Spillner et al. (2014) introduces the V-Model (Figure 4) that is a widely used software development and testing methodology that integrates systematic verification and validation processes at each stage of development. The model emphasizes the interdependence of development and testing activities in which each phase of development has a corresponding testing level (Spillner et al., 2014). This structure enhances defect detection, ensuring that issues are identified and resolved in the early stage of software deployment. Spillner et al. (2014) discusses that defects that are found in the later stage of the development cycle, particularly during system testing or deployment, are significantly more expensive to fix, compared to those identified in earlier stages. Software verification should be an incremental process where testing is conducted at various levels to detect errors and prevent defect propagation. Which means unit testing and component testing, which focus on testing individual components in isolation, enables defect detection and resolves them before they

impact other modules. Integration testing then verifies the correctness of module interactions, preventing interface-related defects. (Spillner et al., 2014) Hence, the systematic execution of these tests enables early fault detection, reduces the cost and complexity of defect resolution.

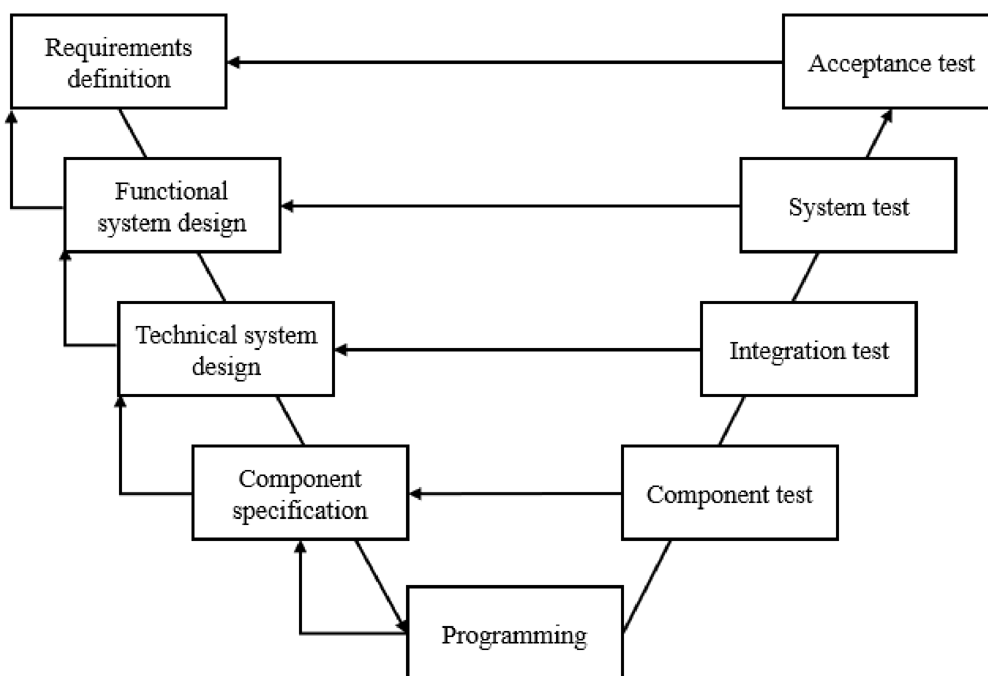

Figure 4. The general V-model (Spillner et al., 2014).

In conclusion, with the adoption of a multi-layered testing strategy in Angular applications, organizations can significantly reduce test fragility, streamline maintenance, and improve feedback cycles (Yadavali, 2022). This layered structure ensures that each level of the application is tested appropriately as well as prevents over-reliance on flaky and slow E2E. The structure does not only facilitate early fault detection but also minimizes the costs and risks associated with late-stage defect resolution (Spillner et al., 2014). Thus, the multi-layered testing approach promotes incremental verification, enabling teams to localize issues early through unit and component testing, and to validate system-wide behavior through controlled integration and E2E tests.

4.4 Conclusions

Unit testing, component testing, integration testing, and E2E testing serve both distinct and complementary roles in ensuring the reliability of Angular applications. While unit testing and component testing focus on isolated scenarios and fail to validate broader system interactions, integration testing and E2E testing are responsible for validating the system at integration level. Specifically, unit testing focuses on the small, isolated code units, which

makes it the fastest testing method for early-stage defect detection. Component testing verifies the interactions of UI elements, lifecycle hooks to ensure the component's functionality in isolation. Integration testing assesses the interaction between different modules, services, and UI components, which ensures that connected components work together as expected before full-system integration. E2E testing provides comprehensive validation of the application's workflow from a user's perspective within different simulated scenarios.

The table below provides a comparative analysis summary of unit testing, component testing, integration testing and E2E testing in terms of scope, bug detection, execution and maintainability:

Criteria	Unit testing	Component testing	Integration testing	E2E testing
Scope	Testing smallest testable parts of an application (functions, classes).	Validating a collection of related modules within a component.	Validating different services, UI components of the application.	Validating the full user journey across UI, backend, and external systems.
Bug detection	Detects syntax errors, logic bugs, and small functional issues.	Detects component-specific logic and UI behaviour defects.	Detects integration issues of multiple components, services, and incorrect data flow.	Detects full workflow failures, user experience issues, and external system failures.
Execution time	Fast	Moderate	Slower than unit testing and component testing	Slowest
Maintainability	Low. Easier to maintain due to isolated nature.	Moderate. Requiring handling UI changes and dependencies.	High. Requiring updates when component logic changes.	Very high. Frequent updates needed due to UI and system changes.

Table 1. A comparative analysis summary of unit testing, component testing, integration testing and E2E testing.

Multi-layered testing strategy provides a structured, scalable testing approach that balances the trade-offs between test reliability, execution time, and maintenance effort. A multi-layered test automation strategy is essential for modern Angular applications to mitigate the limitations of individual testing methods. While unit testing provides quick feedback and robust low-level validation, component testing improves test coverage by validating UI behavior. Integration testing ensures communication between components and services, reduces potential failures before application deployment. Finally, E2E testing serves as the

last validation step, which verifies complete user workflows in a production-equivalent environment. To optimize test efficiency and maintainability, best practices recommend prioritizing unit, component and integration tests, minimizing redundant E2E test cases. Strategic test distribution, as suggested by Yadavali (2022), reduces flakiness in UI testing and ensures a scalable, maintainable test suite. Multi-layered testing approach results in high software quality and long-term test maintainability, making it a fundamental practice in modern web application development.

Below is the suggested testing pyramid that is applicable for testing Angular applications with complex component logic and UI functionality:

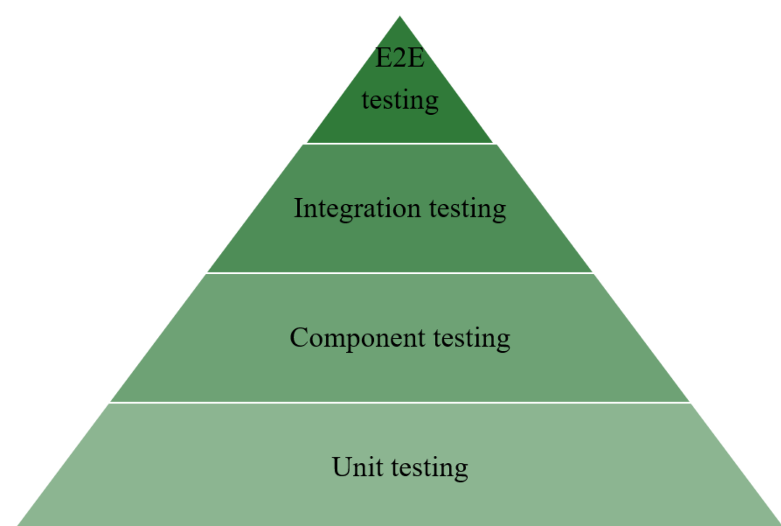

Figure 5. Suggested testing pyramid based on the literature review.

5 Case study

5.1 Background of the case study

The case study is about a project that is developed under a monorepo-based Angular architecture. The repository contains more than five distinct applications, each consisting of two interconnected modules: the main application and a corresponding E2E testing application. Figure 7 visualizes the architecture of the project of the case study.

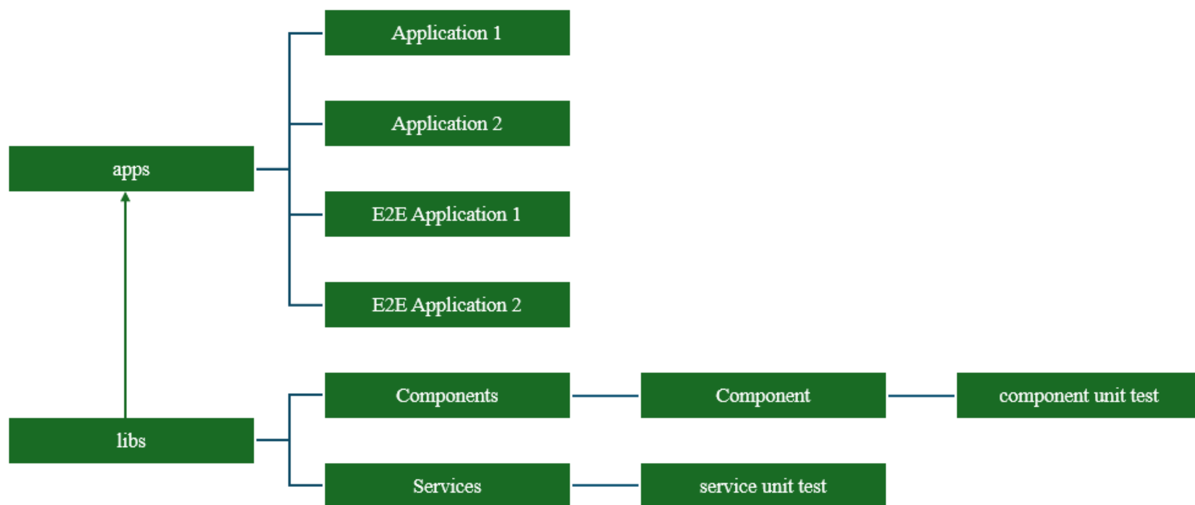

Figure 7. The architecture of the project of the case study.

The project for the case study itself has two distinct applications, hence there are four applications in total to be maintained and developed by the team. In addition, a suite of shared libraries contains reusable components and services. While this setup promotes modularity and code reuse, it introduces significant complexity, particularly in managing testing and ensuring maintainability across all modules.

The key issue in the current test automation strategy of the project for this case study is the lack of a layered test architecture. Component tests and integration tests are executed as E2E tests using Cypress. While Cypress is a powerful tool for simulating full user interactions in the browser and verifying end-to-end workflows, its overuse for lower-level testing tasks has led to inefficiency, flakiness of test suites. In addition to the unstable testing environment, flaky test implementation introduces significant challenges for the development team. This results in prolonged CI/CD pipeline execution and prolonged feedback loops. In addition, the tests intermittently fail or pass without consistent reasons, and no logs were found due to test execution timeout. The lack of detailed logs and diagnostic tools make it difficult for

developers to investigate the root causes of failures. As a result, the testing process becomes a bottleneck in the overall development lifecycle.

The recordings of five E2E test executions are shown in Figure 8. The data is collected from the artifacts of the pipeline that is used for quality checks of each pull request before merging to the main codebase.

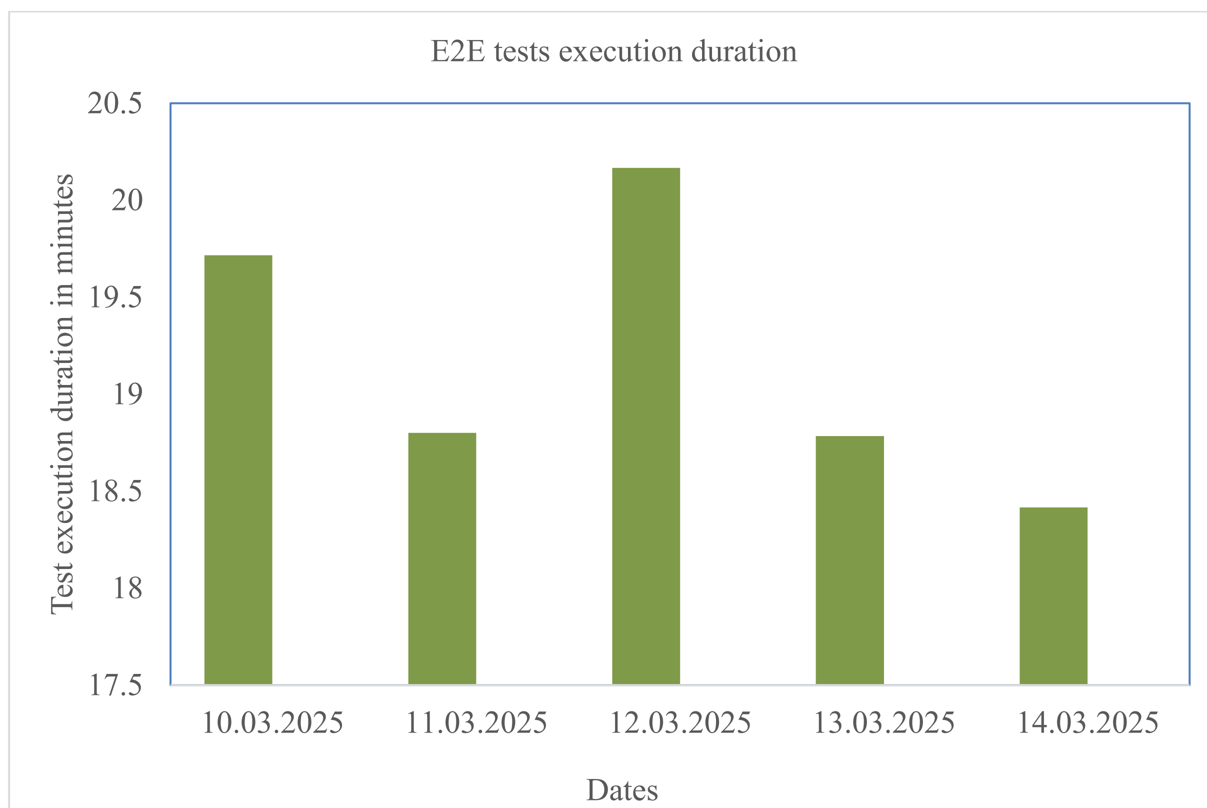

Figure 8. E2E test execution time.

The total time of the E2E tests around 19 minutes on average. This calculation is attributed to the fact that integration tests, E2E tests, and component tests are all implemented as E2E tests, rather than being executed as distinct test layers. The data reflects the cumulative execution time of all test types conducted through the E2E testing mechanism.

In terms of test automation frameworks, the project uses Cypress for E2E testing and Jest for unit testing. Cypress is well-suited for validating end-user scenarios and provides a robust API for interacting with web applications. However, it is not optimized for testing isolated components or services. Jest, on the other hand, offers fast execution, snapshot testing, and mocking capabilities, making it ideal for unit and component tests.

Based on the studies from the literature review, specifically, the suggestions from the studies by Yadavali (2022), Fowler (2019), Spillner et al. (2014) and the discussions within the development team, a multi-layered test automation strategy is taken into consideration to improve the test architecture of the project. This approach divides testing responsibilities into four distinct layers: unit, component, integration, and end-to-end. Each layer is designed to handle a specific type of verification, using the appropriate tools and practices.

- Unit testing focuses on validating individual services and utility functions in isolation. These tests are supposed to be fast and deterministic, providing immediate feedback during development.
- Component testing ensures the UI behaviour of one or two components. Mock services and dependency injection will simulate environment conditions without the performance overhead of full application rendering.
- Integration testing assesses how different modules, services, and components work together. By mocking only external systems, these tests will provide realistic yet efficient validation of internal application interactions.
- E2E testing focuses on validating essential user flows, critical paths so that test execution will be faster and less prone to instability.

The testing architecture was developed in March 2025. Refactoring the existing test suites involves decomposing monolithic E2E tests and redistributing their logic into the appropriate layers. Redundant or overlapping tests will be removed, and new tests will be added to ensure complete coverage. The CI/CD pipeline will be updated to run these layers in isolation and in parallel where possible, reducing feedback time and improving reliability.

5.2 Architecture implementation

5.2.1 Refinement of the test cases

One of the first steps for the implementation of the new test automation strategy is the test cases refinement plan. The plan focuses on refining the existing E2E and integration test cases according to their purpose and scope. This refinement is vital for improving test efficiency as well as ensures that tests provide early feedback to developers at each stage of the development lifecycle. The refinement aims to remain the coverage rates.

The unit test cases remain unchanged throughout the process, primarily due to their satisfactory performance and the high level of code coverage, currently exceeding 80%. This level of coverage indicates that most of the critical application logic is already being tested effectively. Given the current architectural scale of the application, which has a significant number of components and functions, the corresponding volume of unit tests is sufficient. Therefore, after careful evaluation, the existing unit tests do not introduce any significant shortcomings or inefficiencies that require refactoring.

The test cases refinement plan focuses on separating and refining existing E2E test cases and integration test cases into component tests, integration tests and E2E tests as shown in Figure 9.

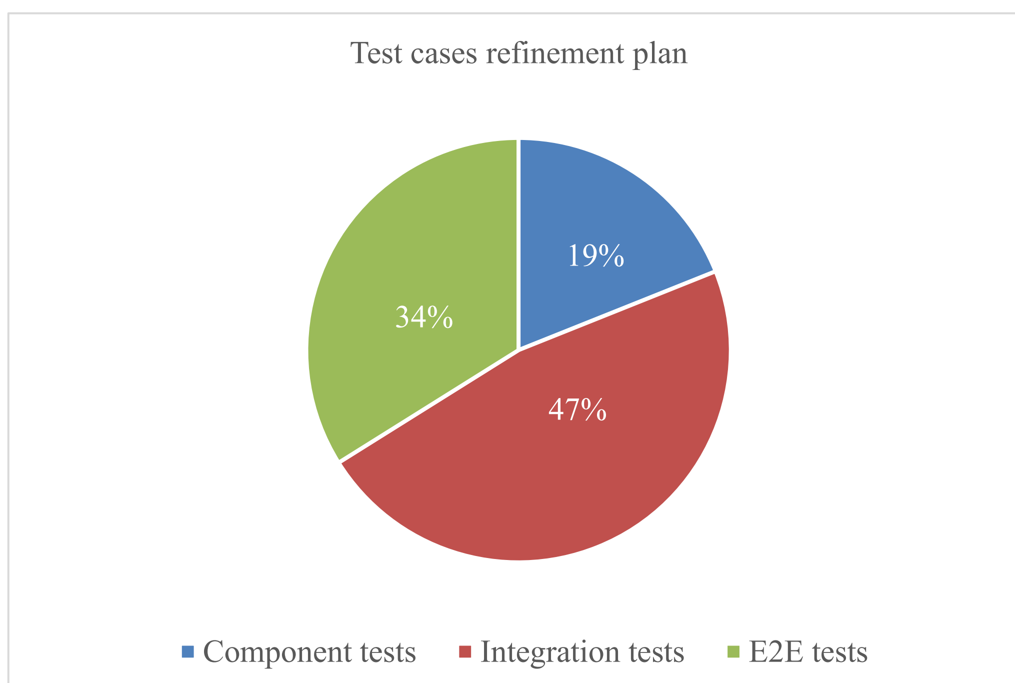

Figure 9. Test cases separation plans.

There are 19% of the existing E2E test cases and integration test cases that are replaced with component tests. Component tests focus on the validation of a small part of the UI behavior which validates both the UI elements and their corresponding functions without requiring complex integration between several components and services. The component tests validate that the component correctly responds to input properties, emits output events, and interacts with its template. There are test cases that validate the interactions between more than one component also considered as component tests. This level of testing is particularly efficient for verifying the UI logic and layout responsiveness without requiring a full application context.

There are about 47% of the existing E2E test cases and integration test cases that are replaced with integration tests. The integration tests verify the interactions between components, services, and modules as well as how applications interact with shared libraries, mocked APIs, and cross-module components. For example, a test ensures that a pop-up is shown when the user clicks a button, then a user consent is checked before sending a request to the backend as well as confirmation for successful response is shown to the user after the process is completed. The integration tests validates whether the component correctly communicates with a shared validation service and dispatches data to an application state management module.

About 34% of the existing E2E test cases stay unchanged. The E2E test cases are essential to simulate the user journey which requires validation of the whole system including UI, APIs as well as the environments that the application is built upon. The scope of the E2E test cases is intentionally limited to cover only critical user paths due to its intensive resource consumption.

5.2.2 Selection of testing tools, frameworks

The monorepo already supports different testing frameworks for Angular applications, such as: Jest, Cypress, and Playwright. The choice of tool for each layer was made deliberately, based on performance needs, testing scope, as well as the needs of specific projects. Cypress and Playwright are mainly used for E2E testing, visual testing in other projects in the monorepo. Cypress is also used for component tests in some projects. However, the project of this case study takes a tailored approach, in which Jest and Angular Testing Framework is used for unit and component testing, Playwright for integration testing, and Cypress for E2E testing. Furthermore, Cucumber is integrated with both Cypress and Playwright to support BDD due to business requirements of the project.

For component testing, one of the primary reasons for choosing Jest and Angular Testing Framework over Cypress is the nature of component testing, which focuses on verifying small, isolated parts of an Angular component’s template and logic. These tests should ideally run without requiring any heavy environment configuration like launching browsers or running against any server. Jest is well integrated with Angular’s testing utilities, such as TestBed. Jest supports mocking and simulating dependencies while TestBed supports the creation and configuration of testing modules, which makes it an effective tool for testing individual components in isolation (Angular, 2025).

Component testing in Cypress, on the other hand, has a different setup. Cypress mounts components in a real browser environment, which offers more visual accuracy and styling verification. Component tests in Cypress involve browser rendering, which is beneficial for visual regressions or styling-related verification. (Cypress, 2025) It is unnecessary when the goal is to validate component logic or interaction with mocked services.

In terms of integration test, Playwright is selected as the testing tool, instead of Cypress. Integration tests in this context aim to verify how multiple components work together in addition to components' interaction with services, backend logic. The integration tests do not require full application behavior validation which includes environment configuration, browser context. Playwright supports automation for Chromium, Firefox, and WebKit browsers and test execution in either headless or headed modes (Playwright, 2025). According to the online article by Swikriti (2025) on the performance evaluation between Playwright and Cypress, Playwright significantly outperforms Cypress in both local and CI environments. In CI pipelines, Playwright completed test execution in approximately 10 seconds, while Cypress took around 17 seconds. The overall pipeline duration was also shorter for Playwright (42 seconds) compared to Cypress (1 minute and 40 seconds). Locally, Playwright was about 42% faster in headless mode and 26% faster in headed mode. Playwright offers significant advantages in speed, efficiency, which is beneficial for performance and time-sensitive CI/CD workflow. (Swikriti, 2025) Therefore, Playwright is an ideal choice for integration testing where a fast feedback loop is needed.

For E2E testing, the project continues to use Cypress. The decision to retain Cypress was made primarily because existing E2E tests were already implemented using Cypress, and these tests have proven effective in validating full user workflows with specific browser configurations. Migrating these tests to another tool like Playwright, which is a newer framework into the project would result in unnecessary overhead. Moreover, Cypress offers several features for E2E testing, such as interactive browser testing capabilities, video snapshots, debugging tools, and test logs. Cypress offers automatic waits for DOM elements to appear and events to complete before proceeding to the next step, which reduces the need for manual delays and flakiness in tests. With Cypress's interactive test runner, developers can see exactly what happened at each test step, which makes it easier to identify issues quickly. (Cypress, 2025) Therefore, for validating complete user workflows within real browser behavior, Cypress remains as a top-tier tool.

An additional tool integrated with both Cypress and Playwright in this case study is Cucumber, which enables BDD. Cucumber allows test scenarios to be written in a domain-specific language called Gherkin, which uses a natural language syntax to describe user interactions and expected outcomes. Test scenarios are not only executable by the testing framework but also serve as documentation for the system's behavior. The use of Cucumber enhances collaboration between developers, testers, and non-technical stakeholders. (Cucumber, 2025) Due to the technical capability of external tools and business requirements of the project, the usage of cucumber and BDD is needed for test case implementations.

5.2.3 The implementation of the new testing architecture and test execution pipeline

In the monorepo, two separate E2E applications were initially implemented to validate two distinct UI applications within the project. However, in January 2025, a business decision was made to combine the two UI applications into one single application. As a result, it is essential to merge the two E2E testing projects into one, in order to ensure long-term maintainability, reduce duplication. The merging process includes migrating all testing configurations, test scripts, and supporting utilities from E2E Application 2 into E2E Application 1, while ensuring that no regression occurred in test execution or. The two E2E applications were designed to operate in the same environment configuration, which simplifies the setup during the migration. Additionally, test scenarios that were duplicated across both E2E applications were refactored for reusability, and common configurations are unified to support the new single application structure. The consolidation does not only reduce maintenance overhead but also provides a clearer view of test coverage for the entire user workflow in one place.

In addition, a new test project was initiated to support integration testing using Playwright. The Playwright project was set up from scratch and integrated with Cucumber to enable BDD. The existing E2E tests that are replaced with integration tests are implemented using mock backend responses as full API availability is not necessary. Some of the test steps are refactored to have common step definitions to avoid duplication. Moreover, as the integration tests are maintained separately from E2E tests, they serve as a faster layer of feedback in the CI pipeline, catching potential integration issues early in the development lifecycle.

For component testing, in the existing setup of Jest, unit test files follow the conventional naming pattern “*.component.spec.ts” and are located in the same directory as the component they test. The component testing uses Jest and Angular Testing Framework; to improve test isolation and make the test structure more modular and maintainable, it is necessary to have

separate component test direction within the same component directory. This allows for tight cohesion between test logic and component code as several components contain complex logic. These component tests focus specifically on rendering behaviour, DOM updates, and template logic under various mocked states. Additionally, the unit test suite is configured with Jest's code coverage requirement set to a minimum of 80%, this threshold is not enforced for the new component tests. Hence, an additional configuration is added to exclude component test files from coverage reporting, preventing them from impacting the global test coverage metrics.

As a result, new project architecture is introduced in this case study with the integration testing project with Playwright and Cucumber, as well as the refined component testing strategy with Jest. The new architecture is illustrated in Figure 10.

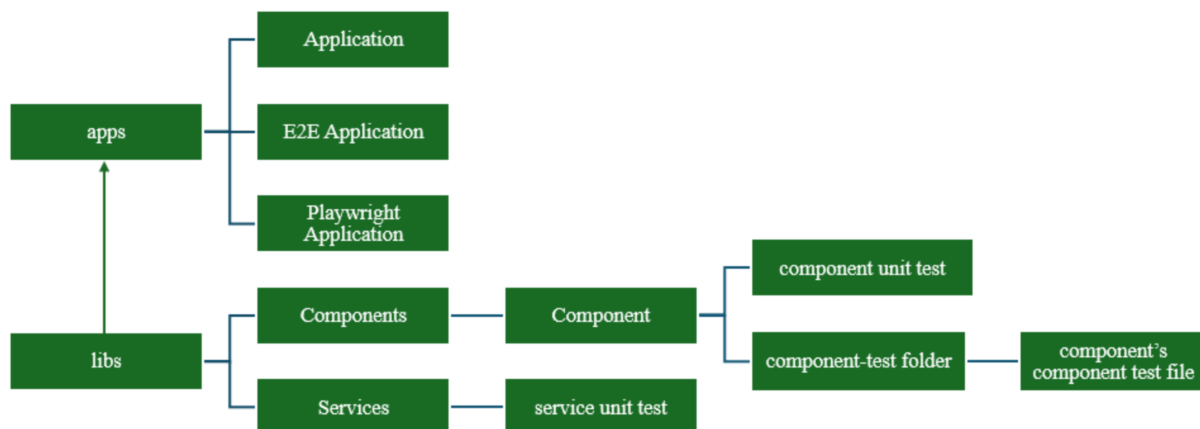

Figure 10. The new architecture of the project of the case study.

In addition to the refactoring and implementation of test cases, a new CI pipeline was developed to streamline and enhance test execution. The existing monorepo already supports a quality check pipeline, which is triggered before merging any code changes into the main branch. This pipeline executes both unit and E2E tests to ensure baseline quality. However, the current setup lacks the flexibility to dynamically target different testing environments, syncing of test cases and test results to external platform, which became a significant limitation during the project verification phase driven by business requirements. In order to address this, a dedicated verification pipeline was introduced as part of the case study implementation. The new pipeline is designed to execute tests layer by layer, supporting execution across multiple environments as needed. The pipeline is implemented using Azure DevOps, which is supported by the DevOps platform of the project. The execution flow follows the testing pyramid, which starts with component tests, followed by integration tests,

and finally E2E tests. The pipeline enforces a fail-fast strategy; if any layer fails, subsequent layers are not executed. This approach accelerates feedback loops, reduces unnecessary compute time, and simplifies debugging by clearly identifying which test layer introduced the failure.

This architecture does not only improve test traceability and maintainability but also aligns with continuous testing principles by validating the application incrementally at each stage of the delivery lifecycle.

5.2.4 Challenges

One of the challenges encountered during the implementation of integration testing authentication handling. In order to validate UI and their interactions with backend services, the UI is required to be in an authenticated state. These tests need to bypass or simulate authentication to allow more focused validation of component and service integration instead of depending on external authentication services. Several approaches were explored, including mocking the API's authentication responses, intercepting login requests. However, due to the application's architecture that depends on token-based session management, these strategies were unsuccessful. As a result, the integration tests are required to perform a full login process before executing any other steps. This dependency on authentication service fails to meet the original goal of integration testing as well as introduces some of the overhead typically associated with E2E testing.

Setting up Playwright with Cucumber for integration testing also introduced additional complexity as Playwright has not been widely adopted in the monorepo. There is limited internal support in terms of technical documentation, shared libraries, and configuration templates. The setup includes initiating and configuring Playwright as a standalone project, integrating it with Cucumber, and customizing scripts for test execution. While Playwright offers several tools and plug-ins, not all of these features were fully utilized due to the constraints from the Cucumber integration. Despite these limitations, the integration of Playwright and Cucumber still provides considerable benefits.

Another challenge during the implementation is the integration of Jest and Cucumber for component testing. While the current component tests successfully meet the technical objectives, which ensure traceability and coverage for UI behavior, they failed to meet the business requirement of adopting BDD practices across all testing layers. Although there are

several open-source libraries and community-supported solutions available for integrating Jest with Cucumber, the monorepo lacks support for this setup. Moreover, due to the project's strict governance and tool validation policies, introducing new packages or libraries into the development workflow requires approval from a formal validation and security review process. Given these constraints, and the project's timeline, it was not feasible to pursue BDD for component testing within the current development phase. As a result, for component tests, using Jest and Angular Testing Framework without BDD syntax is sufficient.

5.3 Case study results

5.3.1 Test execution duration and performance

Since the adoption of multi-layer testing strategy, the execution duration and overall performance of the test automation are considerably improved. According to the test recorded data and systematic observation, no test failures were attributed to flakiness, environmental instability, or infrastructure inconsistencies during the evaluation period. Component tests typically execute at the unit level in the pipeline with the execution time ranges from 1.5 to 2 minutes on average. Figure 11 shows the total execution duration recorded from five test executions of E2E tests and integration tests. All tests were executed under identical conditions, against the same environment, server, and network configuration as those documented in Figure 8, which ensures consistency and comparability of the results.

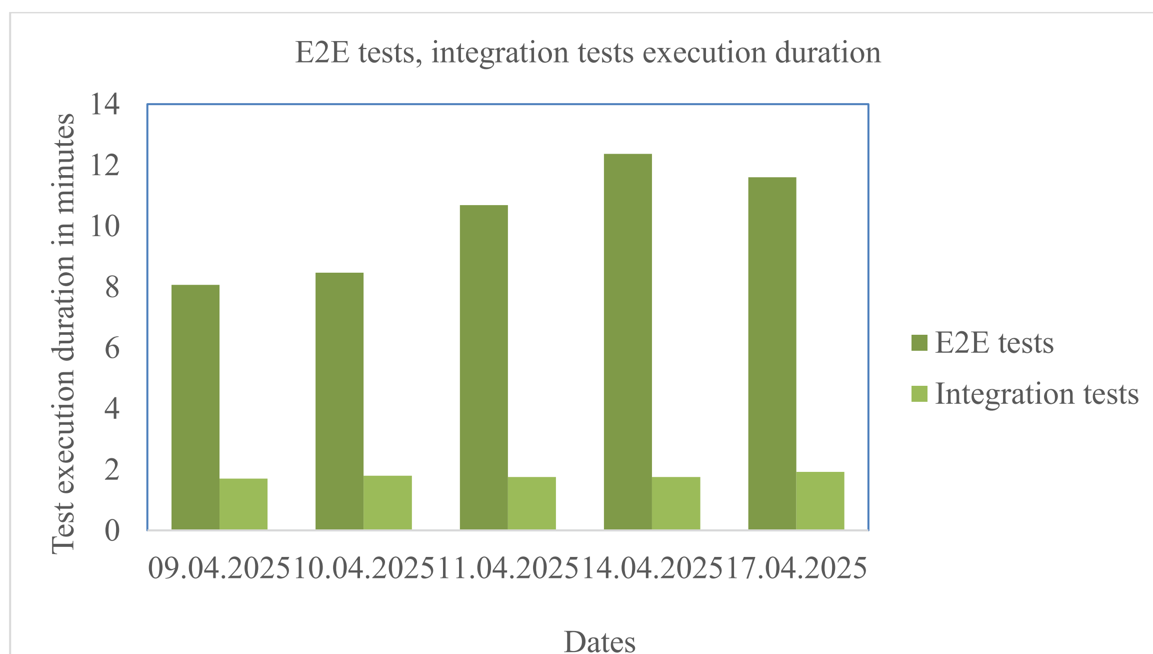

Figure 11. E2E tests, integration tests execution duration since adoption of multi-layer testing strategy.

The test execution is 12 minutes on average for both E2E tests and integration tests, in which the test execution time of integration tests is consistently across all dates, with an average duration of approximately 2 minutes. This significantly contributes to fast feedback within the CI/CD pipeline. The E2E tests, however, represent the most significant portion of the test execution time in this multi-layered strategy with an average of 10 minutes. As E2E tests simulate user workflows and interactions with the full application stack, a longer duration is expected. When including component tests in the overall calculation, the total test suite execution time ranges from 12 to 15 minutes.

5.3.2 Survey results

The survey was conducted as part of a case study focusing on developer's point of view in whether a multi-layered approach effectively improves test automation performance in user interface (UI) testing of Angular applications. The survey targeted developers who are involved in the development of Angular applications in the monorepo. The collected responses provide insights into the practical application and the effectiveness of multi-layered testing architecture.

In terms of concept familiarity, all participants have a moderate to high level of familiarity with 75% have used multi-layered testing strategies in their project. This means they have the basic understandings to evaluate the benefits and challenges of applying such strategies in practice. All participants who are currently working on the studied project, where a multi-layered testing strategy has been adopted, and who are working on other project, reported several improvements in the testing. These include faster bug detection, reduced test flakiness and faster feedback in CI pipeline. The effectiveness of the multi-layered testing approach in improving test automation performance are considered as very effective in improving testing performance in Angular UI projects.

In terms of challenges and limitations, participants identified several challenges associated with multi-layered testing approach. Risk of overlapping test logic is considered as the most challenging. Followed by steep leaning curve and redundant tests across layers. While most of the participants agree that all the test layers are useful, one participant think component test is redundant.

When comparing the multi-layered approach to simpler or single-layered strategies, all participants stated that multi-layered approach is much better to slightly better. As a result,

every respondent indicated that they would recommend adopting a multi-layered testing strategy for UI testing in Angular-based monorepo.

5.4 Findings and discussions

> **RQ3.** Does a multi-layered approach effectively improve test automation performance in user interface testing of Angular applications?

The case study demonstrates that adopting a multi-layered approach to testing Angular applications in monorepo results in considerable improvements of test performance, feedback loop, and maintainability. The test automation strategy was restructured into four layers: unit testing, component testing, integration testing, and E2E testing. There are more than 523 test cases in total, in which 408 test cases are unit tests and component tests, 63 test cases are integration tests and 52 test cases are E2E tests. Integration tests account for 12% of the test suite, which meets the goal of 10-15% integration tests in the system, suggested by Yadavali (2022). Similarly, E2E test cases account for 10% of the whole test suite.

In terms of test execution duration, component tests which are executed at the unit level, completed in an average of 1.5 to 2 minutes. Integration tests follow closely with an average execution time of approximately 2 minutes, consistently across all executions. The E2E tests, though the most time-consuming, average 10 minutes. Together, the total test suite execution time ranged between 12 and 15 minutes, which is slightly faster than before (19 minutes on average).

Additionally, the refactoring of test cases and introduction of new testing tools also contribute to the overall optimization. The significant improvement in execution time of integration tests indicates that the test autmation framework plays a crucial role in testing strategy, as Playwright is used for integration tests instead of Cypress. Jest and Angular Testing Framework were used for fast and isolated component testing instead of Cypress which was supported as component testing framework in other projects of the monorepo. Cypress was retained for E2E testing due to its mature setup, effective debugging tools, and compatibility with existing test cases.

In addition, according to the developer survey, the multi-layered testing approach effectively improves the performance of the test execution. All respondents noted enhanced test reliability, improved bug detection at multiple stages of development, and clearer test

structure. Potential benefits such as increased modularity, improved test coverage and better traceability are acknowledged by the developers who are working in the monorepo.

In summary, the answer for the third research question, whether a multi-layered approach effectively improves test automation performance in user interface testing of Angular applications, is yes. The findings indicate that the multi-layered approach substantially improves the performance, maintainability, and reliability of UI test automation in Angular applications. It enables faster feedback loops, reduces execution times, and supports better testing performance.

> **RQ4.** What challenges are associated with its implementation in user interface testing of Angular applications?

According to the test case refinement plan, the number of E2E tests remains relatively high and nearly equal to the number of integration tests, as many E2E tests were retained due to specific business requirements and the underlying architecture of the project. This highlights how business and system requirements can significantly influence the design and structure of the test architecture. For example, test cases involving complex workflow that require full validation of the whole system.

Another challenge is the technical limitations and tooling constraints. For example, the inconsistent BDD implementation across layers. While both E2E and integration tests adopted BDD syntax using Cucumber, component tests, implemented with Jest and Angular Testing Framework, do not use BDD. This was due to tooling limitations and strict project governance policies. As a result, the component test layer did not meet the business requirement of BDD compliance, introducing inconsistency in how test cases were written and understood. This inconsistency makes it hard to maintain a unified testing strategy across the project as well as collaboration between technical and non-technical stakeholders.

From a maintenance perspective, developers noted that the multi-layered structure introduced a steeper learning curve. While the layered approach brought clarity in test responsibility, it also required developers to have a deep understanding of what is covered at each layer to avoid redundancy.

The findings indicate that while multi-layered testing offers a valuable framework for structuring UI automation, its success depends on alignment between technical goals and business priorities, as well as sufficient tooling support of the project.

6 Conclusions

The thesis has successfully studied the role and effectiveness of a multi-layered test automation strategy in the context of user interface testing for Angular applications of complex systems. Based on the combination of literature review and empirical evaluation of the case study, the study has demonstrated that unit testing, component testing, integration testing, and E2E testing play a potential role in building a comprehensive testing architecture. The conclusions corresponding to each research question examined in this thesis are discussed below.

> **RQ1.** What are the differences between layers in testing architecture: component testing and unit testing, integration testing and E2E testing?

The comparative analysis indicates that unit and component tests are beneficial for early-stage validation by focusing on isolated logic and component interactions. These tests are characterized by fast execution times and lower maintenance costs. Integration and E2E tests provide broader system-level validation, ensuring the correct interaction between modules, services, and are essential for verifying complete workflows and capturing defects when integrating multiple components or services. Each testing layer plays a distinct and complementary role in enhancing the performance and quality of the software.

> **RQ2.** What are the benefits of a multi-layered testing approach in user interface testing of Angular applications?

The literature review on multi-layered testing architecture in Angular application highlights the potential benefits of a well-defined test automation architecture in software development in terms of maintainability and scalability. Distributing testing responsibilities across appropriate layers reduces the issues of flakiness and extensive test execution time. Moreover, it supports early detection of defects, reduces the cost of catching issues late in the development cycle. Overall, the multi-layered testing strategy allows the development team to efficiently isolate problems at the unit and component level, while ensuring complete system functionality through targeted integration and end-to-end testing.

> **RQ3.** Does a multi-layered approach effectively improve test automation performance in user interface testing of Angular applications?

The results from the case study emphasize the feasibility and benefits of a multi-layered testing strategy within a monorepo. The approach demonstrates measurable improvements in execution performance, with overall test durations reduced from 19 minutes to an average of 12–15 minutes. The adoption of Jest and Angular Testing Framework for component testing, Playwright for integration testing and Cypress for E2E testing optimizes test execution within CI pipelines. This highlights the importance of tooling choices in designing test automation strategies.

> **RQ4.** What challenges are associated with its implementation in user interface testing of Angular applications?

Several implementation challenges are identified as well. For example, inconsistencies in BDD syntax support across tools, and steep learning curve associated with managing multiple test layers. Therefore, the multi-layered approach to UI test automation in Angular applications is both effective and sustainable when appropriately tailored to the technical context and business requirements of the project.

7 Recommendations for future work

Based on the findings and limitations identified in this study, several directions for future research and development are proposed to further enhance the effectiveness and scalability of UI test automation in Angular applications.

While the thesis evaluates test performance based on execution time, distribution of test types, more precise metrics, such as test failure rates, defects detection efficiency, and the frequency of test flakiness over time, for test accuracy and effectiveness are recommended. Moreover, the stability of environment, network variability, which potentially affect the performance of E2E tests, should be considered in future assessments. Additionally, comparative analysis, such as the implementation of identical test suites across different testing frameworks could provide a better understanding of the strengths and trade-offs of selected test automation frameworks. The comparisons are essential for the design of test automation strategies. Incorporating these metrics and comparative analyses would provide a more objective and comprehensive evaluation on the impact of multi-layered testing approach in software quality assurance.

Moreover, with the increased use of artificial intelligence in software development, AI-driven tools can assist in several aspects, including automated test case generation, and flakiness detection. Therefore, in addition to available testing frameworks, AI is considerable to be used as assisting tools in test case generation for developers. Future research can explore the integration of AI-augmented tools within the Angular ecosystem to assess their effectiveness, and maintainability in test automation development pipelines. These efforts can contribute to building more adaptive and sustainable testing infrastructures for modern web application development.

References

Ahmed, B. S., Sahib, M. A., & Potrus, M. Y. 2014. Generating combinatorial test cases using Simplified Swarm Optimization (SSO) algorithm for automated GUI functional testing. Engineering Science and Technology, an International Journal. Vol. 17, no.4, p.218–226.

Al-Saqqa, S., Sawalha, S., & Abdel-Nabi, H. 2020. Agile software development: Methodologies and trends. International Journal of Interactive Mobile Technologies. Vol. 14, no. 11, p.246–263.

Angular. 2025. Angular Documentation. Accessed January 2025. Available: https://angular.dev.

Angular Adventurer. 2025. Managing multiple Angular applications and shared code: A complete guide (Part 3). Medium. Accessed March 2025. Available: https://medium.com/@angular-adventurer/managing-multiple-angular-applications-and-shared-code-a-complete-guide-part-3-156ac5be8320

Balsam, S., & Mishra, D. 2024. Web application testing—Challenges and opportunities. Journal of Systems and Software. Elsevier Inc. Vol. 219.

Bastidas, W. 2023. Angular testing made easy: Understanding the testing pyramid and focusing on unit testing 2023. Medium. Accessed February 2025. Available: https://medium.com/williambastidasblog/angular-testing-made-easy-understanding-the-testing-pyramid-and-focusing-on-unit-testing-2023-b4feb8ce5d8e

Björkman, M. 2023. Software Test Automation: Implementation of End-to-End Testing in Web Applications (Master's thesis, Umeå University).

Brito, G., Terra, R., & Valente, M. T. 2018. Monorepos: A Multivocal Literature Review. Federal University of Minas Gerais & Federal University of Lavras.

Capgemini, Sogeti, & Broadcom. 2020. Continuous Testing Report 2020. Capgemini. Accessed January 2025. Available: https://www.capgemini.com/de-de/wp-content/uploads/sites/5/2020/03/Report_Continous_Testing_2020_Sogeti_Capgemini.pdf

Chasidim, H., Almog, D., Sohacheski, D. B., Gillenson, M. L., & Poston, R. 2018. The unit test: Facing CI/CD – Are they elusive definitions? Journal of Information Technology Management.

Cucumber. 2025. Cucumber. Accessed April 2025. Available: https://cucumber.io/docs/ .

Cypress docs. 2025. Cypress.io. Accessed April 2025. Available: https://docs.cypress.io/.

Daka, E., & Fraser, G. 2014. A survey on unit testing practices and problems. IEEE 25th International Symposium on Software Reliability Engineering. p.201-211.

Dooley, J. F. 2017. Unit Testing. In Software Development, Design and Coding. Apress, Berkeley, CA.

Farooq, S. U., & Quadri, S. M. K. 2010. Software testing – Goals, principles, and limitations. International Journal of Computer Applications. Vol. 6, no. 9, p.7–9.

Fowler, M. 2019. Refactoring: Improving the design of existing code. Second edition. Addison-Wesley. Pearson Education, Inc.

Jaspan, C., Jorde, M., Knight, A., Sadowski, C., Smith, E. K., Winter, C., & Murphy-Hill, E. 2018. Advantages and Disadvantages of a Monolithic Repository: A Case Study at Google. ACM/IEEE 40th International Conference on Software Engineering: Software Engineering in Practice.

Gao, J. 2000. Component testability and component testing challenges. San Jose State University Technical Report.

Garousi, V., & Elberzhager, F. 2017. Test automation: Not just for test execution. IEEE Software. Vol. 34, no. 2, p.90-96.

Golian, N. V., Golian, V. V., & Afanasieva, I. V. 2022. Black and white-box unit testing for web applications. Bulletin of the National Technical University. no. 1(7), p.79-88.

Greener, S. 2008. Business Research Methods. Dr Sue Greener & Ventus Publishing ApS.

Imtiaz, J., Sherin, S., Khan, M. U., & Iqbal, M. Z. 2019. A systematic literature review of test breakage prevention and repair techniques. Information and Software Technology. Elsevier B.V. Vol. 113, p.1-19.

International Software Testing Qualifications Board (ISTQB). 2023. Standard Glossary of Terms Used in Software Testing. Version 4.2.

ISO/IEC/IEEE 24765:2010(E). 2010. Systems and software engineering — Vocabulary. International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), and Institute of Electrical and Electronics Engineers (IEEE).

Jose, B. 2021. Test automation: A manager's guide. BCS Learning and Development Ltd.

Kodali, N. 2024. Micro Frontends: A New Paradigm for Scalable Angular Applications. International Journal of Computer Engineering and Technology (IJCET). Vol. 15, no. 5, p.438–449.

Leotta, M., García, B., Ricca, F., & Whitehead, J. 2023. Challenges of end-to-end testing with Selenium WebDriver and how to face them: A survey. IEEE Conference on Software Testing, Verification and Validation (ICST). p.339-350.

Liu, X., Song, Z., Fang, W., Yang, W., & Wang, W. 2024. WEFix: Intelligent automatic generation of explicit waits for efficient web end-to-end flaky tests. Proceedings of the ACM Web Conference 2024. Accessed January 2025. Available: dl.acm.org/doi/pdf/10.1145/3589334.3645628.

Mollah, H., & van den Bos, P. 2023. From user stories to end-to-end web testing. IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). p.140-148.

Myers, G. J., Badgett, T., & Sandler, C. 2011. The art of software testing. Third Edition. John Wiley & Sons, Inc.

Nx. 2025. Nx. Accessed April 2025. Available: https://nx.dev/.

Palmer, J., Cohn, C., & Morearty, M. 2018. Testing Angular applications. Manning Publications.

Petlovyi, V. 2024. Comparative Analysis of End-to-End Testing Tools in Angular Web Development (Bachelor's thesis, Estonian Entrepreneurship University of Applied Sciences).

Ramzan, M. 2024. Adoption of Monorepo Architecture in Web Application Development: Opportunities and Challenges (Master's thesis, Tampere University)

Ricca, F., & Stocco, A. 2020. Web test automation: Insights from the grey literature. 47th International Conference on Current Trends in Theory and Practice of Computer Science. p.1-13.

Saunders, M., Lewis, P. & Thornhill, A. 2012. Research Methods for Business Students. 6th edition. Harlow, England: Pearson Education Limited.

Sawant, A. A., Bari, P. H., & Chawan, P. M. 2012. Software testing techniques and strategies. International Journal of Engineering Research and Applications. Vol. 2, no. 3, p.980–986.

Schäfer, M. 2021. Testing Angular applications. 9elements. Accessed February 2025. Available: https://9elements.com/blog/testing-angular-applications.

Shah, H. 2024. Navigating UI Engineering Architectures: A Deep Dive into Modern Web Application Design [PDF document]. Rochester Institute of Technology

SmartBear. 2025. The Complete Guide to End-to-End Testing. Accessed January 2025. Available: https://spring2019.stpcon.com/wp-content/uploads/2018/12/SB_EBK_End-to-End-Testing.pdf.

Spillner, A., Linz, T. & Schaefer, H. 2014. Software testing foundations: A study guide for the certified tester exam (4th ed.). Rocky Nook.

Swikriti, T. 2025. Comparing Test Execution Speed of Modern Test Automation Frameworks: Cypress vs Playwright. Dev.io . Accessed April 2025. Available: https://dev.to/swikritit/comparing-test-execution-speed-of-modern-test-automation-frameworks-cypress-vs-playwright-3hg8 .

Wang, Y., Mäntylä, M. V., Liu, Z., Markkula, J., Raulamo-jurvanen, P. 2022. Improving test automation maturity: A multivocal literature review. Software Testing, Verification & Reliability. John Wiley & Sons Ltd.

Appendices

Appendix 1 Survey questions

1. How long have you been working with the monorepo?

- Less than a year
- 1–3 years
- More than 3 years

2. Which testing method(s) do you use in your Angular applications? (Select all that apply)

- Unit test
- Component test
- Integration test
- End-to-end test
- Manual testing only
- Other________

3. What is your level of familiarity with multi-layered testing strategies?

(Scale from 1 to 5)

4. Do the projects or applications you are working on apply multi-layered testing strategies?

- Yes
- No

5. We have implemented multi-layered testing strategy in Autoplan project, how effective do you think the multi-layered testing approach is in improving test automation performance?

- Very effective
- Somewhat effective
- Neither effective nor ineffective

- Somewhat ineffective
- Very ineffective
- I am not working in the project

6. If you are working in the Autoplan project, what improvements have you observed after the adoption of the multi-layered testing approach?

(If you are not working in Autoplan project, what are the benefits do you think the multi-layered testing approach can bring to the development of the UI?)

- Faster bug detection
- Reduced test flakiness
- Better test coverage
- Quicker feedback in CI/CD
- No noticeable improvement
- Other

7. In your opinion, do any layers feel redundant or unnecessary?

- Unit test
- Component test
- Integration test
- E2E test
- None – all are useful

8. In your opinion, what are the biggest challenges of multi-layered testing approach?

- High maintenance cost
- Redundant tests across layers
- Slow execution time

- Steep learning curve
- No significant challenges
- Risk of overlapping test logic, e.g., a scenario partially tested in both integration and E2E layers
- Other

9. Compared to simpler strategies, in your opinion, how does multi-layered testing perform?

- Much better
- Slightly better
- About the same
- Slightly worse
- Much worse

10. Would you recommend a multi-layered approach for UI testing in the monorepo?

- Yes
- No
- Maybe